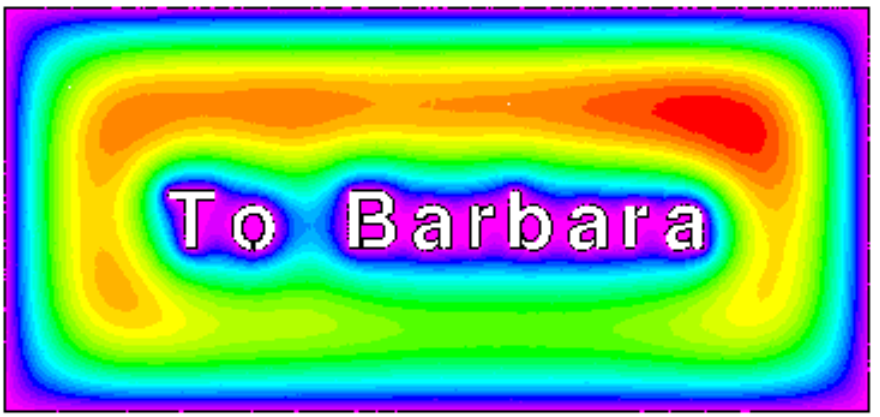
PLTMG:
A Software Package for Solving Elliptic Partial Differential Equations
Users' Guide 9.0

Randolph E. Bank

Department of Mathematics
University of California at San Diego
La Jolla, California 92093-0112

March, 2004

This work was supported by the National Science Foundation
under grants DMS-0208449 and SCREMS-0112413.


To Barbara

Contents

Preface

Many people have made contributions to the development of this version of *PLTMG*; I am indebted to them all for their help. The original grid refinement algorithms used in *PLTMG* were derived in 1976 as joint work with Todd Dupont of the University of Chicago. The approximate Newton strategies incorporated in the present version of *PLTMG* represent joint work with Donald J. Rose of Duke University. The gradient recovery and a posteriori error estimation procedures are joint work with Jinchao Xu of Pennsylvania State University. The algorithms used in the pseudo-arclength continuation procedures are joint work with Tony Chan of the University of California at Los Angeles and Hans Mittelmann of Arizona State University. The interior point algorithms used in the three new optimization problems treated in this version are joint work with Philip Gill of University of California at San Diego. The adaptive mesh smoothing algorithms and the multigraph iterative procedures are joint work with R. Kent Smith of Agere. The X-Windows interface and many of the graphics enhancements were jointly developed with Michael Holst of the University of California at San Diego. The parallel adaptive paradigm is joint work with Michael Holst. The parallel domain decomposition solver is joint work with Shaoying Lu, and the dual function used for parallel adaptive meshing is joint work with Jeffrey Ovall, Both were UCSD graduate students. Many people made contributions to the test problems, reported bugs and suggested improvements that have been incorporated in the current version.

This version of *PLTMG* was supported by the National Science Foundation through grant DMS-0208449 (University of California at San Diego). The UCSD Scicomp Beowulf cluster was built using funds provided by the National Science Foundation through SCREMS grant 0112413, with matching funds from the University of California at San Diego.

University of California at San Diego Randolph E. Bank
March, 2004

Chapter 1
Introduction

1.1 Problem Specification.

Consider the elliptic boundary value problem

$$-\nabla \cdot a(x, y, u, \nabla u, \lambda) + f(x, y, u, \nabla u, \lambda) = 0 \qquad \text{in } \Omega, \tag{1.1}$$

with boundary conditions

$$\begin{aligned} u &= g_2(x, y, \lambda) && \text{on } \partial\Omega_2, \\ a{\cdot}n &= g_1(x, y, u, \lambda) && \text{on } \partial\Omega_1, \\ u, a{\cdot}n \quad &\text{continuous} && \text{on } \partial\Omega_0. \end{aligned} \tag{1.2}$$

Here Ω is a bounded region in $\mathcal{R}^2$, n is the unit normal, a is the vector $(a_1, a_2)^t$, a_1, a_2, f, g_1, and g_2 are scalar functions. $\partial\Omega_0$ is a portion of $\partial\Omega$ where periodic boundary conditions are applied. In some problems solved by *PLTMG*, the parameter λ is not used, while in others $\lambda \in \mathcal{R}$ is a scalar parameter or $\lambda \in \mathcal{H}^1(\Omega)$, where $\mathcal{H}^1(\Omega)$ denotes the usual Sobolev space. Let

$$\begin{aligned} \mathcal{H}^1_p &= \{\phi \in \mathcal{H}^1(\Omega) \,|\, \phi \text{ is continuous on } \partial\Omega_0\}, \\ \mathcal{H}^1_g &= \{\phi \in \mathcal{H}^1_p \,|\, \phi = g_2 \text{ on } \partial\Omega_2\}, \\ \mathcal{H}^1_e &= \{\phi \in \mathcal{H}^1_p \,|\, \phi = 0 \text{ on } \partial\Omega_2\}. \end{aligned}$$

Then the weak form of (1.1)-(1.2) is: find $u \in \mathcal{H}^1_g$ such that

$$a(u, v) = 0 \qquad \text{for all } v \in \mathcal{H}^1_e, \tag{1.3}$$

where

$$a(u, v) = \int_\Omega a(u, \nabla u, \lambda) \cdot \nabla v + f(u, \nabla u, \lambda) v \, dx \, dy - \int_{\partial\Omega_1} g_1(u, \lambda) v \, ds. \tag{1.4}$$

In some problems solved by *PLTMG*, a functional $\rho(u, \lambda)$ plays an important role. Functionals we consider are of the form

$$\rho(u,\lambda) = \int_\Omega p_1(x, y, u, \nabla u, \lambda)\, dx\, dy + \int_\Gamma p_2(x, y, u, \nabla u, \lambda)\, ds, \tag{1.5}$$

where p_1 and p_2 are scalar functions. Here $\Gamma = \partial\Omega \cup \Gamma_0$, where Γ_0 consists of certain internal curves specified by the user.

This version of the *PLTMG* package address five major problem classes. These are briefly described below.

1.1.1 Elliptic Boundary Value Problem.

For this problem, *PLTMG* solves a discrete analog of (1.3). The parameter λ does not play a role in this problem. Let $\mathcal{T}$ denote a triangulation of Ω and let $\mathcal{M}$ be the space of C^0 piecewise linear polynomials associated with $\mathcal{T}$. *PLTMG* usually represents such a piecewise polynomial using the standard nodal basis; a function can then be specified by giving its values at the vertices. Let $\mathcal{I} : \mathcal{H}^1(\Omega) \to \mathcal{M}$ denote continuous piecewise linear interpolation operator that interpolates at the vertices of $\mathcal{T}$. Then

$$\begin{gathered}\mathcal{M}_p = \{\phi \in \mathcal{M} \,|\, \phi \text{ is continuous on } \partial\Omega_0\},\\ \mathcal{M}_g = \{\phi \in \mathcal{M}_p \,|\, \phi = \mathcal{I}(g_2) \text{ on } \partial\Omega_2\},\\ \mathcal{M}_e = \{\phi \in \mathcal{M}_p \,|\, \phi = 0 \text{ on } \partial\Omega_2\}.\end{gathered}$$

The discrete equations solved by *PLTMG* are formulated as follows: find $u_h \in \mathcal{M}_d$ such that

$$a(u_h, v) = 0 \qquad \text{for all } v \in \mathcal{M}_e. \tag{1.6}$$

1.1.2 Obstacle Problem.

The second class of problems addressed by *PLTMG* are the subset of variational inequalities known as obstacle problems. Let

$$\mathcal{K} = \{\phi \in \mathcal{H}^1_g \,|\, \underline{u} \le \phi \le \overline{u}\}.$$

The obstacle problem is formulated as

$$\min_{u \in \mathcal{K}} \rho(u) \tag{1.7}$$

where ρ is a functional of the form (1.5). The parameter λ is not used in this problem. Implicit in our formulation of this problem is an assumption that the Frechet derivative of ρ corresponds to an elliptic boundary problem of the form (1.3). We also assume that the bound constraints are consistent with the boundary conditions.

The discrete form of this problem is as follows. Let

$$\mathcal{K}_h = \{\phi \in \mathcal{M}_g \,|\, \mathcal{I}(\underline{u}) \le \phi \le \mathcal{I}(\overline{u})\}.$$

We then seek $u_h \in \mathcal{K}_h$ that satisfies

$$\min_{u_h \in \mathcal{K}_h} \rho(u_h) \tag{1.8}$$

1.1.3 Continuation Problem.

Continuation problems addressed by *PLTMG* are all of the form (1.3), where the parameter $\lambda \in \mathcal{R}$. Continuation problems also require a functional ρ as in (1.5). Solutions of (1.3)–(1.5) in general define a family of curves on the (λ, ρ) plane. Typical curves are shown in Figure 1.1.

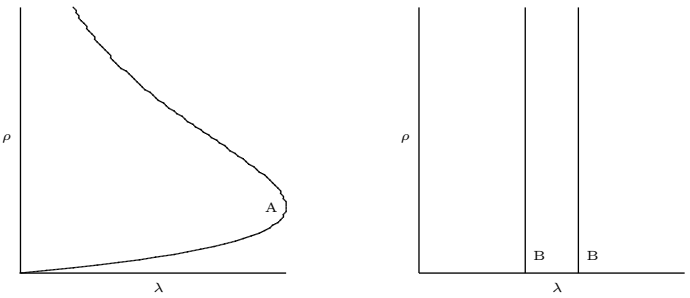


Figure 1.1. *Continuation curves* $\rho = \rho(\lambda)$.

The singular point labeled "A" in the figure on the left is a limit (turning) point, and those labeled "B" in the figure on the right are bifurcation points (this figure corresponds to the special case of a linear eigenvalue problem). The purpose of the continuation process is to compute solutions (u, λ) corresponding to points on these curves.

PLTMG provides a suite of options for solving continuation problems. Among them are options for following a solution curve to a target value in λ or ρ, locating limit and bifurcation points, and switching branches at bifurcation points. Because some problems might have more than one parameter of interest, *PLTMG* also has options for switching parameters and functionals (changing the definitions of λ and ρ) during the calculation, as a means of exploring higher dimensional spaces.

1.1.4 Parameter Identification Problem.

In this problem, a partial differential equation of the form (1.3) appears as a constraint in an optimization problem. Here we seek $\lambda \in \mathcal{R}$ and $u \in \mathcal{H}_g$ that satisfy

$$\min \rho(u, \lambda) \tag{1.9}$$

subject to the constraint (1.3) and the simple bounds

$$\underline{\lambda} \le \lambda \le \overline{\lambda}. \tag{1.10}$$

We define the Lagrangian

$$L(u, v, \lambda) = \rho(u, \lambda) + a(u, v), \tag{1.11}$$

where $v \in \mathcal{H}_e$ is a Lagrange multiplier. We can solve the optimization problem by seeking stationary points of $L(u, v, \lambda)$ constrained by the simple bounds (1.10).

In the discretized problem, we seek $u_h \in \mathcal{M}_g$, a discrete Lagrange multiplier $v_h \in \mathcal{M}_e$, and $\lambda_h \in \mathcal{R}$ that correspond to a stationary point of $L(u_h, v_h, \lambda_h)$, constrained by the simple bounds

$$\underline{\lambda} \le \lambda_h \le \overline{\lambda}. \tag{1.12}$$

As in the case of continuation problems, a problem of the form (1.9)–(1.10) may involve more than one parameter of interest. At present, *PLTMG* does not allow λ to be a vector of parameters, but it does allow parameter switching (redefining the meaning on λ) during the course of the calculation. Thus one can sequentially minimize (1.9) with respect to one of the parameters, holding the others fixed.

1.1.5 Optimal Control Problem.

This problem is very similar to the parameter identification problem, expect now $\lambda \in \mathcal{H}^1(\Omega)$ (or perhaps some weaker space where pointwise values of (1.14) below are defined). Thus we seek $u \in \mathcal{H}_g$ and $\lambda \in \mathcal{H}^1(\Omega)$ that satisfy

$$\min \rho(u, \lambda) \tag{1.13}$$

subject to the constraint (1.3) and the simple bounds

$$\underline{\lambda}(x, y) \le \lambda \le \overline{\lambda}(x, y) \tag{1.14}$$

for $(x, y) \in \Omega$. As before, we define the Lagrangian

$$L(u, v, \lambda) = \rho(u, \lambda) + a(u, v), \tag{1.15}$$

where $v \in \mathcal{H}_e$ is a Lagrange multiplier. We seek stationary points of $L(u, v, \lambda)$ constrained by the simple bounds (1.14).

In the discretized problem, we seek $u_h \in \mathcal{M}_g$, a discrete Lagrange multiplier $v_h \in \mathcal{M}_e$, and $\lambda_h \in \mathcal{M}$ that correspond to a stationary point of $L(u_h, v_h, \lambda_h)$. constrained by the simple bounds

$$\mathcal{I}(\underline{\lambda}) \le \lambda_h \le \mathcal{I}(\overline{\lambda}). \tag{1.16}$$

1.2 Main Subroutines

The software package consists of six primary subroutines. These main routines and their functions are summarized in Table 1.1. The package uses two basic data structures to specify the domain Ω: the triangulation and the skeleton. Loosely speaking, a triangulation specifies the domain Ω as the union of triangles. A skeleton

specifies the domain as the union of one or more subdomains and requires only a description of the boundary of each subdomain. The user can specify the domain as either a triangulation or a skeleton. Specifying a triangulation generally requires less data only for simple domains that can be triangulated with very few triangles. If the domain has a complicated geometry or has internal interfaces that the user would like the triangulation to respect, then it is usually easier to specify the domain as a skeleton. Both data structures are documented in Chapter 2.

Subroutine	Main Function
TRIGEN	Mesh generation and modification
PLTMG	Solve partial differential equation
TRIPLT	Display solution or related function
INPLT	Display input data
GPHPLT	Display performance statistics
MTXPLT	Display sparse matrix

Table 1.1. *The main subroutines in the package.*

Subroutine *TRIGEN* is mainly concerned with transforming the data structures defining the domain. *TRIGEN* also provides a posteriori error estimates for the solution in the $\mathcal{H}^1(\Omega)$ and $\mathcal{L}^2(\Omega)$ norms. *TRIGEN* provides options for creating triangulation and skeleton data structures, and adaptively modifying the triangulation data structure. *TRIGEN* also provides options for various tasks related to parallel processing, namely partitioning the mesh, broadcasting a given mesh to all processors, reconciling a fine mesh distributed among several processors, and (possibly) collecting a fine mesh from many processors onto just one. *TRIGEN* is documented in Chapter 3.

Subroutine *PLTMG* uses finite element discretizations based on C^0 piecewise linear triangular finite elements and includes algorithms to address each of the problem classes described above. In the case of parallel processing, *PLTMG* includes a domain decomposition solver for each problem class. *PLTMG* is described in detail in Chapter 4.

Subroutine *TRIPLT* provides graphical displays of the solution and other grid functions. Three-dimensional color surface/contour plots with shading and an arbitrary viewing perspective are available. Subroutine *INPLT* provides a graphical display of the mesh data (triangulation or skeleton) defining Ω. Subroutine *GPHPLT* provides a variety of graphical displays of convergence histories, statistical data, and other interesting output from *PLTMG*. Subroutine *MTXPLT* displays the stiffness matrix A or the (approximate) LDU factorization of A in a graphical format. These routines are described in detail in Chapter 5.

An elementary interactive test driver, *ATEST*, is described in Chapter 6. *ATEST* provides options for calling each of the main routines, as well as other useful functions such as writing and reading data files, resetting parameters, and executing problem specific subroutines provided by the user. Several short machine dependent routines are required for timing and graphics. These are also described in

Chapter 6. In Chapter 7, the example problem data sets included with the source code are briefly described.

PLTMG was originally conceived as a prototype program to study the theoretical and practical aspects of the multigrid iterative method, adaptive grid refinement and error estimation procedures, and their interaction. As such, *PLTMG* was designed to (formally) handle a wide class of elliptic operators and reasonably general domains. The boundary of the problem class has expanded as problems were encountered that required its enlargement to be solved. The problem class addressed by this version of *PLTMG* should not be interpreted as the limit of the class of problems that could be successfully solved by the techniques embodied by this package. Conversely, one should not assume that every problem (formally) within this class can be solved using the existing code.

As with other versions of the package, time efficiency is a secondary consideration to robustness, versatility, and ease of maintenance. While *PLTMG* is probably not the fastest code that could be used for any particular problem, we believe that it will deliver reasonable execution times in most environments.

1.3 Installation.

The package is provided in both single and double precision versions. The code development was done in single precision, and the program *S*2D of Jim Meyering (available from Netlib) was used to create the double precision version. The source code is contained in several files as indicated in Table 1.2. The majority of the source code is machine independent. The X-Windows interface is based on the Motif widget set and can be used only on systems which support X-Windows. Certain X-Windows libraries must be loaded along with the *PLTMG* software. The OpenGL graphics program *SG* of Michael Holst has been integrated as one of several available graphics devices. *SG* is available elsewhere, and its *MALOC* library must be loaded along with the *PLTMG* software. Finally, the parallel processing options in *PLTMG* are based on MPI, and the MPI library must also be loaded in order to resolve all external names.

File	Contents
pltmg.f	most source code
mgmpi.f (mgmpi_stubs.f)	MPI interface
mgvio.f (mgvio_stubs.f)	SG interface
xgui.c (xgui_stubs.c)	X-Windows interface
mgxdr.c	XDR interface
atest.f	test driver program
burger.f, battery.f, circle.f, control.f domains.f, ident.f, jcn.f, message.f mnsurf.f, naca.f, ob.f, square.f	test problem data sets

Table 1.2. *Files in the basic distribution.*

In MPI is not available or not desired, one can substitute the supplied *stub* interface routines. The stub routines are a set of MPI interface routines with all calls to MPI library functions and subroutines deleted. By using the stub routines in place of the regular interface, one can create an executable with no unresolved external references without loading the MPI library. In this case, however, all the parallel options of *PLTMG* are disabled.

In a similar fashion, if *SG* is not available or not desired, one can use the stub routines in place of standard interface routines. If the stub routines are used, the *MALOC* library is not needed, but the *SG* OpenGL and BH file graphics devices are disabled. Finally, if the X-Windows libraries are not available, one can replace the X-Windows interface with stub routines. In this case, the graphical user interface and the corresponding X-Windows graphics devices are all disabled, but the X-Windows libraries are not needed.

Chapter 2
Data Structures

2.1 Overview.

In this chapter, we define the data structures used in the *PLTMG* package. We begin with the two data structures used to define the region Ω: the triangulation and the skeleton. In this version of *PLTMG*, the triangulation and skeleton data structures have been extended to accommodate the demands of parallel computation. Triangulation and skeleton data structure definitions are summarized in Table 2.1 and described in detail in Sections 2.2 and 2.3.

The arrays *IP*, *RP*, and *SP* contain many scalar parameters, switches, control variables, flags, and pointers, some that must be specified by the user and others that are internally computed but may be of interest to the user. These are described in Section 2.4. Finally, the coefficient functions defining the differential operator and functional ρ in (1.1)–(1.3), and the optional function *QXY* used by *TRIGEN* and *TRIPLT*, are described in Section 2.5.

2.2 The Triangulation.

In this section, we define the triangulation data structure. Let $\mathcal{T}$ denote the triangulation consisting of triangles t_i, $1 \le i \le NTF$, vertices v_i, $1 \le i \le NVF$, and edges b_i, $1 \le i \le NBF$. Triangles may have curved edges, which are approximated by arcs of circles. The centers of the circles are given by c_i, $1 \le i \le NCF$. Curved edges may be on the boundary or in the interior of the region Ω.

For example, consider the circle of radius one with a crack along the positive x-axis. This domain can be triangulated using 8 triangles, 10 vertices, and 10 boundary edges, 8 of which are curved, as illustrated in Figure 2.1. Vertices v_2 and v_{10} have the same (x, y) coordinates, but v_2 is "above" the crack and v_{10} is "below." Similarly, edge b_1 is the top of the crack, while edge b_{10} is the bottom. The ordering of vertices, triangles, and edges is arbitrary.

The arrays *VX* and *VY* are of length *NVF* and contain as their Ith entries the (x, y) coordinates of v_I, illustrated for this example in Table 2.2. If a triangle

array	definition
(VX(I),VY(I))	vertex coordinates
(XM(I),YM(I))	circle center coordinates
IBNDRY(1,I)	first endpoint number
IBNDRY(2,I)	second endpoint number
IBNDRY(3,I)	circle center number
IBNDRY(4,I)	edge type
IBNDRY(5,I)	reserved for parallel processing
IBNDRY(6,I)	edge label
ITNODE for triangulation	
ITNODE(1,I)	first vertex number
ITNODE(2,I)	second vertex number
ITNODE(3,I)	third vertex number
ITNODE(4,I)	reserved for parallel processing
ITNODE(5,I)	element label
ITNODE for skeleton	
ITNODE(1,I)	first vertex number
ITNODE(2,I)	first edge number
ITNODE(3,I)	congruent region number
ITNODE(4,I)	reserved for parallel processing
ITNODE(5,I)	region label

Data structure definitions.

IBNDRY(4,I)	edge type
2	Dirichlet boundary
1	natural boundary
0	internal
$-K$	linked with edge K
3, 4, 5	reserved for parallel processing

Edge type definitions.

Table 2.1.

has a curved edge, that edge is approximated by a circular arc passing through the endpoints of the edge, with the center of the circle located at one of the points c_i. Because there are generally two such arcs for every pair of endpoints, the shorter arc is taken to be the correct edge; therefore, one must specify curved edges that subtend (strictly) less than π of arc; $\pi/4$ is a reasonable upper bound. The centers of the circles used to specify curved edges are given in the arrays *XM* and *YM* of length *NCF*, which contain as their Ith entries the x and y coordinates of the center c_I. The data for our example is shown in Table 2.2.

To simplify data entry, we provide the routine *CENTRE* for computing the center of a circle given three points on its boundary. *CENTRE* is called using the

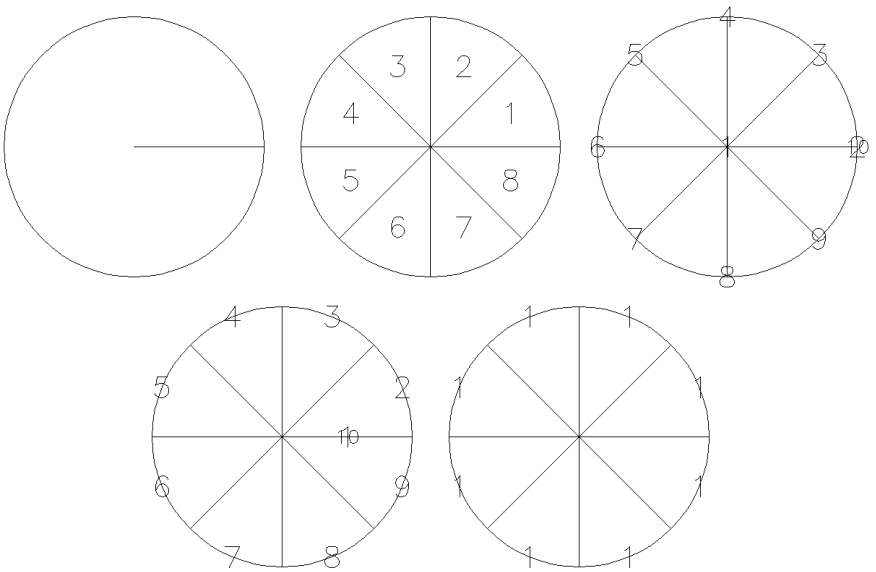


Figure 2.1. *Clockwise, from upper left: example domain; triangle numbers; vertex numbers; curved edges; edge numbers.*

statement

Call CENTRE(X1, Y1, X2, Y2, X3, Y3, XC, YC)

Here *(X1,Y1)* and *(X2,Y2)* are the endpoints of an arc of the circle, and *(X3,Y3)* is a third point on the arc (e.g., the midpoint). *CENTRE* returns the center of the circle in *(XC,YC)*.

A given triangle $t_I \in \mathcal{T}$ is specified by giving an accounting of its three vertices and by specifying an integer label or tag. Such labels are provided strictly for the convenience of the user and can be used to identify differing regions or material properties associated with the element. The array *ITNODE* is a $5 \times NTF$ integer array whose Ith column contains information about t_I. The first three entries of *ITNODE* contain the three vertex numbers of triangle t_I. *ITNODE(J,I)* $= K$, for $1 \le J \le 3$, means *(VX(K),VY(K))* is the Jth vertex of t_I. The ordering of the vertices of a given triangle is arbitrary and independent of the other triangles.[1] Entry *ITNODE(4,I)* is used internally by *PLTMG* in parallel processing, denoting the processor that "owns" t_I; one can simply initialize *ITNODE(4,I)* $= 0$. Entry *ITNODE(5,I)* contains any user provided label for t_I. In our example, we choose to label each element by the quadrant in the Euclidean plane in which it lies. The *ITNODE* array for our example is shown in Table 2.2.

The array *IBNDRY* is a $6 \times NBF$ integer array whose Ith column contains

[1] *PLTMG* reorders vertices as necessary to ensure a counterclockwise orientation for elements.

I	1	2	3	4	5	6	7	8	9	10
$VX(I)$	0	1	$1/\sqrt{2}$	0	$-1/\sqrt{2}$	-1	$-1/\sqrt{2}$	0	$1/\sqrt{2}$	1
$VY(I)$	0	0	$1/\sqrt{2}$	1	$1/\sqrt{2}$	0	$-1/\sqrt{2}$	-1	$-1/\sqrt{2}$	0
$XM(I)$	0									
$YM(I)$	0									

The VX, VY, XM and YM arrays. $NVF = 10$ *and* $NCF = 1$.

I	1	2	3	4	5	6	7	8	9	10
IBNDRY(1,I)	1	2	3	4	5	6	7	8	9	10
IBNDRY(2,I)	2	3	4	5	6	7	8	9	10	1
IBNDRY(3,I)	0	1	1	1	1	1	1	1	1	0
IBNDRY(4,I)	2	2	2	2	2	2	2	2	2	1
IBNDRY(5,I)	0	0	0	0	0	0	0	0	0	0
IBNDRY(6,I)	2	0	0	0	0	0	0	0	0	1

The IBNDRY array. $NBF = 10$.

I	1	2	3	4	5	6	7	8
ITNODE(1,I)	1	1	1	1	1	1	1	1
ITNODE(2,I)	2	3	4	5	6	7	8	9
ITNODE(3,I)	3	4	5	6	7	8	9	10
ITNODE(4,I)	0	0	0	0	0	0	0	0
ITNODE(5,I)	1	1	2	2	3	3	4	4

The ITNODE array. $NTF = 8$.

Table 2.2. *Data structures for a triangulation.*

information the indices of the endpoints of the interval. *IBNDRY(J,I)* $= K$, $1 \le J \le 2$, means *(VX(K),VY(K))* is an endpoint of b_I. Ordering of vertices is arbitrary.[2] Entry *IBNDRY(3,I)* contains the index for the circle center for the edge. *IBNDRY(3,I)* $= K$ means *(XM(K),YM(K))* is the circle center for edge b_I. If the edge is straight, *IBNDRY(3,K)* $= 0$.

The fourth entry defines the boundary type for edge b_I. The possibilities for edge type are shown in Table 2.1. There are several reasons to include internal edges (*IBNDRY(4,I)* $= 0$) in a triangulation. First, if the internal edge is curved, it must be specified in *IBNDRY* in order to be treated properly. Second, the set Γ in equation (1.3) taken as the edge set in *IBNDRY*; thus internal edges which are part of Γ must be defined in *IBNDRY*. An important restriction on internal edges of a triangulation is that they must lie on an internal interface. That is, the two triangles sharing b_I must have different labels as their fourth entries in *ITNODE*.

The fourth type of edge is a linked edge. Linked edges occur only in pairs. If b_I

[2] *PLTMG* orders the vertices of boundary edges to correspond to a left-handed (usually counterclockwise, except for holes) traversal of the boundary.

and b_J are a pair of linked edges, then *IBNDRY(4,I)* $= -J$ and *IBNDRY(4,J)* $= -I$. Linked edges b_I and b_J must be geometrically congruent. That is, b_I must be mapped to b_J using a translation and orthogonal rotation. Continuity of the solution u_h and weak continuity of $a \cdot n$ is imposed on linked edge pairs. Thus if b_I and b_J are boundary edges, this is equivalent to imposing periodic boundary conditions. In the course of parallel processing, *PLTMG* creates edges of types $3 - 5$. Entry *IBNDRY(5,I)* is also used internally by *PLTMG* for parallel processing.

Entry *IBNDRY(6,I)* contains an integer label for the edge, similar to *ITNODE(5,*)*; this label can be used to uniquely identify a particular edge, or to associate some property with the edge. The *IBNDRY* array for our example is shown in Table 2.2.

In our example, we impose Dirichlet boundary conditions on the outer boundary of the circle, and also along the top of the crack, and Neumann boundary conditions on the bottom of the crack. The outer boundary of the circle is labeled 0, the top of the crack 2, and the bottom of the crack 1.

In the case of a singular Neumann problem (e.g., $a_1 \equiv u_x$, $a_2 \equiv u_y$, $f \equiv 0$, and $\partial\Omega_1 = 0$ in (1.1)), the solution u is determined only up to an arbitrary constant. To make such a solution unique, the user may select a distinguished vertex where u will be specified. The parameter *IDBC*, where $1 \le IDBC \le NVF$, denotes the vertex number. In other situations, one should set $IDBC = 0$.

2.3 The Skeleton.

The skeleton data structure is often the easiest data structure for the user to specify by hand, especially if the domain has complicated geometry, symmetry, or internal interfaces. In the skeleton data structure, the domain Ω is viewed as the union of NTF simply connected subregions Ω_i, $1 \le i \le NTF$. The regions need not be convex, and the case $NTF = 1$ is not excluded. A shared boundary between two subregions (an internal interface) will be respected by the triangulation process in *TRIGEN*; that is, the interface will be represented as one or more triangle edges in the triangulation.

The boundary of each Ω_i should be a simple closed curve that does not intersect itself. Thus, for example, if Ω has a hole, adding a single cut between the outer boundary and the hole will not be adequate. At least two subregions will be required in this case.

Having decomposed the domain into NTF subregions, we decompose the boundaries of the subregions into NBF edges b_i, $1 \le i \le NBF$. Each edge has two endpoints ν_i^j, $1 \le j \le 2$, and if it is a curved edge, it will have a circle center c_i. All curved edges are approximated by a circular arc as in the triangulation data structure. Curved edges must subtend less than $\pi/2$ of arc. Globally, the vertices are labeled v_k, $1 \le k \le NVF$, and the circle centers are labeled c_k, $1 \le k \le NCF$. The intersection of any two edges should be at most one common endpoint.

As an example, we consider the square region with a hole illustrated in Figure 2.2. In this example, we decompose the region into 2 subregions ($NTF = 2$), using 10 vertices ($NVF = 10$), 12 edges ($NBF = 12$), and 1 circle center ($NCF = 1$)

as shown.

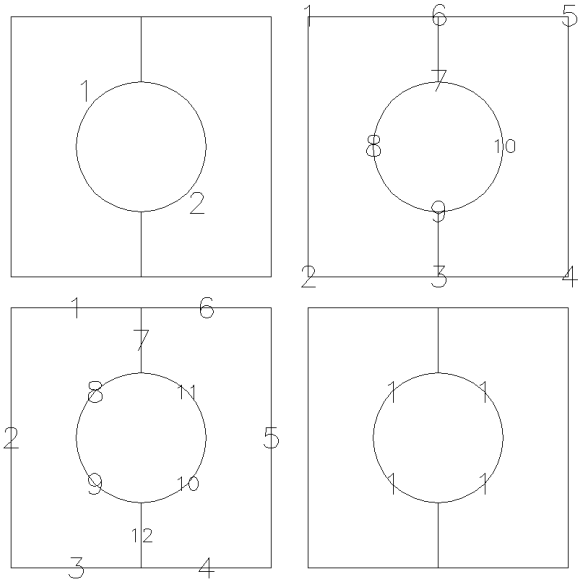


Figure 2.2. *Clockwise, from upper left: example domain decomposed into two subregions; vertex numbers; midpoint numbers; edge numbers.*

Global numbering of the subregions, edges, vertices, and midpoints is arbitrary. The arrays *VX*, *VY*, *XM*, and *YM* have similar definitions for the triangulation and skeleton. These arrays for our example domain are shown in Table 2.3. The (x, y) coordinates of vertex v_k, $1 \le k \le NVF$, are specified in the arrays *VX* and *VY*. The (x, y) coordinates of circle center c_i, $1 \le i \le NCF$, are specified in the arrays *XM* and *YM*.

Edges are specified in *IBNDRY* as in the case of the triangulation. Descendents of Dirichlet, natural, and linked edges are included in the output *IBNDRY* array when Ω is triangulated using *TRIGEN*. Descendents of internal edges are retained only if they separate regions with different labels. Descendent edges inherit the label of the original edge. In our example, we will assign Dirichlet boundary conditions to the left and right sides and the bottom of the domain, and natural boundary conditions elsewhere. The *IBNDRY* array then has the form given in Table 2.3.

A subregion Ω_i, $1 \le i \le NTF$, is defined by an ordered sequence of edges (at least three) that form its boundary. The sequence is ordered such that the boundary of Ω_i is traversed in a counterclockwise direction (thus providing notions of "inside" and "outside"). Each edge in the sequence shares exactly one endpoint with the edge that precedes it and the edge that follows it in the sequence; the first and last edges in the sequence also share one endpoint. A particular edge can appear only once in the sequence.

The array *ITNODE* is used to define the subregions. Column I of *ITNODE*

I	1	2	3	4	5	6	7	8	9	10
VX(I)	-2	-2	0	2	2	0	0	-1	0	1
VY(I)	2	-2	-2	-2	2	2	1	0	-1	0
XM(I)	0									
YM(I)	0									

The VX, VY, XM, and YM arrays. $NVF = 10$ *and* $NCF = 1$.

I	1	2	3	4	5	6	7	8	9	10	11	12
IBNDRY(1,I)	6	1	2	3	4	5	6	7	8	9	7	9
IBNDRY(2,I)	1	2	3	4	5	6	7	8	9	10	10	3
IBNDRY(3,I)	0	0	0	0	0	0	0	1	1	1	1	0
IBNDRY(4,I)	1	2	2	2	2	1	0	1	1	1	1	0
IBNDRY(5,I)	0	0	0	0	0	0	0	0	0	0	0	0
IBNDRY(6,I)	2	1	3	3	1	2	0	4	4	4	4	0

The IBNDRY array. $NBF = 12$.

I	1	2
ITNODE(1,I)	1	4
ITNODE(2,I)	2	5
ITNODE(3,I)	0	1
ITNODE(4,I)	0	0
ITNODE(5,I)	1	2

I	1	2
ITNODE(1,I)	1	5
ITNODE(2,I)	2	6
ITNODE(3,I)	0	-1
ITNODE(4,I)	0	0
ITNODE(5,I)	1	2

The ITNODE array for mapping by rotation (left) and by reflection (right). $NTF = 2$.

Table 2.3. *Skeleton data structures.*

corresponds to the region Ω_I. Entry *ITNODE(1,I)* is a global vertex number for one of the vertices on the boundary of Ω_I. Unless *ITNODE(3,I)* $\neq 0$ (see below) the choice of vertex is arbitrary. The second entry, *ITNODE(2,I)*, is the global edge number of the first edge in a counterclockwise traversal of Ω_I, beginning at vertex v_K, where *ITNODE(1,I)* $= K$.

Entry *ITNODE(3,I)* is used to specify certain symmetries the user may wish to impose on the triangulation. Two subregions are congruent if one can be mapped onto the other using an affine transformation consisting of a translation, an orthogonal rotation, and perhaps a simple reflection. If this mapping also induces one-to-one correspondences between the edges, vertices, and circle centers used to define the regions, then the user can specify that the two regions be triangulated in a similar fashion.

ITNODE(3,I) $= 0$ specifies that Ω_I can be triangulated independently of other regions. *ITNODE(3,I)* $= J$, $0 < J < I$, specifies that Ω_I can be mapped onto Ω_J using just a translation and rotation. *ITNODE(3,I)* $= -J$, $0 < J <$

I, specifies that Ω_I can be mapped onto Ω_J using a translation, rotation, and a reflection. If *ITNODE(3,I)* $= \pm J$, then *ITNODE(1,I)* must correspond to the vertex on $\partial\Omega_I$ which is mapped to the vertex corresponding to *ITNODE(1,J)* on $\partial\Omega_J$. If *ITNODE(3,I)* $\neq 0$, *TRIGEN* will map the triangulation generated for Ω_J onto Ω_I, ensuring the desired symmetry properties of the overall triangulation. Note that this is not a symmetric relation; *ITNODE(3,I)* $= J$ does not mean *ITNODE(3,J)* $= I$. In particular, if $|$ *ITNODE(3,I)* $| \geq I$, *TRIGEN* will return in an error condition.

In our example, Ω_2 can be mapped onto Ω_1 by either rotation or reflection. We can ensure the triangulation for Ω_2 will be similar to that for Ω_1, either under rotation or reflection. The resulting triangulations may be different in the two cases.[3] *ITNODE* arrays for the two situations are illustrated in Table 2.3. Entry *ITNODE(4,I)* is used by *PLTMG* in parallel processing. Entry *ITNODE(5,I)* is a label for the region; all the triangles created in Ω_I inherit this label.

We provide the utility subroutine *SKLUTL* to aid in the creation of the skeleton data structures. Subroutine *SKLUTL* is called using the statement

Call SKLUTL(ISW, VX, VY, XM, YM, ITNODE, IBNDRY, IP, W, IFLAG)

This routine takes as input a skeleton data structure defined *VX*, *VY*, *XM*, *YM*, *IBNDRY*, and *ITNODE*. The integers *NTF*, *NVF*, *NCF*, *NBF* should be specified in the *IP* array. The integer *ISW* specifies the task, as indicated in Table 2.4. *W* is a work array of length *LENW*, whose minimum length depends on *ISW*, but *LENW* $\geq 5(NBF + NVF)$ is sufficient.

ISW	task
0	create *ITNODE* array
1	refine long arcs
2	determine congruent regions

Table 2.4. *The values of ISW.*

If *ISW* $= 0$, *SKLUTL* computes all entries of the *ITNODE* array, given the remaining arrays in the skeleton data structure (*VX*, *VY*, *XM*, *YM*, and *IBNDRY*), and the parameters *NVF*, *NCF*, and *NBF* in the *IP* array. The value of *NTF* is returned in the *IP* array. The regions are labeled with *ITNODE(5,I)* $= I$ for $1 \leq I \leq$ *NTF*, although these labels can subsequently be reset by the user. Also *ITNODE(3,I)* $= 0$ for $1 \leq I \leq$ *NTF*. If *ISW* $= 1$, *SKLUTL* accepts as input a complete skeleton description, and divides curved edges as necessary to ensure that all curved edges subtend less than $\pi/4$ of arc. New edges and vertices are added as necessary, and the relevant skeleton parameters updated. New values of *NBF* and *NVF* are returned in the *IP* array. If *ISW* $= 2$, *SKLUTL* accepts as input a complete skeleton description, and finds congruent regions. The values of *ITNODE(3,I)* (and

[3] We could ensure greater symmetry in the triangulation by decomposing Ω into 4 or 8 congruent regions instead of 2 and then setting *ITNODE(3,I)* appropriately.

possibly *ITNODE(1,I)* and *ITNODE(2,I)*) are reset as necessary. If two regions are congruent but the congruence is not unique, as in our example, an arbitrary choice is made from among the possibilities. Errors are returned in the integer *IFLAG* as described in Table 2.8.

Several other routines in the package check skeleton data structures for common errors in the data. If found, such errors are reported by setting the parameter *IFLAG* as described in Table 2.8.

2.4 Parameter and Work Arrays.

W is a real array of length *LENW*; all internal storage for *PLTMG* and the other routines in the package is allocated from this array. *A* is a real array of length *MAXA*. *JA* is an integer array of length *MAXJA*. These arrays are the main sparse matrix data structures used by the multigraph solver in *PLTMG* [21]. They are used internally and need not be initialized by the user.

IP, *RP*, and *SP* are integer, real, and *CHARACTER*80* arrays, respectively, of length 100 containing various user specified parameters, and internally generated parameters, switches, flags, and pointers. A list of the currently used locations, their names, and brief definitions appears in Tables 2.5–2.7. Parameters marked "u" should be supplied by the user.

The parameter *IFIRST* is an initialization switch. If $IFIRST = 0$, no initialization takes place. If $IFIRST = 1$, the array *W* is partitioned, and the triangulation data structure is checked. The first *MAXV* entries in *W* are allocated to the computed solution *u* ($IUU = 1$), providing the user easy access to the solution. Array entry *IP(25)* is the error flag *IFLAG*. A summary of the possible values for *IFLAG* is given in Table 2.8.

2.5 Coefficient Functions.

Several routines in the package require knowledge of the partial differential equation (1.1), the boundary conditions (1.2), the functional ρ in (1.3), and, on occasion, an alternate function of the solution. This information is provided by the user through subroutines *A1XY*, *A2XY*, *FXY*, *GNXY*, *GDXY*, *P1XY*, *P2XY*, and *QXY*.

Subroutines *A1XY*, *A2XY*, *FXY*, and *P1XY* have identical argument lists.

Call A1XY(X, Y, U, UX, UY, RL, ITAG, VALUES),
Call A2XY(X, Y, U, UX, UY, RL, ITAG, VALUES),
Call P1XY(X, Y, U, UX, UY, RL, ITAG, VALUES),
Call FXY(X, Y, U, UX, UY, RL, ITAG, VALUES).

In these subroutines, all of the arguments except *VALUES* are provided as

I	*IP(I)*	u	definition
1	*NTF*	u	number of triangles / regions
2	*NVF*	u	number of vertices
3	*NCF*	u	number of circle centers
4	*NBF*	u	number of edges
5	*IFIRST*	u	initialization switch
6	*IPROB*	u	problem type
7	*IDBC*	u	special Dirichlet vertex number
8	*ISPD*	u	symmetric / nonsymmetric switch
9	*ITASK*	u	problem task
10	*MXCG*	u	maximum conjugate gradient iterations
11	*MXNWTT*	u	maximum damped Newton iterations
16	*NEVP*	u	number of evaluation points
18	*MAXJA*	u	length of *JA* array
19	*MAXA*	u	length of *A* array
20	*LENW*	u	length of the work array *W*
21	*MAXT*	u	number of columns in the array *ITNODE*
22	*MAXV*	u	length of the arrays *VX* and *VY*
23	*MAXC*	u	length of the arrays *XM* and *YM*
24	*MAXB*	u	number of columns in the array *IBNDRY*
25	*IFLAG*		error flag
26	*IADAPT*	u	mesh generation option switch
27	*IREFN*	u	uniform refinement control
28	*NVTRGT*	u	target value for number of vertices
29	*NRGN*	u	number of contour lines for skeleton
31	*NEWNTF*		number of elements owned by processor
32	*NEWNVF*		number of vertices owned by processor
33	*NEWNBF*		number of edges owned by processor
34	*NVV*		number of interface vertices
35	*NBB*		number of interface edges
36	*NVI*		number of coarse interface vertices
37	*NBI*		number of coarse interface edges
38	*NTG*		global number of elements
39	*NVG*		global number of vertices
40	*NBG*		global number of edges
41	*IUSRSW*	u	*USRCMD* switch
42	*MODE*	u	*ATEST* mode switch
43	*NGRAPH*	u	number of graphics windows

Table 2.5. *IP array definitions.*

I	*IP(I)*	u	definition
44	*FDEVCE*	u	*TRIPLT* graphics device
45	*GDEVCE*	u	*GPHPLT* graphics device
46	*JDEVCE*	u	*INPLT* graphics device
47	*MDEVCE*	u	*MTXPLT* graphics device
48	*MPISW*	u	MPI switch
49	*NPROC*		number of processes
50	*IRGN*		individual process number
51	*MXCOLR*	u	maximum number of colors
52	*IFUN*	u	alternate function switch for *TRIPLT*
53	*INPLSW*	u	alternate graph switch for *INPLT*
54	*IGRSW*	u	alternate graph switch for *GPHPLT*
55	*IMTXSW*	u	alternate matrix switch for *MTXPLT*
56	*NCON*	u	number of contours
57	*ICONT*	u	continuity switch
58	*ISCALE*	u	scale option switch
59	*LINES*	u	line drawing option switch
60	*NUMBRS*	u	numbering option switch
61	*NX*	u	
62	*NY*	u	*(NX,NY,NZ)*
63	*NZ*	u	is the viewing perspective for *TRIPLT*
64	*MX*	u	*(MX,MY,MZ)*
65	*MY*	u	is the viewing perspective for *GPHPLT*
66	*MZ*	u	and *MTXPLT*
67	*LEVEL*	u	matrix level
68	*ICRSN*	u	graphics coarsening switch
69	*ITRGT*	u	target size of graphics mesh
70	*IBASE*		MPI internal edge base number
71	*NVDD*		total number of interface vertices
72	*LIPATH*		length of *IPATH* array
73	*LENJA*		used locations in *JA* array
74	*LENA*		used locations in *A* array
75	*LVL*		number of multigraph levels
76	*NEF*		number of error functions
77	*NGF*		number of grid functions
78	*DBCTAG*		element label associated with *IDBC*
79	*IEVALS*		number of function evaluations on last call
80	*ITNUM*		number of Newton iterations on last call
81	*MAXPTH*		maximum size for *IPATH* array

TABLE 2.5, continued.
IP array definitions.

I	*IP(I)*	u	definition
83	*IUU*		pointer to the solution u
84	*IUX*		pointer to the gradient u_x
85	*IUY*		pointer to the gradient u_y
86	*IU0*		pointer to the initial solution u_0
87	*IUDOT*		pointer to the tangent $\dot{u}$ or vector vx_0
88	*IU0DOT*		pointer to the initial $dotu_0$ or vector vy_0
89	*IUDL*		pointer to the dual solution
90	*IEVR*		pointer to the right singular function ψ_r
91	*IEVL*		pointer to the left singular function ψ_ℓ
92	*JTIME*		pointer to timing array
93	*JHIST*		pointer to convergence history array
94	*JPATH*		pointer to continuation path history array
95	*KA*		pointer to multigraph pointer array
96	*JSTAT*		pointer to parallel statistics array
97	*IEE*		pointer to local error estimates array
98	*IPATH*		pointer to *IPATH* array
99	*IZ*		pointer to temporary workspace

TABLE 2.5, continued.
IP array definitions.

input. In particular $(X, Y) \in \Omega$ is the evaluation (quadrature) point, and

$$\begin{aligned} U &= u_h(X, Y), \\ UX &= \frac{\partial u_h}{\partial x}(X, Y), \\ UY &= \frac{\partial u_h}{\partial y}(X, Y), \\ RL &= \lambda_h, \end{aligned}$$

($RL = \lambda_h(X, Y)$ when $\lambda_h \in \mathcal{S}$). The parameter *ITAG=ITNODE(5,I)* is the user specified label associated with element $\tau_I \in \mathcal{T}$ containing (X, Y). From this input data, the user provides values of the given function and its derivatives in the array *VALUES*. This array is of size 15. All entries are initially set to zero by the calling routine; thus the user need supply only nonzero values.

To simplify this process, PLTMG supplies a labeled common block

common /VAL0/ K0, KU, KX, KY, KL, KUU, KUX, KUY, KUL, KXU,
KXX, KXY, KXL, KYU, KYX, KYY, KYL, KLU, KLX, KLY, KLL

containing a predefined list of integer pointers mapping function and derivative values to particular entries in the *VALUES* array. The details of this mapping are given in Table 2.9 for the case of f; the identical mapping is used for a_1, a_2 and p_1.

I	*RP(I)*	u	definition
1	*RLTRGT*	u	target value for λ
2	*RTRGT*	u	target value for $\rho(u,\lambda)$
3	*RMTRGT*	u	target value for μ
4	*RLLWR*	u	lower bound for λ
5	*RLUPR*	u	upper bound for λ
6	*DTOL*	u	drop tolerance for incomplete factorization
8	*SMIN*	u	lower limit for contour colors
9	*SMAX*	u	upper limit for contour colors
10	*RMAG*	u	window magnification factor
11	*CENX*	u	*(CENX,CENY)* are the window center coordinates
12	*CENY*	u	
15	*HMAX*	u	approximate largest element size
16	*GRADE*	u	largest growth factor for adjacent elements
17	*HMIN*	u	approximate smallest edge length
21	*RL*		current value of λ_h
22	*R*		current value of $\rho(u_h,\lambda_h)=\rho_h$
23	*RLDOT*		current value of $\dot{\lambda}_h$
24	*RDOT*		current value of $\dot{\rho}_h$
25	*SVAL*		current value of smallest singular value
26	*RLSTRT*		starting value for λ_h
27	*RSTRT*		starting value for $\rho(u_h,\lambda_h)$
31	*RL0*		previous value of λ_h
32	*R0*		previous value of $\rho(u_h,\lambda_h)=\rho_h$
33	*RL0DOT*		previous value of $\dot{\lambda}_h$
34	*R0DOT*		previous value of $\dot{\rho}_h$
35	*SVAL0*		previous value of smallest singular value
37	*ENORM1*		estimate for $\|u-u_h\|_{\mathcal{H}^1(\Omega)}$
38	*UNORM1*		the norm $\|u_h\|_{\mathcal{H}^1(\Omega)}$
39	*ENORM2*		estimate for $\|u-u_h\|_{\mathcal{L}^2(\Omega)}$
40	*UNORM2*		the norm $\|u_h\|_{\mathcal{L}^2(\Omega)}$

Table 2.6. *RP array definitions.*

For example, if

$$f=\lambda\frac{\partial u}{\partial x}+u^2,$$

then the following code fragment would be included in *Subroutine FXY*.

```
VALUES(K0)= RL * UX + U**2
VALUES(KX)= RL
VALUES(KU)= 2. * U
VALUES(KL)= UX
```

I	*RP(I)*	u	definition
51	*EPS*		the machine epsilon
52	*STEP*		damping step s for Newton's method
53	*BNORM*		norm of current Newton residual $\|\mathcal{G}\|$
54	*RELERR*		relative size of Newton update $\|\delta U\|/\|U\|$
55	*ANORM*		maximum diagonal entry in Jacobian matrix
56	*RELRES*		the relative residual $\|\mathcal{G}_k\|/\|\mathcal{G}_0\|$
57	*BRATIO*		the relative residual $\|\mathcal{G}_k\|/\|\mathcal{G}_{k-1}\|$
58	*BNORM0*		norm of initial Newton residual $\|\mathcal{G}_0\|$
59	*RELER0*		relative size of fisrt Newton update $\|\delta U\|/\|U\|$
60	*DNEW*		the discrete inner product $-\langle G_u \delta U, G\rangle$
63	*RMU*		current value of μ
64	*RMU0*		previous value of μ
67	*SCLEQN*		current value of scalar equation $N - \sigma$
68	*SCALE*		scaling factor for scalar equation
69	*THETAL*		$(2-\theta)\dot{\lambda}_h$ in scalar equation
70	*THETAR*		$\theta\dot{\rho}_h$ in scalar equation
71	*SIGMA*		the step σ for scalar equation
72	*DELTA*		Newton update for λ_h
73	*DRDRL*		the value of $\partial\rho/\partial\lambda$
74	*SEQDOT*		the value of $\dot{N}$
76	*QUAL*		target element quality
77	*ANGMN*		target minimum angle
78	*DIAM*		approximate diameter of Ω
79	*BEST*		value of *TRIGEN* quality function
80	*AREA*		area of Ω
81	*TOLA*		angle tolerance
82	*ARCMIN*		minimum arc
82	*ARCMAX*		maximum arc
84	*TOLZ*		contour tolerance
85	*TOLF*		function value tolerance
87	*XMIN*		
88	*XMAX*		$\Omega \subset (XMIN, XMAX) \times (YMIN, YMAX)$
89	*YMIN*		
90	*YMAX*		

TABLE 2.6, continued.
RP array definitions.

VALUES(KUU)= 2.
VALUES(KLX)= 1.

The subroutine corresponding to p_2 is *P2XY* and is called using

I	*SP(I)*	u	definition
1	*ITITLE*	u	title for *INPLT*
2	*FTITLE*	u	title for *TRIPLT*
3	*GTITLE*	u	title for *GPHPLT*
4	*MTITLE*	u	title for *MTXPLT*
5	*SHCMD*	u	string for shell command
6	*RWFILE*	u	save file for read/write commands
7	*JRFILE*	u	read file for journal command
8	*JWFILE*	u	write file for journal command
9	*BFILE*	u	output file
10	*JTFILE*	u	temporary file for journal command
11	*IOMSG*		error message string
12	*CMD*		current command string
12	*LOGO*	u	logo for X-Windows display
14	*BGCLR*	u	background color for X-Windows display
15	*BTNBG*	u	button background color for X-Windows display
18	*PSFILE*	u	root name for PostScript files
19	*XPFILE*	u	root name for xpm files
20	*BHFILE*	u	root name for bh files
21	*SGHOST*	u	host name for *SG* display

Table 2.7. *SP array definitions.*

Call P2XY(X, Y, DX, DY, U, UX, UY, RL, ITAG, JTAG, VALUES).

The arguments are a superset of those of the previous subroutines, and all arguments with the same name serve the same purpose. This routine is called only with points (X, Y) lying on some edge $e_J \in \Gamma$. The additional arguments (DX, DY) are the unit normal direction for the edge, and *JTAG=IBNDRY(6,J)* is the user specified label for the given edge. The mapping given in Table 2.9 is used here as well.

The subroutine corresponding to g_1 is *GNXY* and is called using

Call GNXY(X, Y, U, RL, ITAG, VALUES).

This routine is called only for points $(X, Y) \in \partial\Omega_1$, and as in the previous cases, all arguments except the array *VALUES* are input. In this case *ITAG=IBNDRY(6,I)* is the user supplied label for the edge, and *VALUES* is an array of size 6. Here the labeled common block

common /VAL1/ K0, KU, KL, KUU, KUL, KLU, KLL

IFLAG	general return codes
0	normal return
25	wrong input data structure
IFLAG	*PLTMG* and *TRIGEN* errors
1	zero pivot in sparse factorization
2	Newton method line search failed
6	illegal problem type
9	continuation procedure failed
10	multigraph iteration failed to converge
11	Newton (Newton/DD) iteration failed to converge
48	MPI was off for a command needing MPI
49	$NPROC > NTF$ in load balance
71	no interface unknowns in DD solver
72	*IPATH* array not created
97	unmatched interface edges in *IPATH*
IFLAG	storage errors
18	storage exhausted in matrix array *JA*
19	storage exhausted in matrix array *A*
20	storage exhausted in work array *W*
21	storage exhausted in array *ITNODE*
22	storage exhausted in arrays *VX* and *VY*
23	storage exhausted in arrays *XM* and *YM*
24	storage exhausted in array *IBNDRY*
IFLAG	data errors for triangulation
-31	illegal *ITNODE(K,*)* $K = 1, 2, 3$
-32	overlapping triangles in *ITNODE*
IFLAG	data errors for triangulation and skeleton
-40	illegal value for *NVF*, *NCF*, *NTF*, or *NBF*
-41	illegal *IBNDRY(K,*)* $K = 1, 2$
-42	illegal *IBNDRY(3,*)*
-43	illegal *IBNDRY(4,*)*
-44	incorrect circle center coordinates
-45	arc greater than $\pi/2$ in length
-46	error in linked edges
-47	boundary vertex without two boundary edges
-48	*ITNODE* and *IBNDRY* are not consistent
IFLAG	data errors for skeleton
-51	illegal *ITNODE(1,*)*
-52	illegal *ITNODE(2,*)*
-53	skeleton tracing error
-54	region specified in clockwise order
-55	illegal *ITNODE(3,*)*

Table 2.8. *Error flag values.*

pointer	index	function
K0	1	f
KU	2	f_u
KX	3	f_{u_x}
KY	4	f_{u_y}
KL	5	f_λ
KUU	6	f_{uu}
KXX	7	$f_{u_x u_x}$
KYY	8	$f_{u_y u_y}$
KUX=KXU	9	f_{uu_x}
KUY=KYU	10	f_{uu_y}
KXY=KYX	11	$f_{u_x u_y}$
KUL=KLU	12	$f_{u\lambda}$
KXL=KLX	13	$f_{u_x\lambda}$
KYL=KLY	14	$f_{u_y\lambda}$
KLL	15	$f_{\lambda\lambda}$

Table 2.9. *VALUES array for subroutine FXY.*

assists in mapping function and derivative values to particular entries in the *VALUES* array. The details of the mapping are given in Table 2.10.

pointer	index	function
K0	1	g
KU	2	g_u
KL	3	g_λ
KUU	4	g_{uu}
KUL=KLU	5	$g_{u\lambda}$
KLL	6	$g_{\lambda\lambda}$

Table 2.10. *VALUES array for subroutine GNXY.*

The subroutine corresponding to g_2 is *GDXY* and is called using

Call GDXY(X, Y, RL, ITAG, VALUES).

This routine also supplies the upper and lower bounds for the inequality constraints on u_h for the obstacle problem, bounds on λ_h in the case that $\lambda = \lambda(x, y)$, and the initial guess u_0 for the solution u_h. When called to supply a Dirichlet boundary condition, $(X, Y) \in \partial\Omega_2$ and *ITAG=IBNDRY(6,I)* is an edge label. When called in regard to inequality constraints and the initial guess, $(X, Y) \in \Omega$ and *ITAG=ITNODE(5,I)* is the element label supplied by the user. Similar to the other routines, *VALUES* is an output array of size 6. It's entries can be conveniently

accessed through pointers provided in the labeled common block

common /VAL2/ K0, KL, KLL, KLB, KUB, KIC

The details are provided in Table 2.11.

pointer	index	function
K0	1	g
KL	2	g_λ
KLL	3	$g_{\lambda\lambda}$
KLB	4	$\underline{u}$, $\underline{\lambda}$
KUB	5	$\overline{u}$, $\overline{\lambda}$
KIC	6	u_0

Table 2.11. *VALUES array for subroutine GDXY.*

Subroutine *QXY* is

Call QXY(X, Y, U, UX, UY, RL, ITAG, VALUES)

This routine provides the alternate function to display in *TRIPLT* and the alternate function for adaptive algorithms and skeleton generation in *TRIGEN*. The arguments are defined as in the other coefficient functions. The output array *VALUES* has dimension 5; It's entries can be conveniently accessed through pointers provided in the labeled common block

common /VAL3/ KF, KF1, KF2, KSK, KAD

whose entries are documents in Table 2.12.

pointer	index	function
K0	1	alternate scalar function for *TRIPLT*
KF1	2	first component of vector function for *TRIPLT*
KF2	3	second component of vector function for *TRIPLT*
KSK	4	alternate function for skeleton creation in *TRIGEN*
KAD	5	alternate function for adaptive algorithms in *TRIGEN*

Table 2.12. *VALUES array for subroutine QXY*

2.6 Sparse Matrix Storage.

Matrices generated in the solution process are stored in the sparse matrix format described in [3] using an integer array *JA* and a real array *A*. As an example,

consider the 4×4 matrix given by

$$A = \begin{pmatrix} a_{11} & a_{12} & & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ & a_{32} & a_{33} & \\ a_{41} & a_{42} & & a_{44} \end{pmatrix}.$$

This matrix is stored in *JA* and *A* as illustrated in Table 2.13. All nonzeros are stored in the array *A*. First the diagonal entries are stored, followed by the upper triangular entries, stored row by row. If the matrix is nonsymmetric, this is followed by the lower triangular entries, stored column by column. Symmetric and nonsymmetric storage is governed by the parameter *ISPD* as indicated in Table 2.14. The first $NVF+1$ entries of *JA* are pointers. In particular, entries *JA(I)* to *JA(I+1)* -1 of the *JA* array contain column indices for nonzeros in row I of the strict upper triangle. As illustrated in Table 2.13, the column indices stand in correspondence to the nonzeros of the upper triangle stored in the array *A*. If nonsymmetric storage is used, entries of the *transposed* lower triangle are stored in the same order as the upper triangle.

I	1	2	3	4	5	6	7	8	9	10	11	12	13
JA(I)	6	8	10	10	10	2	4	3	4				
A(I)	a_{11}	a_{22}	a_{33}	a_{44}	–	a_{12}	a_{14}	a_{23}	a_{24}	a_{21}	a_{41}	a_{32}	a_{42}

Table 2.13. *Sparse matrix data structures. JA has 9 entries. A has 9 entries if* $ISPD = 1$ *or 13 entries if* $ISPD = 0$.

ISPD	storage/iteration options
0	nonsymmetric/biconjugate gradient
1	symmetric/conjugate gradient

Table 2.14. *The values of ISPD.*

In the multigraph iterative method [21, 3], sparse matrices for each level, as well as *ILU* factorizations, permutation matrices, and inter level transfer matrices arise. These matrices are all stored in the user supplied arrays *JA* and *A*. *JA* is an integer array of length *LENJA*; *A* is a real array of length *LENA*; . An internal array, *KA*, allocated from the work array *W*, contains pointers into the *JA* and *A* arrays for all the matrices generated by the multigraph solver. See [3] for details.

Chapter 3
Mesh Generation

3.1 Overview.

Subroutine *TRIGEN* creates or adaptively modifies the data structures defining the region Ω. There are options to generate a triangulation from a skeleton, a skeleton from a triangulation, adaptively refine or unrefine a triangulation, uniformly refine a triangulation, and adaptively smooth the vertices of a triangulation. *TRIGEN* also has several options for partitioning and mesh management in parallel computation environments. The parameter *IADAPT* specifies various options for *TRIGEN*, summarized in Table 3.1.

TRIGEN is called using the statement

Call TRIGEN(VX, VY, XM, YM, ITNODE, IBNDRY, JA, A,
IP, RP, SP, IU, RU, SU, W, QXY)

Except for the case $IADAPT = 5$, on input the arrays *VX*, *VY*, *XM*, *YM*, *ITNODE*, and *IBNDRY* should define a triangulation. For $IADAPT = 5$, the input should be a skeleton. The arrays *JA*, *A*, and *W* provide workspace, while *IU*, *RU*, and *SU* are broadcast and received in MPI communication steps, but are not directly used in *TRIGEN*. When *TRIGEN* is used to adaptively modify an existing triangulation the procedures generally rely on local a posteriori error estimates for the finite element approximation, although some options are provided for adaptation based on other functions.

3.2 Creating a Triangulation from a Skeleton.

When $IADAPT = 5$, on input the arrays *VX*, *VY*, *XM*, *YM*, *ITNODE*, and *IBNDRY* should define a skeleton as described in Section 2.3. *TRIGEN* triangulates the subregions defining the skeleton in the order that they are given in *ITNODE*, taking into account shared internal boundaries and the symmetry requirements.

Let t be a triangle with area a and side lengths h_1, h_2, and h_3. The quality

IADAPT			option
	u_h	*QXY*	
	0		error estimates only
	1	-1	refine *or* unrefine mesh
	2	-2	unrefine *and* refine mesh
	3	-3	smooth mesh points
4			uniform refinement
5			skeleton → triangulation
	6	-6	triangulation → skeleton
	7		load balance (MPI)
8			reconcile mesh (MPI)
9			gather mesh (MPI)

Table 3.1. *Some options use a posteriori error estimates for the computed solution u_h or interpolation errors for the alternative function QXY. Other options require MPI for parallel communication.*

of t, $q(t)$, is measured using the formula

$$q(t) = 4\sqrt{3}a/(h_1^2 + h_2^2 + h_3^2). \tag{3.1}$$

The function $q(t)$ is normalized to equal one for an equilateral triangle and to approach zero for triangles with small angles. In attempting to compute a high quality triangulation, *TRIGEN* uses

$$q(t) \geq .6 \tag{3.2}$$

as a test for acceptability of a triangle (sufficiently small interior angles on the boundaries of the subregions Ω_i could cause (3.2) to be violated).

The triangulation process for those regions for which $ITNODE(3, I) \neq 0$ is simple and is carried out by generating the appropriate affine mapping. The triangulation process for subregions with $ITNODE(3, I) = 0$ is somewhat complicated but embodies three straightforward heuristics.

Given a subregion viewed as a polygon (possibly with curved edges, and interior angles of size π or greater), *TRIGEN* first tries to reduce the order of the polygon by one by "chopping" off a triangle using a vertex with small interior angle. Inequality (3.2) and several less obvious conditions must be satisfied for a successful chop. When the chopping strategy is no longer successful, *TRIGEN* checks to see if the remaining polygon is convex with six or fewer sides. If it is, *TRIGEN* tries to triangulate the entire remaining subregion by adding the centroid as a vertex and connecting it to each boundary vertex. All the resulting triangles must satisfy (3.2) and some other conditions for this strategy to be successful.

If the second strategy fails or is inapplicable, *TRIGEN* tries to break the polygon into two smaller polygons by connecting two nonadjacent vertices by a straight line. *TRIGEN* excludes many potential cuts as geometrically infeasible or

otherwise undesirable. From the remaining possibilities *TRIGEN* picks the cut that maximizes the minimum of the four interior angles the cut creates. *TRIGEN* then applies the three strategies to the two newly created polygons in recursive fashion. After the region has been successfully triangulated, *TRIGEN* tries to improve the triangulation by (locally) rearranging edges and adjusting vertex locations such that the criterion (3.2) is optimized.

The user can control the triangulation process to some extent through the parameters *HMAX* and *GRADE*. Element size is controlled by *HMAX*. Normally, one should choose $0 < HMAX \leq 1$. *TRIGEN* then attempts to create triangles with edges shorter than $HMAX \cdot \mathrm{diam}(\Omega)$. If $HMAX \leq 0$ or $HMAX > 1$, *TRIGEN* will reset $HMAX = 1$. Setting *HMAX* only places an upper bound on triangle sizes; the sizes of the triangles actually generated depend strongly on the geometry of the Ω_i and may not achieve the bound.

GRADE is (approximately) the largest ratio of sizes of elements sharing a common edge ($1/GRADE$ is the smallest ratio). *GRADE* should be set on the interval $1.5 \leq GRADE \leq 2.5$; values outside this interval are set to the appropriate end point. Generally speaking, smaller values of *GRADE* result in smoother transitions from regions of large elements to those of small elements, and a higher overall quality measured by (3.1). On the other hand, larger values of *GRADE* tend to produce meshes with fewer elements, more rapid transitions in element size, and lower overall quality. One may have to experiment to achieve the proper balance between these conflicting objectives.

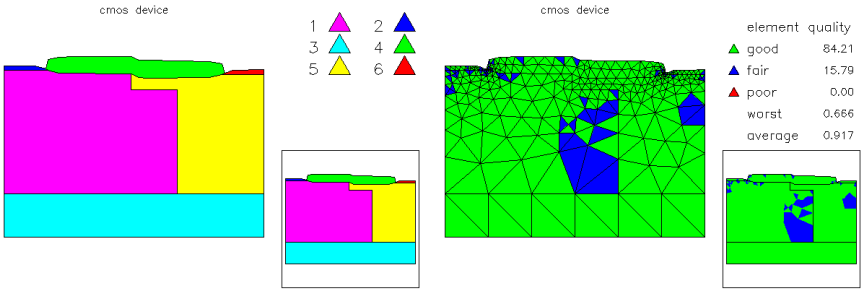
For example, consider the domain pictured in Figure 3.1, top left. The remaining pictures in Figure 3.1 show triangulations generated by *TRIGEN* for various values of *HMAX* and *GRADE*, illustrating their effect on the resulting triangulation.

The pictures are made by *INPLT* (see Section 5.3), which draws the mesh with elements colored according to the quality measure $q(t)$ in (3.1). In the pictures, an element is "good" if $q(t) \geq \sqrt{3}/2$, "fair" if $.6 \leq q(t) < \sqrt{3}/2$, and "poor" if $q(t) < .6$. This is an interesting region to triangulate because the two narrow subregions at the top require small elements. *TRIGEN* tries to use larger elements in the larger subregions, but is constrained by the choices of *HMAX* and *GRADE*. Decreasing *HMAX* or *GRADE* tends to improve the overall quality of the triangulation, at the expense of introducing more elements.

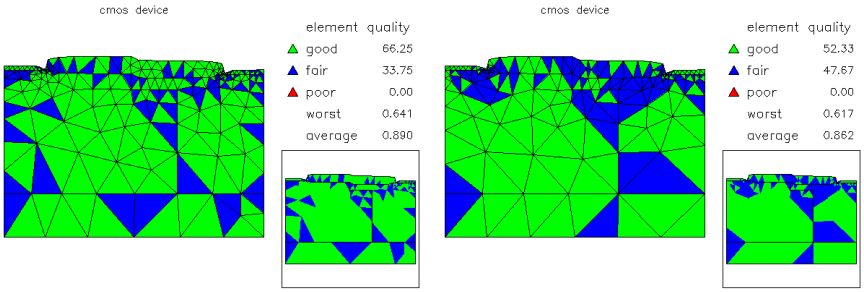
3.3 A Posteriori Error Estimates.

Of central importance to the adaptive procedures is the computation of a posteriori local error estimates [2, 1, 40, 42]. Our present a posteriori error estimate is based on a superconvergent approximation of ∇u [22, 23]. In particular, given the finite element function u_h, we compute the piecewise linear vector function $S^m Q_h \nabla u_h$, where Q_h is the $\mathcal{L}^2$ projection, and S is a smoothing operator based on the discrete Laplace operator; in *PLTMG*, we take $m = 2$. See [22, 23] for details. The recovered gradient is stored in *PLTMG* in the grid arrays *UX* and *UY*.

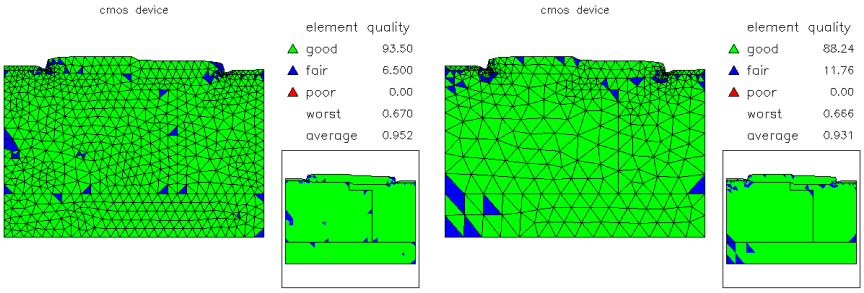
Using the recovered gradient, we compute a local error estimate ϵ_t for $t \in \mathcal{T}$.



A skeleton with NTF = 6, NVF = 30, NBF = 35, NCF = 0 (left). The triangulation for HMAX = 0, GRADE = 1.5 has NTF = 513, NVF = 294 (right).



The triangulation for HMAX = 0, GRADE = 2.0 has NTF = 323, NVF = 196 (left). The triangulation for HMAX = 0, GRADE = 2.5 has NTF = 258, NVF = 161 (right).



The triangulation for HMAX = .03, GRADE = 1.5 has NTF = 1323, NVF = 722 (left). The triangulation for HMAX = .06, GRADE = 1.5 has NTF = 672, NVF = 380 (right).

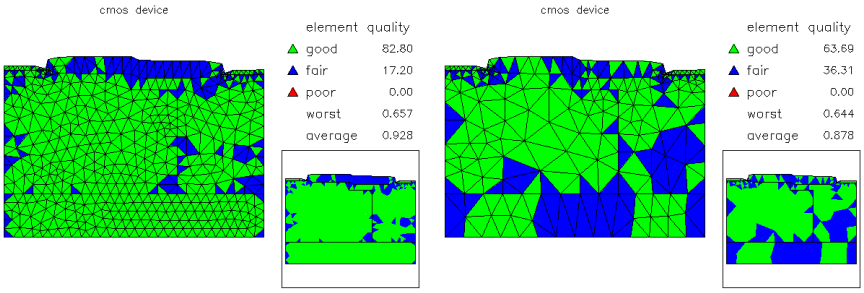


Figure 3.1. *The triangulation for HMAX = .03, GRADE = 2.5 has NTF = 913, NVF = 512 (left). The triangulation for HMAX = .06, GRADE = 2.5 has NTF = 369, NVF = 223 (right).*

Our estimate is motivated by noting that under certain circumstances, $\|\nabla(u_q - u_I)\|_{\mathcal{L}^2(\Omega)}$ is an asymptotically exact estimate of $\|\nabla(u - u_h)\|_{\mathcal{L}^2(\Omega)}$. Here u_q is the piecewise quadratic interpolant for u. Thus $u_q - u_I$ is a locally defined quadratic polynomial with value zero at all vertices of the mesh. On a given element t, $u_q - u_I$ can be expressed as a linear combination of quadratic "bump functions" q_k associated with the edge midpoints of t,

$$u_q - u_I = \sum_{k=1}^{3} \ell_k^2 \boldsymbol{t}_k^t M_t \boldsymbol{t}_k \, q_k(x, y) \tag{3.3}$$

where ℓ_k is the length of edge k, $\boldsymbol{t}_k$ is the unit tangent, and

$$M_t = -\frac{1}{2} \begin{pmatrix} \partial_{xx} u_q & \partial_{xy} u_q \\ \partial_{yx} u_q & \partial_{yy} u_q \end{pmatrix}.$$

is the Hessian matrix. All terms on the right hand side of (3.3) are known except for the second derivatives appearing in the Hessian matrix M_t. In our local error indicator, we simply approximate the second derivatives in the Hessian matrix M_t using derivatives of $S^m Q_h \nabla u_h$. In particular, let

$$\begin{aligned} \tilde{M}_t &= -\frac{1}{2} \begin{pmatrix} \partial_x S^m Q_h \partial_x u_h & \partial_x S^m Q_h \partial_y u_h \\ \partial_y S^m Q_h \partial_x u_h & \partial_y S^m Q_h \partial_y u_h \end{pmatrix}, \\ \bar{M}_t &= \frac{\alpha_t}{2} (\tilde{M}_t + \tilde{M}_t^t), \\ \epsilon_t &= \sum_{k=1}^{3} \ell_k^2 \boldsymbol{t}_k^t \bar{M}_t \boldsymbol{t}_k \, q_k(x, y). \end{aligned} \tag{3.4}$$

The normalization constant α_t is chosen such that the local error indicator η_t satisfies

$$\eta_t \equiv \|\nabla \epsilon_t\|_{\mathcal{L}^2(t)} = \|(I - S^m Q_h) \nabla u_h\|_{\mathcal{L}^2(t)}.$$

Normally we expect that $\alpha_t \approx 1$, which is likely to be the case in regions where the Hessian matrix for the true solution is well defined. Near singularities, u is not smooth and we anticipate difficulties in estimating the Hessian. For elements near such singularities, α_t provides a heuristic for partly compensating for poor approximation.

Global a posteriori estimates $\|\epsilon_t\|_{\mathcal{L}^2(\Omega)}$ and $\|\nabla \epsilon_t\|_{\mathcal{L}^2(\Omega)}$ are stored as the parameters *ENORM2* and *ENORM1*, respectively.

3.4 Adaptive Mesh Refinement and Unrefinement.

When $IADAPT = 1$, the current mesh is adaptively refined or unrefined. When $NVTRGT > NVF$, the mesh is refined, while if $NVTRGT < NVF$, the mesh is unrefined. In either case, the goal is to achieve the best possible mesh using (approximately) $NVTRGT$ vertices.

When $IADAPT = 2$, both refinement and unrefinement are employed. First, the mesh is unrefined to obtained a mesh with approximately $NVTRGT < NVF$

vertices. The mesh is then refined to obtain a mesh with approximately *NVF* vertices. The output triangulation thus has approximately the same number of vertices as the input triangulation, but the topology of the mesh and the distribution of mesh points can be quite different.

Our basic refinement algorithm uses the longest edge bisection procedure of Rivara [30, 37] and does not generate a refined element tree. All current elements are placed in a heap data structure according to the size of the error estimates. The element with largest error estimate is at the root of the heap. This element is selected for refinement and is bisected along its longest edge. The neighbor element sharing that edge is also bisected along its longest edge. If the result is a triangulation (i.e., the longest edge for both elements is the same), the process stops. Otherwise, it is recursively applied to the longest edge neighbors of all refined elements. An example is shown is Figure 3.2. This process is known to have finite termination, typically in a very small number of steps. When the longest edge bisection process finally results in a triangulation, the new elements are created and added to the triangulation data structures. New elements inherit the second derivative information from their parents, so error estimates can be computed and the heap updated. Using the updated heap, the refinement process continues, until a mesh with approximately *NVTRGT* vertices is created. Local edge swapping and mesh smoothing algorithms are applied to locally optimize the shape regularity of the final mesh in terms of the quality measure (3.1).

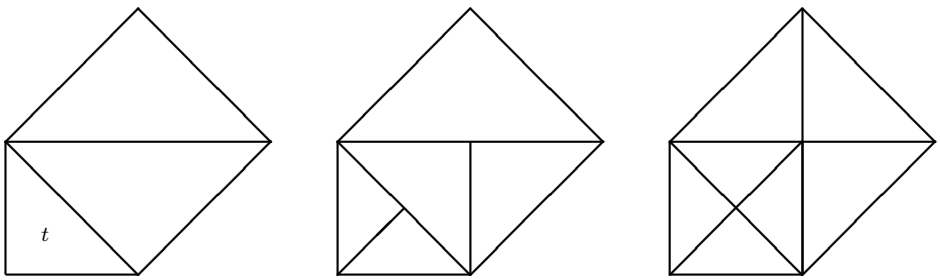


Figure 3.2. *Element t is refined by the longest edge bisection method. The original mesh is on the left. The first step of bisection (middle) does not yield a compatible triangulation. However, the second step (right) does yield a triangulation.*

In the case of unrefinement, the basic step consists of deleting vertices from the mesh, rather than directly unrefining elements. Each vertex v is associated with a region Ω_v, as illustrated in Figure 3.3. The error associated with vertex v is the largest error of any element contained in Ω_v. With these definitions, the unrefinement procedure is quite analogous to the refinement procedure described above. All the vertices are placed in a heap based on their errors, with the vertex of smallest error at the root. Certain vertices, which are critical to the geometric

integrity of the domain as a whole (e.g., corner vertices on the boundary of the region), are given artificially large errors. Vertices of low degree have their errors reduced a bit to favor their elimination.

In the elimination step, the root vertex of the heap is eliminated from the mesh. The region Ω_v associated with this mesh is then triangulated using the boundary vertices, as shown in Figure 3.3. The newly created elements inherit second derivative information from the original elements in Ω_v (through suitable averaging), and error estimates are computed for the new elements using (3.4) The vertices lying on $\partial\Omega_v$ have their errors updated as required, and the heap is updated. The process is continued until a mesh with *NVTRGT* vertices is achieved. As in the case of refinement, local edge swapping and mesh smoothing are used to improve the shape regularity of the final mesh in terms of the quality measure (3.1).

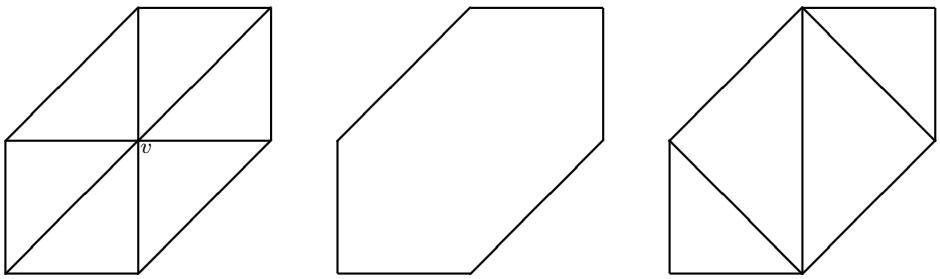


Figure 3.3. *On the left is the subregion Ω_v, associated with vertex v. To unrefine the mesh, vertex v and all its incident edges are removed from the triangulation (middle). The region Ω_v is then triangulated using the boundary vertices (right).*

If $IADAPT = -1$ or $IADAPT = -2$, the refinement and/or unrefinement processes are carried out using interpolation errors for the function *QXY* in place of the a posteriori error estimates. In particular, for a given element t, let q_2 denote the quadratic interpolating polynomial for *QXY*, characterized by nodes at vertices and edge midpoints of t, and let q_1 denote the linear interpolant characterized by the vertices of t. Then $\tilde{e}_h = q_2 - q_1$ is a quadratic polynomial that is zero at the vertices of t. Second derivatives for the interpolation error are computed in a fashion analogous to the case of error estimates and used in the adaptive processes.

We do not anticipate that this option will be used much; it was originally implemented to allow subroutine *TRIGEN* to be debugged independently of subroutine *PLTMG*. On the other hand, there may be special cases where some function other than $\|\nabla\epsilon_t\|_{\mathcal{L}^2(t)}$ should be optimized. Note that if *TRIGEN* is called before a solution u_h is computed by *PLTMG*, the arguments *U*, *UX*, *UY*, and *RL* in function *QXY* will be arbitrary and should be ignored.

Some examples are shown in Figure 3.4. In these examples, we employ the

alternate function $QXY = r^{1/4}\sin(\theta/4)$ defined on the circular domain with a crack shown in Figure 2.1. The initial mesh with $NVF = 10$ is shown in Figure 3.4, upper left. Three refined meshes were generated from this mesh using calls to *TRIGEN* with $IADAPT = -1$ and $NVTRGT = 40, 160, 640$.

3.5 Adaptive Mesh Smoothing.

When $IADAPT = 3$, subroutine *TRIGEN* does no refinement or unrefinement of the mesh but rather adjusts the (x, y) coordinates of the mesh points (VX and VY) in an attempt to optimize the mesh.

The procedure consists of a Gauss–Seidel-like iteration on the vertices in the mesh, where each vertex is locally optimized with all other vertices held fixed [19]. Four sweeps are performed. Typically a given vertex v is allowed to move within the region Ω_v shown in Figure 3.3. Not all vertices in the mesh are allowed to move. Some boundary and interface vertices must remain fixed to preserve the definition of the region. These vertices are called *corners*. Corners include actual geometric corners of the region, vertices where boundary conditions change type or label, vertices where interfaces intersect the boundary, and vertices where two or more interfaces intersect. An interface here is taken as any sequence of triangle edges that separate triangles with different user defined labels. Vertices on the boundary or on interfaces that are not designated corners are allowed to move only along the boundary or interface. The remaining vertices, called *interior* vertices, are allowed to move freely within Ω_v. As in our refinement algorithms, some local mesh smoothing based on (3.1) is used to locally optimize the shape regularity of the mesh.

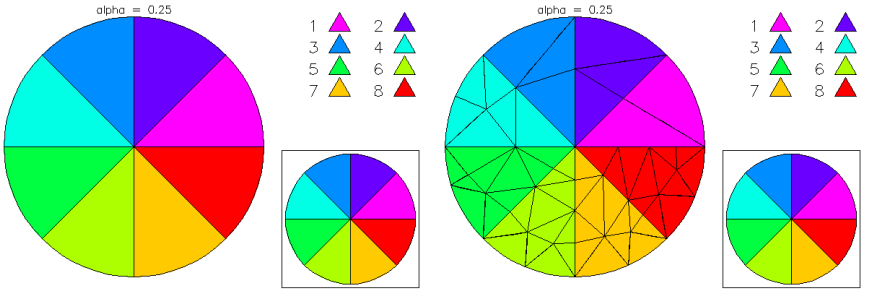
For each vertex $v = (x, y)$ in the mesh, we solve the minimization problem

$$\min_{x,y} \|\nabla \epsilon_t\|^2_{\mathcal{L}^2(\Omega_v)} \tag{3.5}$$

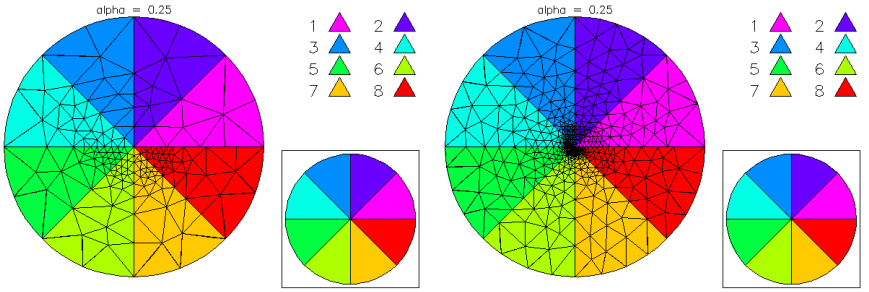
of order two by a damped Newton's method. As noted above, we assume the second derivatives are constant in each element t having v as a vertex, leading to an overall piecewise constant approximation of the second derivatives on Ω_v. All other dependencies on $v = (x, y)$ are taken into account by Newton's method. Boundary and interface vertices have an additional constraint equation, so an appropriately constrained version of problem (3.5) is solved for those vertices. Besides its usual task of ensuring sufficient decrease, the damping strategy for Newton's method is also used to ensure that the point (x, y) remains well within Ω_v, so that all triangles are always well defined. It is interesting to note that the function $\|\nabla \epsilon_t\|_{\mathcal{L}^2(\Omega_v)}$ contains a natural barrier function that becomes infinite as (x, y) approaches $\partial\Omega_v$.

In the case $IADAPT = -3$, the adaptive smoothing procedure uses the interpolation errors for the function QXY in place of the a posteriori error estimates, in a fashion analogous to the cases of refinement and unrefinement with $IADAPT < 0$.

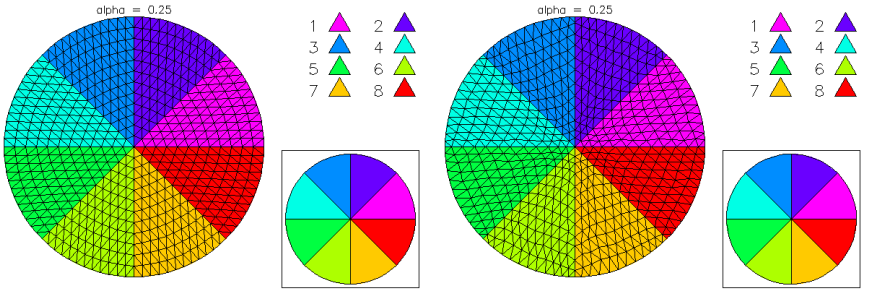
The mesh smoothing option is illustrated in Figure 3.4. We first uniformly refined the original mesh with $IADAPT = 4$ and $IREFN = 12$ (see Section 3.6). We then made two calls to *TRIGEN* with $IADAPT = -3$ to smooth the mesh points.



The initial triangulation with NTF = 8, NVF = 10, NBF = 10 and NCF = 1 (left). The refined triangulation with IADAPT = −1, NVTRGT = 40 has NTF = 60, NVF = 40 (right).



The refined triangulation with IADAPT = −1, NVTRGT = 160 has NTF = 287, NVF = 160 (left). The refined triangulation with IADAPT = −1, NVTRGT = 640 has NTF = 1217, NVF = 640 (right).



Beginning again from the original mesh with NTF = 8, the uniformly refined triangulation with IADAPT = 4 and IREFN = 12 has NTF = 1152, NVF = 637 (left). The same triangulation after one call to TRIGEN with IADAPT = −3 (right).

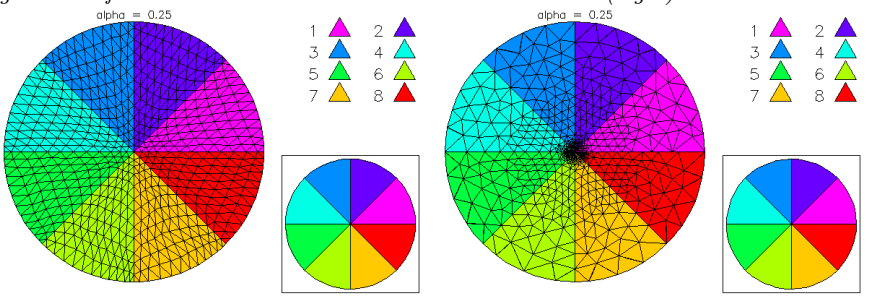


Figure 3.4. *The same triangulation after two calls to TRIGEN with IADAPT = −3 (left). The triangulation after calling TRIGEN with IADAPT = −2 and NVTRGT = 400 has NTF = 1209, NVF = 638 (right).*

3.6 Uniform Refinement.

When $IADAPT = 4$, subroutine *TRIGEN* will perform a uniform refinement of the existing triangulation. The refinement is controlled by the parameter $IREFN > 1$. Each element in the triangulation is uniformly divided into $IREFN^2$ similar triangles. Some examples are shown in Figure 3.5.

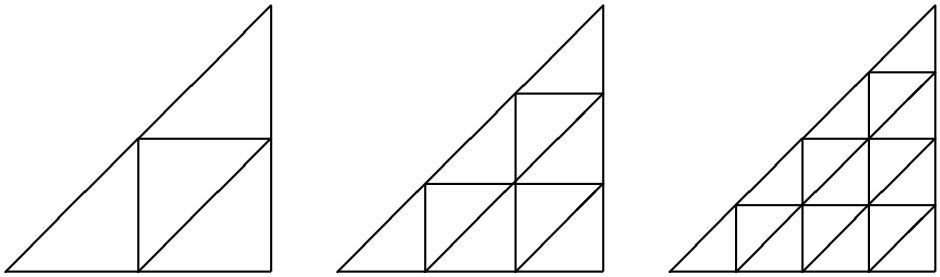


Figure 3.5. *Uniform refinement for the cases* $IREFN = 2, 3, 4$.

3.7 Creating a Skeleton from a Triangulation.

When $IADAPT = \pm 6$, subroutine *TRIGEN* generates skeleton data structures from a triangulation. This skeleton can then be used to generate a new triangulation (using *TRIGEN* with $IADAPT = 5$), providing what amounts to a static rezoning capability. This might be useful in situations where it is important or desirable to have grid lines in the mesh aligned with contour lines of a given function. Generating such a skeleton by hand might be cumbersome, or even impossible a priori if the function in question depends on the solution u. If $IADAPT = 6$, the solution is used to define the contour lines. If $IADAPT = -6$, the alternate function QXY is used. *TRIGEN* evaluates QXY at each vertex of each element in the mesh. QXY generally will be multivalued at the vertices because of discontinuities in ∇u_h. Therefore, *TRIGEN* computes a weighted average of QXY at each vertex, with weights proportional to the area of each element containing the vertex. The resulting grid function is then interpreted as a continuous piecewise linear polynomial.

NRGN equally spaced contour lines for the function specified by *IADAPT* are used as subregion boundaries. The value of *NRGN* has a significant impact on new triangulations later produced by *TRIGEN*. Larger values of *NRGN* generally result in the creation of more subregions. Since the length scales of the subregions are used in determining the length scales of the resulting triangles, triangulating a skeleton with thin subregions will result in many small triangles. Using fewer contours generally will result in larger length scales and potentially fewer triangles in the resulting mesh.

Contour spacing is also controlled to some extent through the parameter *HMIN*, which must satisfy $0 < HMIN \leq 1$. This parameter controls minimum contour spacing by (approximately) ensuring the contours are at least $HMIN \cdot \text{diam}(\Omega)$ apart. This requirement may effectively reduce the value of *NRGN* in conflicting situations.

At a conceptual level, the problem of creating a skeleton is similar to the problem of drawing a contour map in *TRIPLT*. However, in *TRIPLT*, except for the global problem of ordering the triangles for a surface plot, all the calculations proceed on an element-by-element basis, with the calculation for one element not interacting in any significant algorithmic way with the calculation for any other element. Here there are significant interactions on a global level, requiring a data structure that can contain the entire contour map.

Thus we develop a data structure in which Ω is partitioned into polygonal subregions. The boundary of a given subregion consists of portions of triangle edges and contour lines. The contours of a piecewise linear polynomial are straight lines in each element, with continuity between elements. Initially, each subregion is contained within a single triangle of the mesh and has 3–5 sides, depending on the orientation and number of specified contour lines that appear in the element.

These subregions could, by themselves, be developed into a skeleton. However, such a data set would have many more subregions and vertices than necessary. Thus *TRIGEN* performs transformations on the list of regions, aimed at reducing both the number of subregions and the number of vertices required to define them.

One basic step is to merge two subregions that share a common boundary into one larger subregion, thus eliminating all the internal edges and vertices along the common boundary. *TRIGEN* attempts to merge smaller subregions to form larger ones, generally respecting the following guidelines:

- Subregions with different labels cannot be merged. The labels are those originally provided by the user in *ITNODE*.
- If the common boundary is a contour edge, then the subregions cannot be merged.
- If the common boundary is not contiguous, then the subregions cannot be merged, as this would create a non–simply connected subregion.

The second guideline may be violated for exceptionally small subregions, which can occur frequently in the initial decomposition. If retained, they would cause many small triangles to be created by *TRIGEN*. If a subregion has an area A satisfying $A \leq HMIN^2 \mid \Omega \mid$, then *TRIGEN* will try to merge it with a larger subregion, even if it must violate the second guideline to do so. Generally, *TRIGEN* tries to create the largest subregions possible within its constraints.

A vertex is said to have *degree k* if it has k incident polygon edges. A *path* is a sequence of connected degree two vertices, generally terminated at each end by a vertex of degree greater than two. *TRIGEN* eliminates unnecessary vertices, adhering to the following guidelines:

- A vertex is a candidate for deletion only if it has degree two. This means that

the vertex is an internal vertex shared by only two subregions or a boundary vertex contained in only one subregion.

- A boundary vertex cannot be removed if the two boundary edges it separates have different boundary condition types or different labels. The labels are those originally provided by the user in *IBNDRY*.
- A vertex is removed only if it is (approximately) collinear with the vertices on the path containing the given vertex, or if it is a redundant vertex on a circular arc approximation of the path.

The data reduction transformations described above maintain a data set corresponding to a valid skeleton. Thus, after the transformations are completed, the remaining subregions are used to generate the appropriate skeleton data structures.

As an example, we consider the region shown in Figure 3.6, upper left. In this example, we first generated a triangulation using *TRIGEN* with $IADAPT = 5$, $HMAX = .1$, and $GRADE = 1.5$, shown in Figure 3.6 upper right. Using this triangulation as input, we generated three additional skeletons, with $NRGN = 5, 10, 20$. For all cases $HMIN = .05$, and $IADAPT = -6$, with the alternative function $QXY = x^2 + y^2$. For purposes of comparison, for each skeleton we computed a new triangulation based on that skeleton, using *TRIGEN* with $IADAPT = 5$, $HMAX = .1$, and $GRADE = 1.5$.

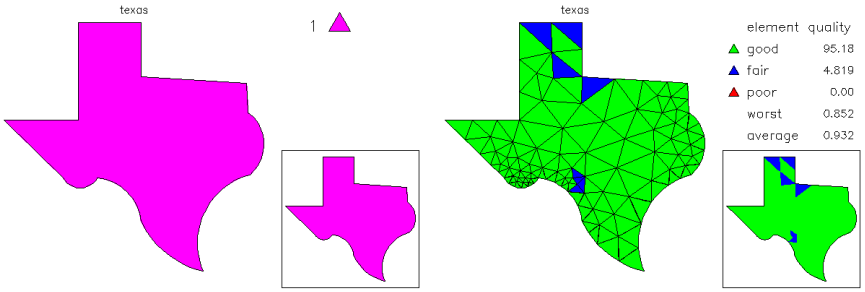
Note that increasing *NRGN* increases the complexity of the skeleton, tending to make more narrow regions, which in turn forces *TRIGEN* to create triangulations with more elements. On the other hand, using more regions forces the resulting triangulation to more closely follow the alternate function $QXY = x^2 + y^2$.

3.8 Parallel Adaptive Methods.

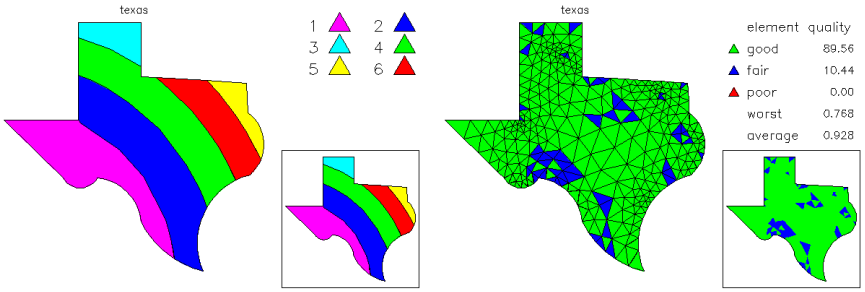
In this section we summarize the general strategy for adaptive mesh generations that is implemented in *PLTMG*. A number of static and dynamic load balancing approaches for unstructured meshes have been proposed in the literature [41, 38, 27, 28, 24, 29, 25]; most of the dynamic strategies involve repeated application of a particular static strategy. One of the difficulties in all of these approaches is the amount of communication that must be performed both to assess the current load imbalance severity, and to redistribute the work among the processors once the imbalance is detected and an improved distribution is calculated.

The approach used by *PLTMG* is based upon the Bank-Holst algorithm [10, 11, 31, 32], that addresses the load balancing problem in a new way, requiring far less communication. Another important point is that our approach allows *PLTMG* to run in a parallel environment without a large investment in additional coding. This approach has three main components:

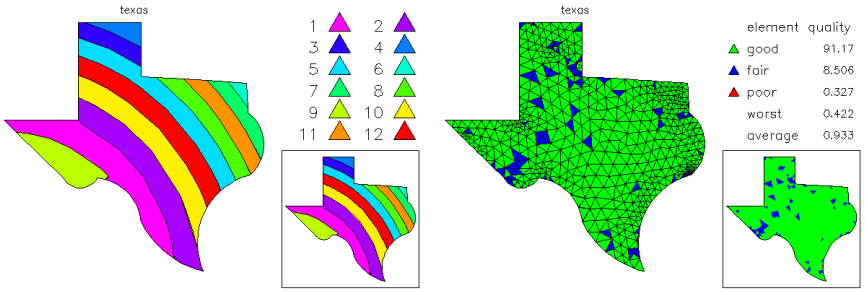
Step 1: A small (nonlinear) problem is solved on an initial coarse mesh, and *a posteriori* error estimates are computed for the coarse grid solution. The triangulation is partitioned such that each subdomain has approximately equal *error* (although they can significantly differ in size and numbers of elements).



The original skeleton with NTF = 1, NVF = 30, NBF = 30, NCF = 5 (left). The triangulation has NTF = 166, NVF = 108 (right).



A skeleton created with NRGN = 5, based on the original triangulation, has NTF = 6, NVF = 61, NBF = 66, NCF = 7 (left). The new triangulation has NTF = 517, NVF = 303 (right).



A skeleton created with NRGN = 10, based on the original triangulation, has NTF = 12, NVF = 95, NBF = 106, NCF = 9 (left). The new triangulation has NTF = 917, NVF = 522 (right).

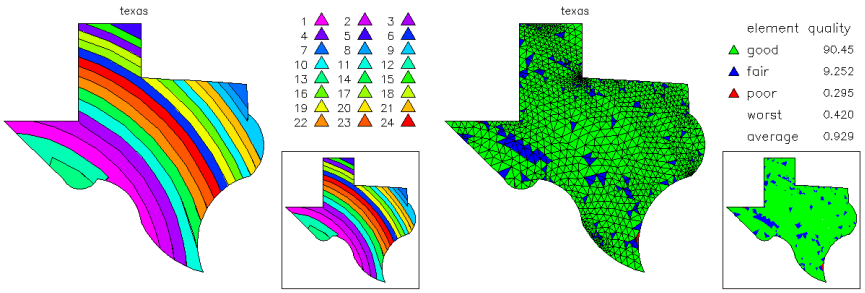


Figure 3.6. *A skeleton created with NRGN = 20, based on the original triangulation, has NTF = 24, NVF = 171, NBF = 194, NCF = 13 (left). The new triangulation has NTF = 2032, NVF = 1109 (right).*

Step 2: Each processor is provided the complete coarse mesh and solution, and instructed to solve the *entire* (nonlinear) problem, with the stipulation that its adaptive refinement should be limited largely to its own partition. Load balancing is achieved by instructing each processor to create a refined mesh with the same number of nodes.

Step 3: A final mesh is computed using the union of the refined partitions provided by each processor. This mesh is reconciled such that the (virtual) mesh made up of the refined subregions would be conforming. A final solution is computed, using a domain decomposition method. An initial guess is provided by the local solutions.

The above approach has several interesting features. First, the load balancing problem (Step 1) is reduced to the numerical solution of a small problem on a single processor, without requiring any modifications to *PLTMG*. Second, the adaptive mesh generation calculation (Step 2) takes place independently on each processor, and can also be performed with no communication.

The only parts of the calculation requiring communication are

1. the initial fan-out of the mesh distribution to the processors, once the decomposition is determined by the error estimator.
2. the mesh regularization, requiring communication to produce a global conforming mesh.
3. the final solution phase. Note that a good initial guess for Step 3 is provided in Step 2 by taking the solution from each subregion restricted to its partition.

The options $7 \le IADAPT \le 9$ provide basic parallel mesh management tools that support this paradigm. The domain decomposition solver is implemented as an option in subroutine *PLTMG*.

3.9 Mesh Partitioning.

When $IADAPT = 7$, *TRIGEN* computes *a posteriori* error estimates and partitions the mesh as in the Bank-Holst paradigm. If *PLTMG* is running on *NPROC* processors, then the mesh is partitioned into *NPROC* subregions, such that each subregion has approximately equal error. This algorithm is a variant of the recursive spectral bisection algorithm [26, 36, 39]. While this particular mesh partitioning algorithm is one of the more expensive of the choices that we could make, it is typically used only once on a relatively small problem. Although this calculation is important in the parallel processing environment, it is done on a single processor and does not use the MPI library. At the conclusion of the load balancing step, *TRIGEN* creates new internal edges in *IBNDRY* at the interface between different subregions.

We define the $NTF \times NTF$ symmetric, positive semi-definite M-matrix A by

$$A_{ij} = \begin{cases} -1 & i \neq j \text{ and } t_i \text{ and } t_j \text{ share a common edge} \\ 0 & i \neq j \text{ and } t_i \text{ and } t_j \text{ do not share a common edge} \\ s_i & i = j,\ s_i = -\sum_{k \neq i} A_{ik} \end{cases}$$

A typical row of A will have three nonzero off-diagonal elements and $A_{ii} = 3$; this is the so-called *discrete Laplacian* for the dual graph for the triangulation (the triangles themselves are the nodes of the dual graph, and the edges are defined by the adjacency relation). We consider the eigenvalue problem

$$A\psi = \lambda\psi \tag{3.6}$$

Our approach is standard; by construction, the smallest eigenvalue for (3.6) is $\lambda_1 = 0$ and $\psi_1 = (1, 1, \ldots, 1)^t$. Our interest is in the second eigenvector ψ_2, known as the Fiedler vector.

We use a standard binary tree with $2\,NPROC - 1$ nodes ($NPROC$ leaves and $NPROC - 1$ internal nodes). The root is labeled $i = 1$ and node i has children $2i$ and $2i + 1$, $1 \le i \le NPROC - 1$. Associated with each node is a weight ω_i denoting the number of leaves contained in its subtree. In particular, $\omega_i = 1$, $i = 2\,NPROC - 1, \ldots, NPROC$ and $\omega_i = \omega_{2i} + \omega_{2i+1}$ for $i = NPROC - 1, \ldots, 1$.

The entire mesh is assigned to root, and it is partitioned among its two children as follows. We first approximately solve the eigenvalue problem (3.6) for the whole mesh, and then create a permutation of the elements $\{q_i\}$ such that

$$\psi_{2,i} \le \psi_{2,j} \quad \text{if and only if} \quad q_i < q_j.$$

We then find the index k which provides the best partition of the form

$$\frac{1}{\omega_2} \sum_{q_i \le k} \|\nabla \epsilon_t\|^2_{\mathcal{L}^2(t_{q_i})} \approx \frac{1}{\omega_3} \sum_{q_i > k} \|\nabla \epsilon_t\|^2_{\mathcal{L}^2(t_{q_i})}.$$

The corresponding submeshes are assigned to the children nodes.

We apply this procedure recursively, at each level dividing each group of elements into two smaller groups by solving an eigenvalue problem of the type (3.6) restricted to that group of elements. The final result is $NPROC$ subregions with approximately equal error.

We now briefly describe some details of our procedure for computing the second eigenvector of (3.6). Our procedure is essentially just a classical Rayleigh quotient iteration [35], modified both to bias convergence to λ_2, and to account for the fact that the linear systems arising in the inverse iteration substep are solved (approximately) by an iterative process. To simplify notation and avoid multiple subscripts, we let $\phi_k \approx \psi_2$, where k now denotes the iteration index.

We suppose that we have a current iterate ϕ_k which satisfies $\phi_k^t \phi_k = 1$ and $\psi_1^t \phi_k = 0$. Using ϕ_k we compute the approximate eigenvalue $\tilde{\lambda}_{2,k} \approx \lambda_2$ from the Rayleigh quotient $\tilde{\lambda}_{2,k} = \phi_k^t A \phi_k$, and approximately solve the linear system

$$A\tilde{\delta}_k = r_k \equiv \tilde{\lambda}_{2,k}\phi_k - A\phi_k.$$

Note that by construction $\psi_1^t r_k = \phi_k^t r_k = 0$. This linear system is solved using the multigraph procedure [21].

From $\tilde{\delta}_k$, we form the vector δ_k satisfying $\delta_k^t \delta_k = 1$ and $\psi_1^t \delta_k = \phi_k^t \delta_k = 0$. Finally, we solve the 2×2 eigenvalue problem for $\hat{A}$, where

$$\hat{A} = \begin{pmatrix} \phi_k^t \\ \delta_k^t \end{pmatrix} A \begin{pmatrix} \phi_k & \delta_k \end{pmatrix}.$$

If $v = (\alpha, \beta)^t$ is an eigenvector corresponding to the smaller eigenvalue, we form $\tilde{\phi}_{k+1} = \alpha\phi_k + \beta\delta_k$. Then ϕ_{k+1} is formed from $\tilde{\phi}_{k+1}$ by normalization and orthogonalization to ψ_1.

3.10 Parallel Communication.

TRIGEN has three options ($7 \leq IADAPT \leq 9$) that require MPI library routines for communication. When $IADAPT = 7$, following the load balance computation, the processor corresponding to $IRGN = 1$ broadcasts its mesh, solution, and supporting data to all processors.

The option $IADAPT = 8$ reconciles the mesh. This is the most complex of the MPI options in *TRIGEN*, and is typically called once, at the conclusion of the second step of the Bank-Holst paradigm. It **must** be called before the domain decomposition solution in subroutine *PLTMG*, as *PLTMG* makes use of the parallel interface data structure *IPATH* generated by this call.

In creating the *IPATH* data structure, each processor first organizes its triangulation and solution data structures. Generally, edges and vertices on the interface between region *IRGN* and the rest of the domain appear first in their respective arrays (*IBNDRY*, *VX*, *VY*, *U*, etc). This data is organized to correspond to counter clockwise traversal of the interface. Next in all arrays comes data corresponding to the interior of subregion *IRGN*; generally, this should be the majority of the data. Finally, at the end of each array appears data corresponding to regions other than *IRGN*. Each processor then assembles its contributions to the preliminary *IPATH* array based on the reordered data, and this information is then exchanged among processors using MPI. *IPATH* in an integer array of length $4 \times LIPATH$ containing information about the interface edges.

After the boundary exchange, each processor tries to match its boundary interface edges to those provided by neighboring regions, in order to establish the structure of the global mesh. Typically this mesh is not conforming. When nonmatching edges are found, the region that is less refined does additional refinement until its boundary edges form a one-to-one match with those of its neighbors. An example is shown in Figure 3.7.

Each processor then reorders its data structures and communicates its contribution to the *IPATH* array a second time. This time the edge matching process concludes with no nonconforming edges found. By matching boundary edges at the interface, one also effectively matches vertices on the interface; this information is needed for the domain decomposition solver.

The option $IADAPT = 9$ is in some sense the inverse of $IADAPT = 7$. In this case, the global conforming mesh is assembled on the single processor corresponding to $IRGN = 1$. Each processor organizes its data as in the case $IADAPT = 8$, but now the data outside of *IRGN* is discarded. The remaining data is then gathered by the processor corresponding to $IRGN = 1$ and global conforming data structures are generated on this processor. The option $IADAPT = 9$ is provided as a convenience feature, and is not needed in the Bank-Holst paradigm. We remark that the *ITNODE*, *IBNDRY*, *VX*, *VY*, and other arrays need to be sufficiently large to

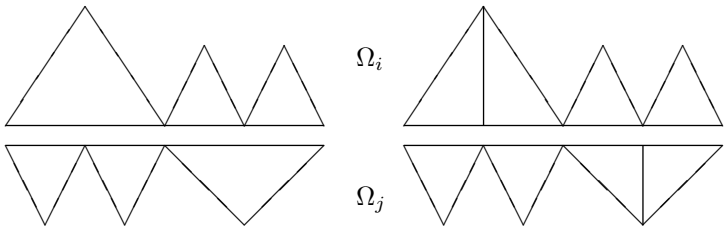


Figure 3.7. *The coarse side of a non matching interface (left) is refined to make the global mesh conforming (right).*

accommodate the entire global conforming mesh, which may not be possible for large parallel computations.

Chapter 4
Equation Solution

4.1 Overview.

Subroutine *PLTMG* solves the problems described in Section 1.1. The solution process for each class of problems has certain unique aspects, but all make use of Newton's method. Solvers for the resulting systems of linear equations are all based on the multigraph iterative method.

Subroutine *PLTMG* is entered using the statement

Call PLTMG(VX, VY, XM, YM, ITNODE, IBNDRY, JA, A, IP, RP, SP, W, A1XY, A2XY, FXY, GNXY, GDXY, P1XY, P2XY)

On input, the arrays *VX*, *VY*, *XM*, *YM*, *ITNODE*, and *IBNDRY* define a triangulation. Fortran subroutines *A1XY*, *A2XY*, *FXY*, *GNXY*, *GDXY*, *P1XY*, and *P2XY* are documented in Section 2.5. Parameters *IP*, *RP*, and *SP* arrays read and written by *PLTMG* are summarized in Tables 2.5–2.7. The arrays *JA*, *A*, and *W* provide workspace.

The parameter *IPROB* indicates the problem class; the various options are shown in Table 4.1. The case $IPROB > 0$ indicates a standard sequential solve, either on a single processor, or on multiple processors as part of the second phase of the Bank-Holst paradigm. The case $IPROB < 0$ indicates the global parallel domain decomposition solve as part of the Bank-Holst paradigm. Because this is a global solve it involves some MPI communication at each iteration step. When $IPROB < 0$, the parallel domain decomposition solve is preceded by a local solve on each processor, in order to insure the quality of the initial guess for the global problem.

The cases $IPROB = \pm 3$ and $IPROB = \pm 4$ have various suboptions unique to their particular problem class. The available options are specified through the parameter *ITASK*. These are summarized in Table 4.2.

IPROB	problem option
1	elliptic boundary value problem
2	obstacle problem
3	continuation problem
4	parameter identification problem
5	optimal control problem
-1	DD solve for elliptic boundary value problem
-2	DD solve for obstacle problem
-3	DD solve for continuation problem
-4	DD solve for parameter identification problem
-5	DD solve for optimal control problem

Table 4.1. *The parameter IPROB.*

ITASK	*IPROB*	option
0	1	default
0	2	default
0	3	continue to the nearest target point
1		continue to the nearest target or singular point
2		switch branches at a bifurcation point
3		switch λ and/or ρ; initialize with λ fixed
4		switch λ and/or ρ; initialize with ρ fixed
5		solve with $\sigma = 0$, $\theta = 0$ (λ fixed)
6		solve with $\sigma = 0$, $\theta = 2$ (ρ fixed)
7		solve with $\sigma = 0$, $\theta = 1$
0	4	default
8		switch λ and initialize
0	5	default

Table 4.2. *The parameter ITASK.*

4.2 Elliptic Boundary Value Problems.

When *IPROB* = 1, *PLTMG* solves the discrete system (1.6). If the underlying boundary values problem is not self-adjoint some upwinding terms based on the Scharfetter–Gummel discretization scheme [4, 8] are added to the discretization; in this case (1.6) should be replaced by: find $u_h \in \mathcal{M}_d$ such that

$$a_h(u_h, v) = 0 \qquad \text{for all } v \in \mathcal{M}_e, \tag{4.1}$$

where $a_h(u_h, v)$ reflects the additional stabilization terms. In any event, (4.2) corresponds to the system of nonlinear equations

$$\mathcal{G}(\mathcal{U}) = 0, \tag{4.2}$$

where the unknown vector $\mathcal{U}$ corresponds to the values of the finite element solution u_h at the vertices of the triangulation. The Jacobian matrix

$$\mathcal{A}(\mathcal{U}) = \frac{\partial \mathcal{G}(\mathcal{U})}{\partial \mathcal{U}}$$

is a sparse stiffness matrix corresponding to a *linear* elliptic boundary value problem (linearized about $\mathcal{U}$). Even in the event the the original problem is linear, *PLTMG* solves all problems with $IPROB = 1$ as nonlinear, and formally applies Newton's method to (4.2). In Figure 4.1, we summarize our approximate Newton procedure with line search.

Procedure Newton

N1 Begin with initial guess $\mathcal{U}_0$, and a sufficient decrease parameter τ. Set $k \leftarrow 0$, $s_0 \leftarrow 1$, and compute $\mathcal{G}_0$ and $\|\mathcal{G}_0\|$.

N2 solve (approximately) $\mathcal{A}_k \delta\mathcal{U}_k = -\mathcal{G}(\mathcal{U}_k)$.

N3 compute $\mathcal{U}_{k+1} = \mathcal{U}_k + s_k \delta\mathcal{U}_k$, $\mathcal{G}_{k+1}$, $\|\mathcal{G}_{k+1}\|$, and $\xi_{k+1} = \|\mathcal{G}_{k+1}\|/\|\mathcal{G}_k\|$.

N4 if $1 - \xi_{k+1} < \tau s_k$, then decrease s_k and go to **N3**; else set $s_{k+1} \leftarrow s_k/(s_k + (1 - s_k)\xi_{k+1}/100)$ and $k \leftarrow k + 1$.

N5 if converged, then exit; else go to **N2**.

Figure 4.1.

The scalar s_k is the damping parameter. When the sufficient decrease criterion is not satisfied on line N4 and s_k must be reduced, the next value is found through application of one step of a guarded secant/bisection algorithm to the one-dimensional minimization problem

$$\min_{s_k} \|\mathcal{G}(\mathcal{U}_k + s_k \delta\mathcal{U}_k)\|.$$

If sufficient decrease is achieved, the current s_k is used to predict s_{k+1}; this formula is designed to force rapid increase of $s_{k+1} \to 1$ when ξ_{k+1} becomes small as superlinear convergence occurs, and at the same time provide a reasonable first guess in the early stages of the Newton iteration, when damping is most important. A maximum of *MXNWTT* damped Newton iterations are allowed. *PLTMG* reports the actual number of Newton iterations used on the most recent call in the parameter *ITNUM*, and the number of evaluations of $\mathcal{G}$ as *IEVALS*; $IEVALS \geq ITNUM$, since more than one function evaluation may be used in each line search.

All sets of linear equations involving the matrices $\mathcal{A}(\mathcal{U})$ and $\mathcal{A}(\mathcal{U})^t$ have the appearance of finite element discretizations of linear elliptic boundary value problems. These systems are solved using the multigraph iterative method [21, 20]. The multigraph iteration is an algebraic multigrid method, governed by several input parameters. The parameter *ISPD* described in Table 2.14 specifies whether

symmetric or nonsymmetric storage is used in the A array. At each level an *ILU* factorization is used as a smoother. The parameter *DTOL* is the drop tolerance for this approximate factorization. Generally, smaller values of *DTOL* result in more accurate *ILU* factorizations, but higher costs in space and time per iteration. The extreme case $DTOL = 0$ results in a sparse direct factorization (if sufficient storage is available).

The multigraph iteration is used as a preconditioner for a composite step conjugate gradient or biconjugate gradient iteration, specified through *ISPD* as indicated in Table 2.14. The composite step algorithms [7, 6] are similar to the standard biconjugate gradient and conjugate gradient methods, respectively, except that they occasionally proceed from the iterate for step k to the iterate for step $k+2$. Such composite steps are taken to improve the stability of the recurrence relations and smooth the behavior of the residual norm. The maximum number of iterations to be used per solution is specified by the parameter *MXCG*. Note that as many as *MXCG* iterations are used each time a system of linear equations is solved.

As a simple example, we solve the Poisson equation

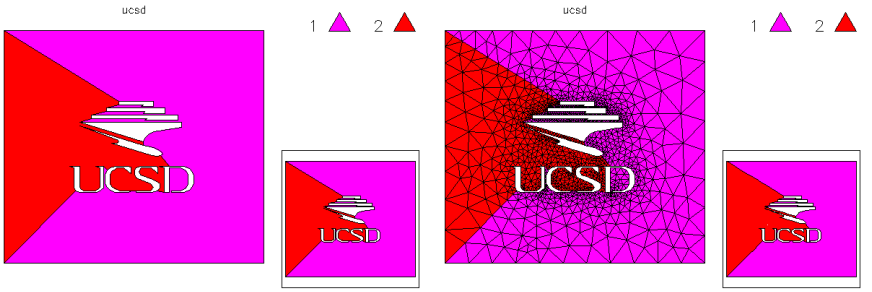
$$\begin{aligned} -\Delta u &= 1 \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned}$$

The domain Ω was provided as a skeleton and is shown in Figure 4.2. This problem was solved using eight processors. The skeleton was triangulated, and then a mesh with $NVF = 2000$ was adaptively created on one processor. The processor then did a load balance step ($IADAPT = 7$ in *TRIGEN*) and broadcast this mesh to all processors. The load balance and initial solution are shown in Figure 4.2. Each processor then independently continued the refinement process on its subregion, creating a local mesh with approximately 8000 vertices. The global refined mesh was made conforming ($IADAPT = 8$ in *TRIGEN*) and the domain decomposition solver invoked in *PLTMG* ($IPROB = -1$). The resulting global refined mesh had $NVG = 32908$ vertices. The global mesh, solution, and a posteriori error estimate are shown in Figure 4.2. The mesh is colored by element size, and element edges are not drawn.

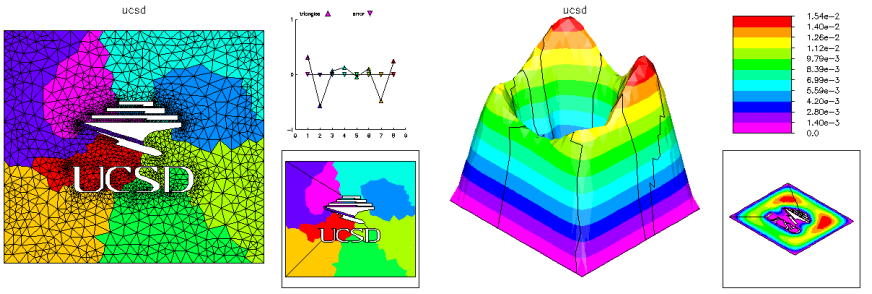
4.3 Domain Decomposition Solver

Here we describe the domain decomposition algorithm implemented in *PLTMG* for Step 3 of the Bank-Holst paradigm (see Section 3.8). This algorithm is described in detail in [14]. It is motivated by and similar to the domain decomposition algorithms described in [13, 12]. In the case $IPROB = -1$, this solver is used in place of the simple multigraph solver in line N2 of Procedure Newton given in Figure 4.1.

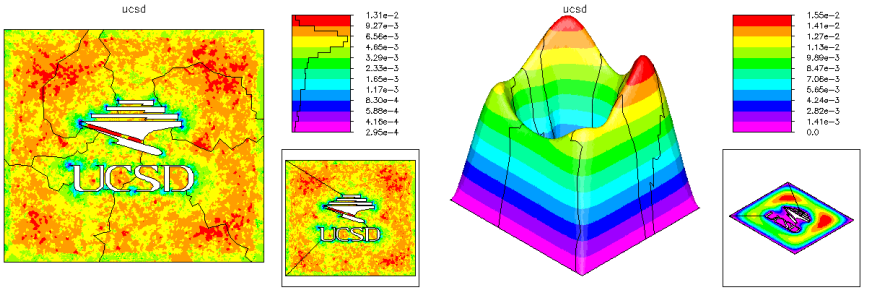
For simplicity in our discussion here, we restrict attention to the case of just two subdomains. In our scheme, each subregion contributes equations corresponding all fine mesh points, including its interface. Thus in general there will be multiple unknowns and equations in the global system corresponding to the interface grid points. This is handled by equality constraints that impose continuity at all mesh points on the interface. The result is a mortar-element like formulation, using Dirac



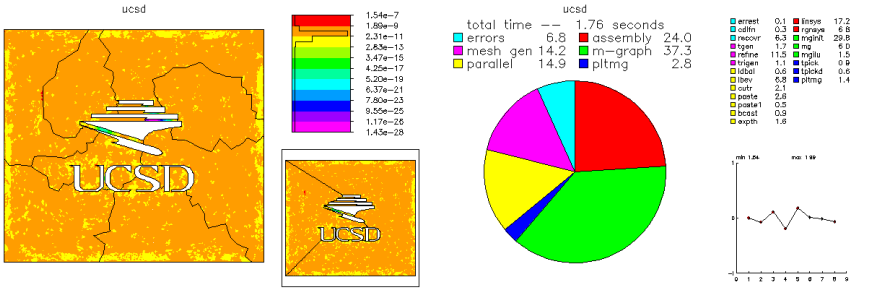
The skeleton and initial mesh with $NVF = 1382$.



The mesh with $NVF = 2000$ *showing the load balance and corresponding solution.*



The global refined mesh with $NVG = 32908$ *and corresponding solution.*



δ functions for the mortar element space. In any event, with a proper ordering of unknowns, the global system of equations has the block 5×5 form

$$\begin{pmatrix} A_{11} & A_{1\gamma} & & & \\ A_{\gamma 1} & A_{\gamma\gamma} & & & I \\ & & A_{\nu\nu} & A_{\nu 2} & -I \\ & & A_{2\nu} & A_{22} & \\ & I & -I & & \end{pmatrix} \begin{pmatrix} \delta U_1 \\ \delta U_\gamma \\ \delta U_\nu \\ \delta U_2 \\ \Lambda \end{pmatrix} = \begin{pmatrix} R_1 \\ R_\gamma \\ R_\nu \\ R_2 \\ U_\nu - U_\gamma \end{pmatrix}. \tag{4.3}$$

Here A_{11} and A_{22} correspond to the fine grid points on processors 1 and 2, respectively, that are not on the interface, while $A_{\gamma\gamma}$ and $A_{\nu\nu}$ correspond to interface points. The fifth block equation imposes continuity, and its corresponding Lagrange multiplier is Λ. The identity matrix appears because the global fine mesh is conforming. The introduction of the Lagrange multiplier and the saddle point formulation (4.3) are only for expository purposes; indeed, Λ is never computed or updated.

On processor 1, we develop a similar but "local" saddle point formulation. That is, the fine mesh subregion on processor 1 is "mortared" to the remaining course mesh on processor 1. This leads to a linear system of the form

$$\begin{pmatrix} A_{11} & A_{1\gamma} & & & \\ A_{\gamma 1} & A_{\gamma\gamma} & & & I \\ & & \bar{A}_{\nu\nu} & \bar{A}_{\nu 2} & -I \\ & & \bar{A}_{2\nu} & \bar{A}_{22} & \\ & I & -I & & \end{pmatrix} \begin{pmatrix} \delta U_1 \\ \delta U_\gamma \\ \delta \bar{U}_\nu \\ \delta \bar{U}_2 \\ \Lambda \end{pmatrix} = \begin{pmatrix} R_1 \\ R_\gamma \\ R_\nu \\ 0 \\ U_\nu - U_\gamma \end{pmatrix}, \tag{4.4}$$

where quantities with a bar (e.g., $\bar{A}_{22}$) refer to the coarse mesh. A system similar to (4.4) can be derived for processor 2. With respect to the right hand side of (4.4), the interior residual R_1 and the interface residual R_γ are locally computed on processor 1. We obtain the boundary residual R_ν, and boundary solution U_ν from processor 2; processor 2 in turn must be sent R_γ and U_γ. The residual for the coarse grid interior points is set to zero. This avoids the need to obtain R_2 via communication, and to implement a procedure to restrict R_2 to the coarse mesh on processor 1. Given our initial guess, we expect $R_1 \approx 0$ and $R_2 \approx 0$ at all iteration steps. R_γ and R_ν are not generally small, but $R_\gamma + R_\nu \to 0$ at convergence.

As with the global formulation (4.3), equation (4.4) is introduced mainly for exposition. The goal of the calculation on processor 1 is to compute the updates δU_1 and δU_γ, which contribute to the global conforming solution. To this end, we formally reorder (4.4) as

$$\begin{pmatrix} & -I & & I & \\ -I & \bar{A}_{\nu\nu} & & & \bar{A}_{\nu 2} \\ & & A_{11} & A_{1\gamma} & \\ I & & A_{\gamma 1} & A_{\gamma\gamma} & \\ & \bar{A}_{2\nu} & & & \bar{A}_{22} \end{pmatrix} \begin{pmatrix} \Lambda \\ \delta \bar{U}_\nu \\ \delta U_1 \\ \delta U_\gamma \\ \delta \bar{U}_2 \end{pmatrix} = \begin{pmatrix} U_\nu - U_\gamma \\ R_\nu \\ R_1 \\ R_\gamma \\ 0 \end{pmatrix}. \tag{4.5}$$

Block elimination of the Lagrange multiplier Λ and $\delta \bar{U}_\nu$ in (4.5) leads to the block

3×3 Schur complement system

$$\begin{pmatrix} A_{11} & A_{1\gamma} & \\ A_{\gamma 1} & A_{\gamma\gamma} + \bar{A}_{\nu\nu} & \bar{A}_{\gamma 2} \\ & \bar{A}_{2\nu} & \bar{A}_{22} \end{pmatrix} \begin{pmatrix} \delta U_1 \\ \delta U_\gamma \\ \delta \bar{U}_2 \end{pmatrix} = \begin{pmatrix} R_1 \\ R_\gamma + R_\nu + \bar{A}_{\nu\nu}(U_\nu - U_\gamma) \\ \bar{A}_{2\nu}(U_\nu - U_\gamma) \end{pmatrix}. \tag{4.6}$$

The system matrix in (4.6) corresponds to the final adaptive refinement step on processor 1, with possible modifications due to global fine mesh regularization. It is exactly the matrix used in the preliminary local solve to generate the initial guess for the global domain decomposition iteration. In the solution of (4.6), the components δU_1 and δU_γ contribute to the global solution update, while $\delta \bar{U}_2$ is discarded. We remark that the global iteration matrix corresponding to this formulation is not symmetric, even if all local system matrices are symmetric.

The domain decomposition algorithm is incorporated as the solver for the approximate Newton iteration described in Figure 4.1. In particular, only one domain decomposition iteration (a so-called *inner iteration*) is used in each approximate Newton step. Thus, loosely speaking, each solve of (4.6) alternates with a line search step in which the global solution is updated. The Newton line search procedure requires global communication to form some norms and inner products, as well as the boundary exchange described above.

4.4 Obstacle Problems.

When $IPROB = 2$, *PLTMG* solves the obstacle problem (1.8). The inequality constraints are treated via an interior point procedure [9]. In particular, we consider the Lagrange function

$$\rho(u_h) - \mu \sum_{i=1}^{NVF} d_i \left\{ \log(u_h(p_i) - \underline{u}(p_i)) - \log(\overline{u}(p_i) - u_h(p_i)) \right\} \tag{4.7}$$

where $\mu > 0$ is a small barrier parameter; the user specifies the target value in *RMTRGT*. Vertices of the triangulation are denoted by $p_i = (x_i, y_i)$, and d_i is the diagonal entry of the mass matrix corresponding to p_i. The weights $d_i = O(h_i^2)$ scale the barrier terms in a fashion similar to $\rho(u_h)$, and make μ independent of the mesh.

The overall solution strategy is to compute stationary points of the Lagrange function (4.7) for a decreasing sequence of $RMTRGT = \mu > 0$ values, following a smooth trajectory moving towards the boundary of the feasible region. This has much in common with the more general path following procedures used in the case $IPROB = 3$. If one alternates solution steps with adaptive refinement steps as in a typical adaptive feedback loop, one should generally reduce μ as $O(\sqrt{NVF})$ so that errors introduced by the continuation procedure are roughly comparable in size to the approximation errors introduced by the finite element discretization.

The assembly and solution procedures are quite similar to the case $IPROB = 1$. In particular, the right hand side is modified by terms of the form

$$-\mu d_i \left\{ (u_h(p_i) - \underline{u}(p_i))^{-1} + (u_h(p_i) - \overline{u}(p_i))^{-1} \right\},$$

and the diagonal of the stiffness matrix (Hessian matrix of the functional $\rho(u_h)$) is modified by terms of the form

$$\mu d_i \left\{ (u_h(p_i) - \underline{u}(p_i))^{-2} + (u_h(p_i) - \overline{u}(p_i))^{-2} \right\}.$$

With these modifications, the approximate Newton strategy described in Section 4.2 is used here.

When $IPROB = -2$, the domain decomposition algorithm outlined in Section 4.3 is used, with appropriate modifications to the stiffness matrix and right hand sides. As in the case $IPROB = -1$, only one domain decomposition solve (inner iteration) is used in each approximate Newton iteration.

As an example, we use *PLTMG* to solve the variational inequality

$$\min_{u \in K} \int_\Omega \{ |\nabla u|^2 - 2f(x,y)u \} dx\, dy$$

where the domain $\Omega = (0,1) \times (0,1)$, and

$$K = \left\{ u \in \mathcal{H}_0^1(\Omega) : |u| \le \frac{1}{4} - \frac{1}{10} \sin(\pi x) \sin(\pi y) \right\},$$
$$f(x,y) = -\Delta(\sin(3\pi x) \sin(3\pi y)).$$

In the absence of the obstacle, this is a simple elliptic equation with exact solution $u = \sin(3\pi x) \sin(3\pi y)$.

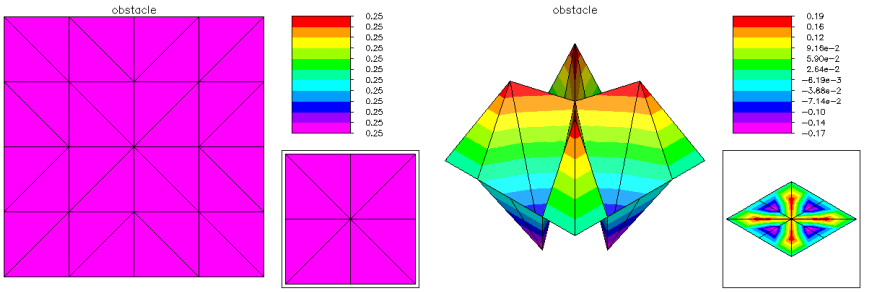
In this calculation, we began with a uniform 5×5 mesh, and adaptively refined towards a final mesh with $NVF = 16000$. On the initial mesh we took $\mu = 1$. At each refinement step, we first solved the problem with the existing value of μ and then reduced μ by a factor of 2 and solved a second time; $\mu = 2^{-7}$ in the final mesh. As previously mentioned, this strategy tries to balance discretization errors with the errors introduced by μ. In Figure 4.3, we show several of the meshes and corresponding solutions. Elements in the meshes are colored by size, and for the finer meshes, triangle edges are no longer drawn. We note the smallest elements occur near the boundaries of the contact zones, which are resolved as part of the computation; the largest elements appear in the interiors of the contact zones. In Figure 4.3, we also show the a posteriori error estimate for the finest mesh. Here we see that the error is relatively small in the contact zones. It is somewhat larger but reasonably uniform outside the contact patches, indicating that the adaptive procedure has done a good job in this example.

4.5 Continuation Problems.

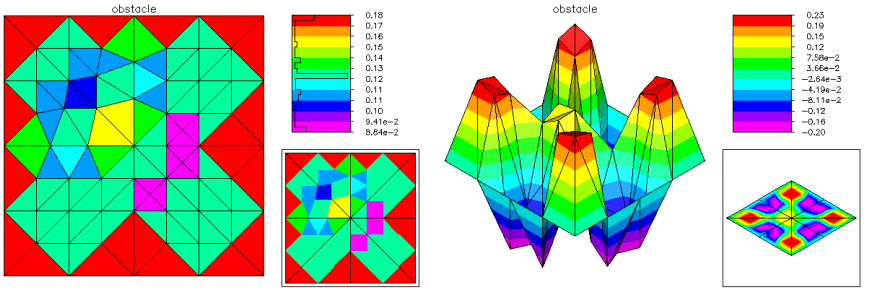
In the case of continuation problems ($IPROB = 3$), the parameter *ITASK* specifies the the continuation option. Available options are summarized in Table 4.2. For convenience in notation, we will systematically drop the subscript h from all variables in this section (e.g., λ_h will be denoted λ).

When the continuation process is used, we use a normalization equation of the form

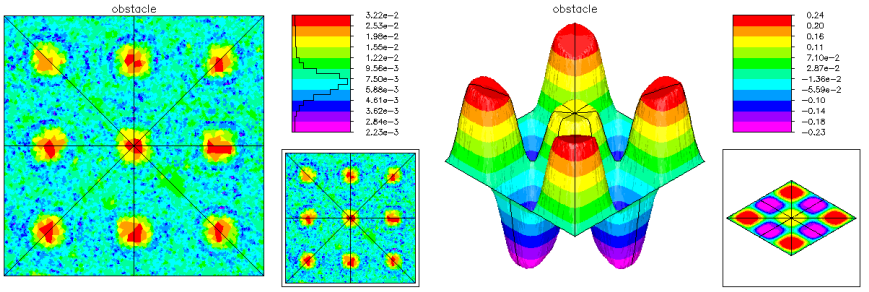
$$N(u, \lambda) = \sigma.$$



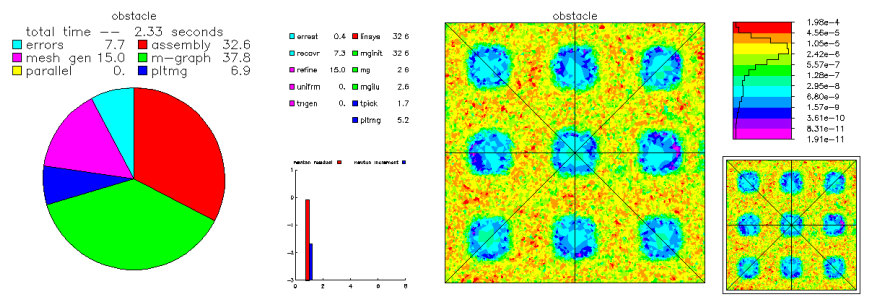
The initial uniform mesh and solution.



The first adaptive mesh with $NVF = 64$.



The final adaptive mesh with $NVF = 16000$.



The scalar $\sigma = SIGMA$ is the steplength. *PLTMG* uses two different normalization equations [5, 34]. Most frequently, we use

$$N(u,\lambda) = \theta\dot{\rho}_0(\rho - \rho_0) + (2-\theta)\dot{\lambda}_0(\lambda - \lambda_0). \tag{4.8}$$

Here $\theta = THETA$ is a parameter selected by *PLTMG*; by choosing θ and σ properly, it is possible to achieve target values in either ρ or λ. The vector (u_0^t, λ_0) is the current solution point and $(\dot{u}_0^t, \dot{\lambda}_0)$ the current unit tangent vector. The scalar $\dot{\rho}$ is defined formally using the chain rule for differentiation:

$$\dot{\rho} = \rho_u \dot{u} + \rho_\lambda \dot{\lambda}.$$

The other normalization used in *PLTMG* is based on the pseudo-arclength method, characterized by

$$N(u,\lambda) = \theta \int_\Omega \dot{u}_0(u - u_0)\,dx\,dy + (2-\theta)\dot{\lambda}_0(\lambda - \lambda_0). \tag{4.9}$$

The normalization (4.9) is used when (4.8) is not well defined (e.g., if $\dot{\rho}_0 = \dot{\lambda}_0 = 0$). For reasonable choices of ρ, these will be isolated points on the solution manifold, such as symmetry-breaking bifurcation points. In such instances, (4.9) is used on a temporary basis until (4.8) is defined again.

The values $0 \le ITASK \le 4$ embody the basic continuation path following options available in *PLTMG*. The values $5 \le ITASK \le 7$ are designed for updating the solution at a fixed point when the mesh has been changed by a call to *TRIGEN*.

An initial solution is provided by the user through subroutine *GDXY*. Thereafter, the continuation proceeds from the last successfully computed point. A brief outline of the basic continuation process ($ITASK = 0$ or $ITASK = 1$) is given in Figure 4.4.

Procedure Continue

C1 Begin with initial solution (u_0^t, λ_0) and tangent vector $(\dot{u}_0^t, \dot{\lambda}_0)$.
C2 compute the step σ for the normalization equation; predict $(u^t, \lambda) \leftarrow (u_0^t, \lambda_0) + \alpha(\dot{u}_0^t, \dot{\lambda}_0)$.
C3 correct $(u^t, \lambda) \leftarrow NWT(u^t, \lambda)$; compute ψ_ℓ, ψ_r, and ν; compute tentative $\dot{u}$ and $\dot{\lambda}$.
C4 if a singular point was detected and $ITASK = 1$, then go to **C7**.
C5 set $(u_0^t, \lambda_0) \leftarrow (u^t, \lambda)$ and $(\dot{u}_0^t, \dot{\lambda}_0) \leftarrow (\dot{u}^t, \dot{\lambda})$.
C6 if (u_0^t, λ_0) is a target point, then exit; else go to **C2**.
C7 compute the singular point using secant/bisection algorithm on $\nu(\sigma) = 0$; exit.

Figure 4.4.

PLTMG always returns with $(RLTRGT, RTRGT) = (RL, R) \equiv (\lambda, \rho)$. To continue with $ITASK = 0$ or $ITASK = 1$, the user specifies a target value for either $RTRGT$ or $RLTRGT$. If $RLTRGT \neq RL$, then *PLTMG* seeks a solution with $\lambda = RLTRGT$. Similarly, if $RTRGT \neq R$, then *PLTMG* seeks a solution with $\rho = RTRGT$.

A step σ and a predicted solution are computed on line C2. The predictor is a standard Euler type commonly used in continuation procedures. The step size calculation is influenced not only by the user request but also by imposed requirements that the predicted solution be sufficiently accurate. The procedures used in this portion of the calculation are described in detail in [16]. The solution is corrected on line C3. The correction process symbolized by the operator NWT involves the solution of a set of nonlinear equations, and is discussed in greater detail below.

PLTMG locates singular points by computing the smallest singular value ν of the Jacobian matrix. A modified inverse iteration procedure computes the left and right singular vectors ψ_ℓ and ψ_r corresponding to ν as part of each correction step C3. If the matrix is symmetric ($ISPD = 1$), then $\psi_\ell \equiv \psi_r$. In a somewhat nonstandard fashion for singular values, we normalize the singular vectors to have unit length and satisfy

$$\int_\Omega \psi_\ell \psi_r \, dx > 0.$$

Requiring the sign of the inner product of ψ_ℓ and ψ_r to be positive forces the singular value ν to change sign at a singular point (normally one requires $\nu \geq 0$ and then the inner product changes sign at singular points). Unfortunately, while ν changes sign in a continuous fashion at singular points, it can also change sign *discontinuously* at regular points. For example, in the linear eigenvalue problem, along the trivial branch ν will continuously pass through zero at each eigenvalue and will discontinuously change sign at some point *between* each consecutive pair of eigenvalues where the smallest singular value of the Jacobian changes from the preceding to the following eigenvalue.

If *PLTMG* detects a change in sign in ν along the solution curve between the starting point and target point, and if $ITASK = 1$, the computation of the target point is abandoned in favor of computation of the possible singular point. A secant/bisection algorithm for the equation $\nu(\sigma) = 0$ is used. More details of these procedures can be found in Bank and Chan [5] and the references therein. At the conclusion of this iteration, some tests are made to determine if the point is a bifurcation point, a limit point, or a regular point.

The algorithms in *PLTMG* were designed to handle only simple limit and bifurcation points, although on occasion we have observed them to work on some higher degree singular points as well. When a target or singular point has been successfully computed, *PLTMG* returns with $(RLTRGT, RTRGT)$ set to the current values of (λ, ρ).

If *PLTMG* is called with $ITASK = 2$ at a bifurcation point, parameters relevant for the continuation procedure are initialized for the bifurcating branch, but the solution itself remains unchanged. In the next call to *PLTMG* with $ITASK = 0$

or $ITASK = 1$, the continuation procedure will follow the bifurcating branch.

If *PLTMG* is called with $ITASK = 3$ or $ITASK = 4$, parameters relevant for the continuation procedure are reinitialized using the new parameter or functional; the solution itself remains unchanged. The two cases differ in that either λ or ρ can be held fixed during the reinitialization; for either case it is possible to specify either a new continuation parameter λ, a new functional ρ, or both.

The successful use of the continuation procedure requires guidance from the user. For example, it is possible to specify target values that cannot be reached. Also, since singular points are detected by changes in sign of ν, one can fool the singular-point detection algorithm by specifying target values sufficiently far away that two sign changes are passed on one step.

We now consider the cases $5 \leq ITASK \leq 7$. We begin by noting that the discretization process can introduce spurious solution curves or cause significant distortions in the solution curves of the continuous problem (1.1); one must therefore be cautious in interpreting the numerical results [33]. As the mesh is refined or the mesh points are smoothed, the solution curves generally will move; the assumption of *PLTMG* is that, as a function of the discretization, the solution curves converge in some uniform fashion to those of the continuous problem, and that the mesh is sufficiently fine to capture the qualitative features of the continuous problem's solution curves in the regions of interest [5, 15]. Typically, for each point on the current grid, there are three natural points on a nearby new grid solution curve that can be associated with it: the point with the same λ value ($ITASK = 5$), the point with the same ρ value ($ITASK = 6$), and the point of intersection with the perpendicular hyperplane passing through the current solution point ($ITASK = 7$). Some typical examples are illustrated in Figure 4.5.

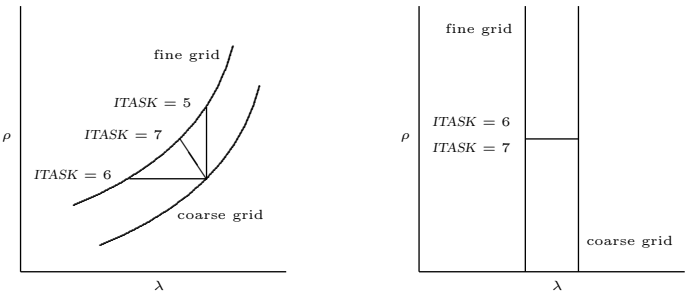


Figure 4.5. *The effect of ITASK in the case of mesh refinement.*

In some situations, all three points may not exist, or they may not be distinct. This is illustrated in Figure 4.5, right, where $ITASK = 6$ and $ITASK = 7$ correspond to the same fine grid point, while no (nearby) solution exists for $ITASK = 5$.

We now consider the linear algebraic aspects of the problem. As with other problem types, the nonlinear systems for $IPROB = 3$ are solved by the approximate Newton iteration [18, 17] described in Figure 4.1. The nonlinear system to be solved

has the form

$$G(u,\lambda) = 0,$$
$$N(u,\lambda) = \sigma.$$

Here the operator G represents the finite element equations of order NVF, and N the normalizing equation used in the continuation process; σ is the steplength. At each step of the Newton process, the linear system to be solved has the form

$$\begin{pmatrix} G_u & G_\lambda \\ N_u & N_\lambda \end{pmatrix} \begin{pmatrix} \delta u \\ \delta\lambda \end{pmatrix} = - \begin{pmatrix} G(u,\lambda) \\ N(u,\lambda) - \sigma \end{pmatrix}, \tag{4.10}$$

where δu is a vector of length NVF and $\delta\lambda$ is a scalar. The solution is constructed by solving

$$\begin{aligned} G_u v &= -G, \\ G_u w &= -G_\lambda - c_0 G_u \dot{u}, \\ \delta\lambda &= -\frac{N_u v + N - \sigma}{N_u(w + c_0\dot{u}) + N_\lambda}, \\ \delta u &= v + \delta\lambda(w + c_0\dot{u}). \end{aligned}$$

The coefficient $c_0 = 1/\dot{\lambda}$, provided $\dot{\lambda} \neq 0$; $c_0 = 0$ otherwise. Thus the right-hand side $G_\lambda + c_0 G_u \dot{u}$ has the appearance of a residual, and w may be viewed as an incremental update. In particular, the vector $w + c_0\dot{u}$ is proportional to the next tangent vector at convergence. The tentative tangent $\dot{u}$ is thus easily updated at every Newton step, along with u, λ, ρ, ψ_r, and ψ_ℓ and is available for regularizing the right-hand side for w on the next Newton step. The linear systems involving G_u and G_u^t are solved by the multigraph algorithm.

The block elimination process is embedded in the overall damped Newton process [16, 17] given in Figure 4.1. Here $\mathcal{U}_k^t = (u^t, \lambda)$ is the kth Newton iterate, $\delta\mathcal{U}_k^t = (\delta u^t, \delta\lambda)$, and $\mathcal{G}_k^t = (G^t, N - \sigma)$. The norm $\|\mathcal{G}_k\|$ is given by

$$\|\mathcal{G}_k\|^2 = \|G\|^2 + c|N - \sigma|^2,$$

where c is a scaling parameter ($SCALE$ in the RP array) chosen to balance the two terms appropriately.

The case $IPROB = -3$ corresponds to a parallel solve of the block linear system (4.10), embedded in the overall Newton iteration. It is defined only for the cases $ITASK = 5, 6, 7$; at present there is no parallel implementation of the basic path following options. Thus we assume that the continuation is done on a coarse mesh on a single processor, and parallel computation is used only in the context of computing a highly refined solution at a particular point.

For continuation problems, *PLTMG* provides a limited amount of written output summarizing the state of the computation. All formats are designed for output devices having a minimum of 80 characters per line. All output is directed to the subroutine $FILUTL$, which is responsible for creating the files *BFILE* and *JWFILE*.

For each call to *PLTMG* a banner is printed. Each continuation step results in a single line of output containing seven numbers. A typical example of such output is illustrated below:

```
pltmg:     lambda        rho       lambda dot    rho dot     eigenvalue
  0  3  0.99004E+01  0.39814E+01 -0.80768E-02  0.39890E+01 -0.94673E-04
```

The first column contains the current value of *IFLAG* (in this example, $IFLAG = 0$). The second contains the value of *ITNUM*, the actual number of approximate Newton iterations used. The next four columns contain the current values of the parameter λ, the functional ρ, and their derivatives with respect to arclength along the current solution manifold $\dot{\lambda}$ and $\dot{\rho}$. The column labeled "eigenvalue" gives an approximation to the smallest singular value ν of the Jacobian matrix $\mathcal{G}_u$.

As an example, we consider the nonlinear eigenvalue problem

$$\begin{aligned} -\Delta u &= \lambda \sin u \quad \text{in } \Omega \equiv (0,1) \times (0,1), \\ u &= 0 \qquad\quad \text{on } \partial\Omega, \end{aligned}$$

with the functional given by

$$\rho(u, \lambda) = \int_\Omega u^2 \, dx \, dy.$$

This problem has bifurcation points at the eigenvalues of the linear eigenvalue problem, $-\Delta u = \lambda u$, which are given by $\lambda_{k\ell} = (k^2 + \ell^2)\pi^2$, $k = 1, 2, \ldots$, $\ell = 1, 2, \ldots$. We chose as our coarse mesh a 17×17 uniform mesh.

Our goal is to compute the first four eigenvalues and eigenfunctions. The first and third eigenvalues have multiplicity one. The second and fourth eigenvalues have multiplicity two. While the algorithms in *PLTMG* are not designed to handle multiplicities greater than one, the code performed in a satisfactory fashion and computed all four eigenvalues without difficulty. As a cautionary remark, one should not assume that the situation in this respect will always be so favorable.

We initialize at $\lambda = 0$ and continue to $\lambda = 10$ with $ITASK = 0$ and then to $\lambda = 22$ with $ITASK = 1$. At $\lambda = 22$, the sign of ν (eigenvalue) has changed, so *PLTMG* computes the singular point, in this case the first eigenvalue.

```
pltmg:     lambda        rho       lambda dot    rho dot     eigenvalue
  0  1  0.00000E+00  0.00000E+00  0.10000E+01  0.00000E+00  0.76859E-01
  0  2  0.10000E+02  0.00000E+00  0.10000E+01  0.00000E+00  0.38191E-01
  0  2  0.22000E+02  0.00000E+00  0.10000E+01  0.00000E+00 -0.82128E-02
pltmg: find limit / bifurcation point
  0  2  0.19876E+02  0.00000E+00  0.10000E+01  0.00000E+00 -0.35329E-06
pltmg: probable bifurcation point
  0  0  0.19876E+02  0.00000E+00  0.10000E+01  0.00000E+00 -0.35329E-06
```

Note that the secant/bisection algorithm converged in one step. After determining that the singular point was a bifurcation point, *PLTMG* makes an additional calculation to ensure that the tangent vector $\dot{u}_h$ corresponds to the current branch (in this case, the trivial branch).

We save the solution in a file in order to continue from this point to the second eigenvalue in a convenient manner (see Section 6.7), and switch branches ($ITASK = 2$). We then routinely continue on the bifurcating branch in several steps ($\rho = .01, \lambda = 25, 50, 100, 150, 300, 500$). At $\lambda = 500$, we refine the mesh with a call to *TRIGEN* ($IADAPT = 1$, $NVTRGT = 1000$), creating a mesh with $NVF = 1000$. We follow with a call to *PLTMG* with $ITASK = 7$. The eigenfunction and mesh are shown in Figure 4.6.

We restore the solution at the bifurcation point and continue along the trivial branch to the second eigenvalue. We save the solution, switch branches and explore the bifurcating branch in a fashion similar to the first eigenvalue. A similar procedure is repeated for the third and fourth eigenvalues. The eigenfunctions computed on a refined mesh of $NVF = 1000$ are shown in Figure 4.6. In Figure 4.7, we show the complete history of the calculation in terms of the continuation path.

4.6 Parameter Identification Problems.

When $IPROB = 4$, *PLTMG* solves the parameter identification problem (1.9)-(1.12) The simple bounds on λ_h are treated in a fashion analogous to the case $IPROB = 2$. In particular, we consider the Lagrangian

$$\rho(u_h, \lambda_h) + a(u_h, v_h) - \mu \left\{ \log(\lambda_h - \underline{\lambda}) - \log(\overline{\lambda} - \lambda_h) \right\} \tag{4.11}$$

where $\mu > 0$ is the barrier parameter and v_h is the Lagrange multiplier (a member of the finite element subspace). Our procedure computes a stationary point of the Lagrangian (4.11) using an approximate Newton method.

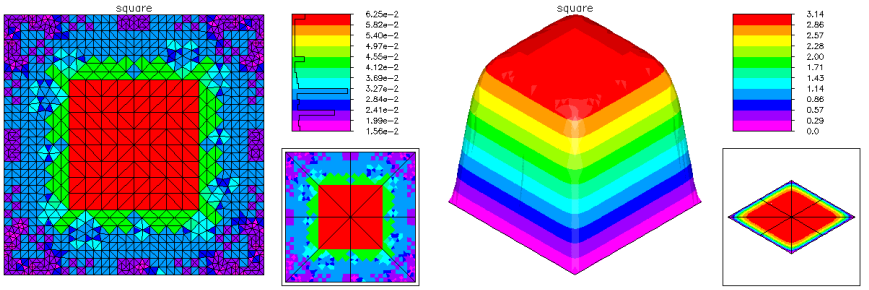
The linear algebra problem at each Newton iteration is of the form

$$\begin{pmatrix} H & A^t & C_u \\ A & 0 & C_v \\ C_u^t & C_v^t & D \end{pmatrix} \begin{pmatrix} \delta u \\ \delta v \\ \delta\lambda \end{pmatrix} = \begin{pmatrix} b_u \\ b_v \\ b_\lambda \end{pmatrix}. \tag{4.12}$$

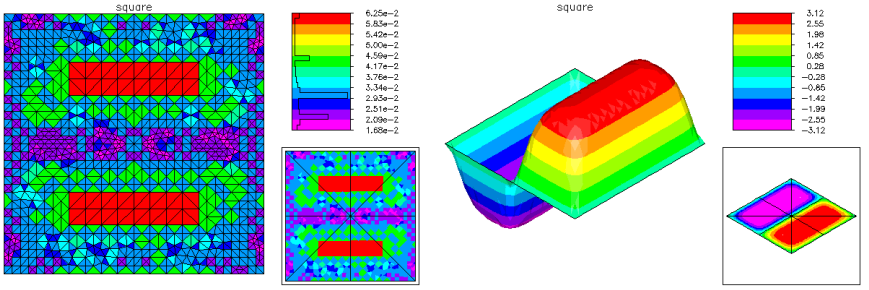
Here the matrix A is the Jacobian matrix corresponding the the bilinear form $a(u_h, v_h)$. In particular, linear systems involving A (or A^t) are solved using the multigraph iteration. The matrix H is symmetric and has the same sparsity pattern as A; other characteristics strongly depend on the particular problem. C_u and C_v are generally dense column vectors, and D is a scalar. The vectors δu and δv are the (Newton) updates for u_h and the Lagrange multiplier v_h, respectively, and $\delta\lambda$ is the scalar (Newton) update for λ_h. b_u, b_v and b_λ correspond to the appropriate Newton residuals.

Our solution algorithm is just standard block elimination, with a small algebraic modification that reduces the number of solves with A or A^t from 4 to 3. Here we summarize the procedure. First we solve

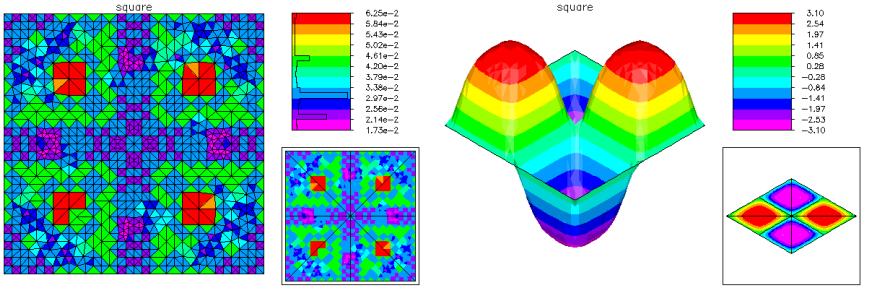
$$A\bar{b}_v = b_v,$$
$$A\bar{C}_v = C_v.$$



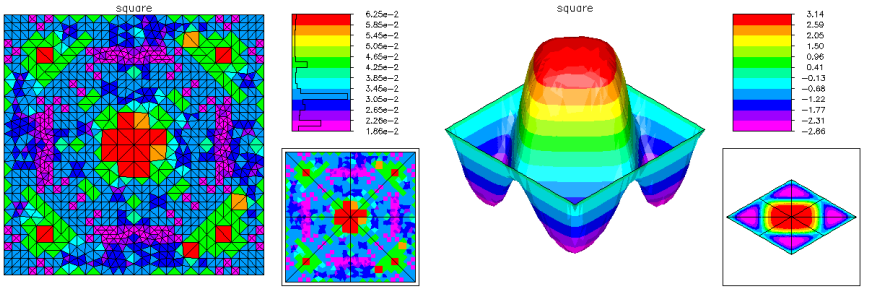
The first eigenfunction.



The second eigenfunction.



The third eigenfunction.



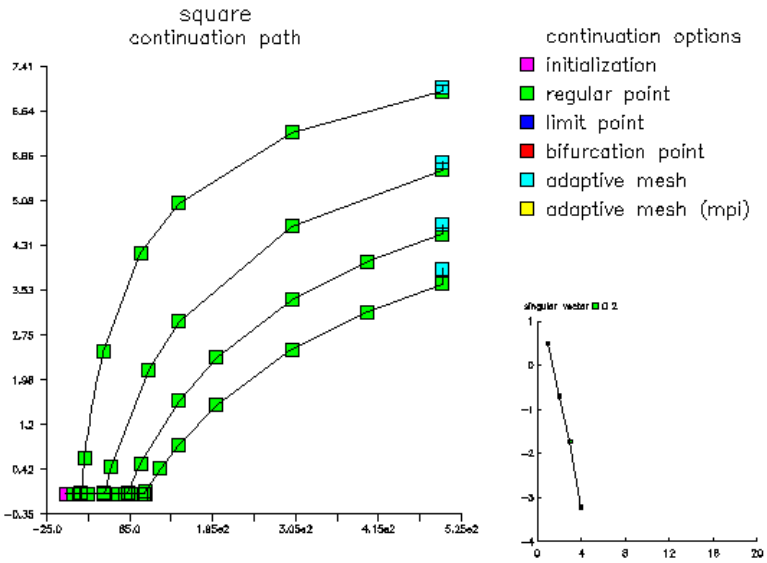


Figure 4.7. *The continuation path.*

Both systems are (approximately) solved using the multigraph iteration. Then we form

$$\bar{b}_u = b_u - H\bar{b}_v,$$
$$\bar{C}_u = C_u - H\bar{C}_v,$$

which requires two sparse matrix multiplications with H. Next we compute $\delta\lambda$ using the (scalar) Schur complement

$$\delta\lambda = \frac{b_\lambda - C_u^t\bar{b}_v - \bar{C}_v^t\bar{b}_u}{D - C_u^t\bar{C}_v - \bar{C}_v^t\bar{C}_u}.$$

We then form δu from

$$\delta u = \bar{b}_v - \bar{C}_v\delta\lambda,$$

and find δv from

$$A^t\delta v = \bar{b}_u - \bar{C}_u\delta\lambda.$$

The latter requires the use of the multigraph iteration for a third time. The basic Newton iteration is again that given in Figure 4.1 with the interpretation $\mathcal{U}^t = (u_h^t, v_h^t, \lambda_h)$ and $\mathcal{G}^t = (b_u^t, b_v^t, b_\lambda)$.

The parameter identification problem has one additional option, specified by $ITASK = 8$. If the problem has more than one scalar control parameter, one can switch parameters, sequentially optimizing the solution with respect to one parameter with the others held fixed. If $ITASK = 8$, *PLTMG* reinitializes certain variables that depend on λ before starting the Newton iteration.

When $IPROB = -4$, a parallel Newton algorithm is implemented, similar in structure to the case $IPROB = -1$. A domain decomposition solver analogous to that described in Section 4.3 is incorporated into the block elimination algorithm defined above.

As an example, we consider the problem

$$\min \int_\Omega e^{-20(x^2+y^2)}(u-1)^2\,dx,$$

subject to the boundary value problem

$$\begin{aligned} -(1+\lambda^2)\Delta u &= 1 && \text{in } \Omega \\ \nabla u \cdot n &= 0 && \text{on } \partial\Omega_1 \\ u &= 0 && \text{on } \partial\Omega_2 \end{aligned}$$

and the inequality constraints

$$0 \le \lambda \le 2.$$

The domain was provided as a skeleton and the initial mesh generated by *TRIGEN*. Both are shown in Figure 4.8. The initial mesh was then refined in five steps to a final mesh with $NVF = 10001$ vertices. On this final mesh $\lambda = .9931$. The final mesh, the solution, and the Lagrange multiplier are shown in Figure 4.8. The interior point parameter μ was set to $\mu = 1$ for the initial mesh, and reduced by a factor of 2 at each mesh refinement step, yielding $\mu = 10^{-5}$ on the finest mesh.

4.7 Control Problems.

When $IPROB = 5$, *PLTMG* solves the control problem (1.13)-(1.16). This problem is similar to the case $IPROB = 4$ except that now λ_h is a piecewise linear polynomial rather than a scalar. Here we consider the Lagrangian

$$\rho(u_h,\lambda_h)+a(u_h,v_h)-\mu\sum_{i=1}^{NVF} d_i\left\{\log(\lambda_h(p_i)-\underline{\lambda}(p_i))-\log(\overline{\lambda}(p_i)-\lambda_h(p_i))\right\} \quad (4.13)$$

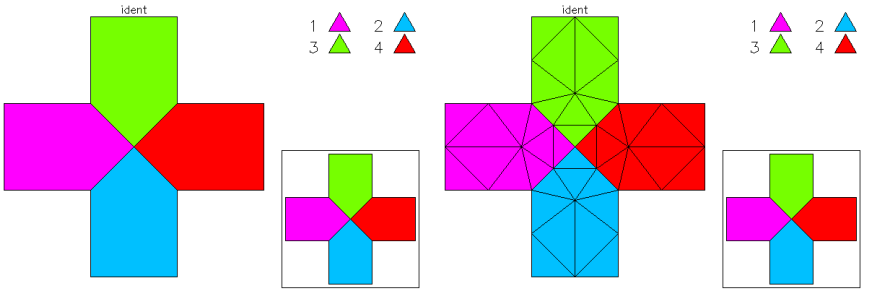
where $\mu > 0$ is the barrier parameter, d_i is the diagonal of the mass matrix corresponding to vertex p_i, and v_h is the Lagrange multiplier. As usual, our algorithm seeks a stationary point of the Lagrangian (4.13) using an approximate Newton method.

The linear algebra problem at each Newton step is of the form

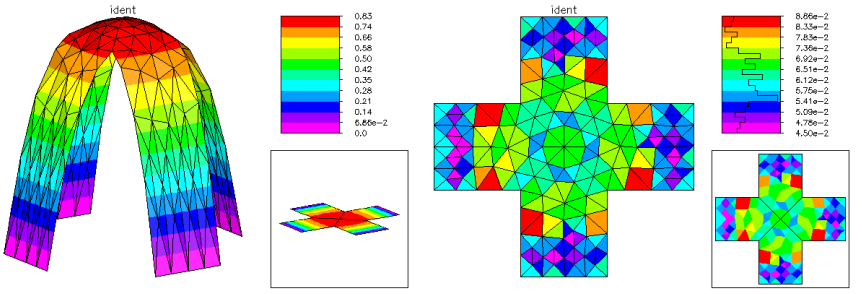
$$\begin{pmatrix} H & A^t & S_u \\ A & 0 & S_v \\ S_u^t & S_v^t & G \end{pmatrix} \begin{pmatrix} \delta u \\ \delta v \\ \delta\lambda \end{pmatrix} = \begin{pmatrix} b_u \\ b_v \\ b_\lambda \end{pmatrix}. \quad (4.14)$$

Here H and A are defined as before. In typical problems G is a symmetric, positive definite matrix. Often the functional ρ contains regularization terms for the control function λ_h. For example, if the regularization term is of the form

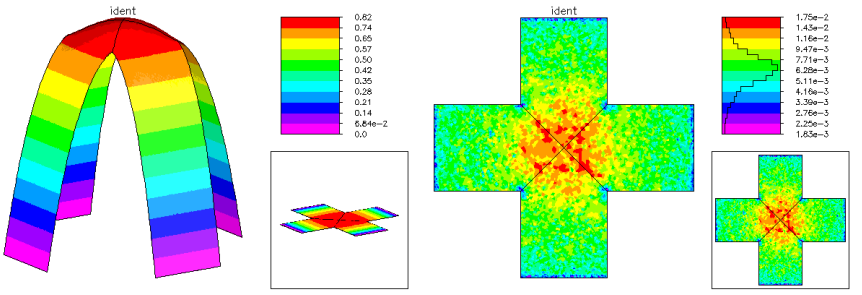
$$\gamma \int_\Omega \lambda^2\,dx,$$



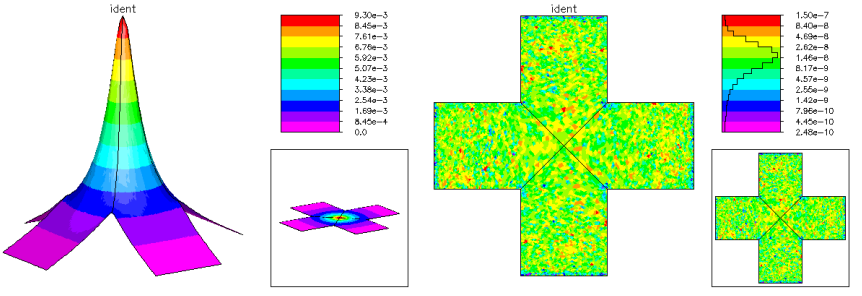
The skeleton and initial mesh with $NVF = 33$.



The solution and first adaptive mesh with $NVF = 152$.



The solution and final adaptive mesh with $NVF = 10001$.



then the corresponding contribution to G is γM, where M is the mass matrix. As before, linear systems involving A and A^t are easily solved using the multigraph iteration. Additionally, linear systems involving G are approximately solved using a simple symmetric Gauss-Seidel iteration with conjugate gradient acceleration. The matrices S_u and S_v have the same symmetric sparsity structure as G and A, but are generally not symmetric.

Our solver is based on block Gaussian elimination, similar to the case $IPROB = 4$. However, in the case of (4.14), it is too expensive to compute an exact Schur complement for the $3,3$ block; instead we approximate the Schur complement by G. Thus, our solution algorithm is really just a preconditioner. In particular, it is one step of a block symmetric Gauss-Seidel iteration. This is realized as follows:

$$\begin{aligned} A\tilde{c}_u &= b_v, \\ A^t\tilde{c}_v &= b_u - H\tilde{c}_u, \\ G\delta\lambda &= b_\lambda - S_u^t\tilde{c}_u - S_v^t\tilde{c}_v, \\ A\delta u &= b_v - S_v\delta\lambda, \\ A^t\delta v &= b_u - H\delta u - S_u\delta\lambda. \end{aligned}$$

Linear systems involving A and A^t are solved using the multigraph iteration, and that involving G is solved by SGS/CG. If G were replaced by the Schur complement and all linear systems solved exactly, this would yield the exact solution.

When $IPROB = -5$, a parallel Newton algorithm is implemented, similar in structure to the case $IPROB = -1$. A domain decomposition solver analogous to that described in Section 4.3 is incorporated into the block preconditioner defined above. As in the case $IPROB = 5$, only one block SGS iteration is used.

As an example, we solve the optimal control problem

$$\min \int_\Omega (u - u_0)^2 + \gamma\lambda^2 \, dx$$

subject to the constraint equation

$$\begin{aligned} -\Delta u &= \lambda \quad \text{in } \Omega \equiv (0,1) \times (0,1), \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned}$$

and the inequalities

$$1 \le \lambda \le 10.$$

The target function u_0 was

$$u_0 = \sin(3\pi x)\sin(3\pi y)$$

and the regularization parameter $\gamma = 10^{-4}$.

This problem was solved in parallel using eight processors. An initial 9×9 uniform mesh was adaptively refined to $NVF = 1000$. See Figure 4.9. This mesh was load balanced and broadcast to the eight processors. Each processor then adaptively refined its partition to $NVF \approx 4000$, yielding a global refined mesh with

$NVG = 16668$. The interior point parameter $\mu = \mu_0 = .01$ on the 9×9 mesh, and thereafter was reduced by a factor of 2 in each refinement step, both in creating the mesh with $NVF = 1000$ that was used for the load balance, and in the subsequent parallel adaptive refinement steps. The final $\mu = \mu_0 2^{-4}$. The global refined mesh, solution u, Lagrange multiplier v, and the control function λ are shown in Figure 4.9, along with the a posteriori error estimate and some timing data.

4.8 Subroutine PLTEVL.

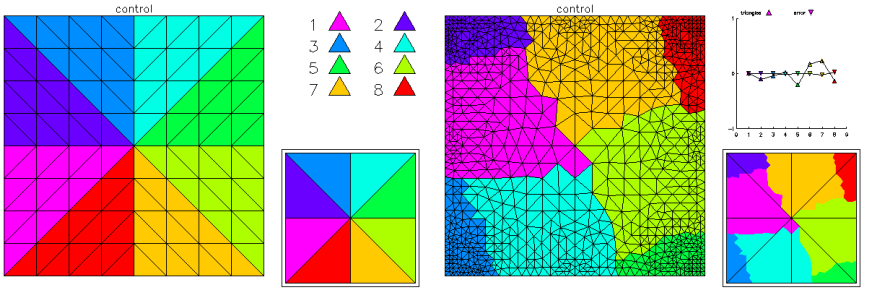
Subroutine *PLTEVL* evaluates the solution and its gradient at a list of user specified evaluation points. *PLTEVL* is called using the statement

Call PLTEVL(X, Y, U, UX, UY, VX, VY, XM, YM,
ITNODE, IBNDRY, IP, RP, W)

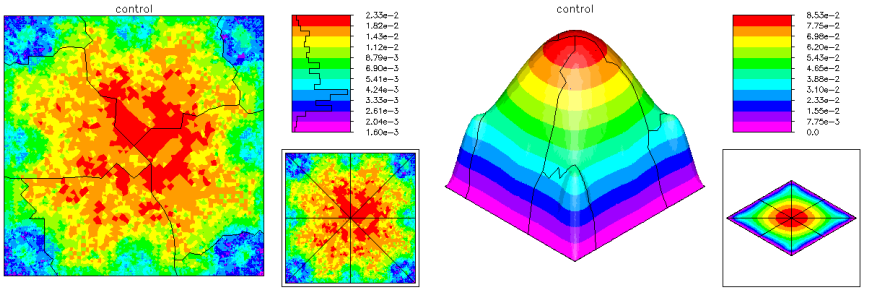
The arrays *VX*, *VY*, *XM*, *YM*, *ITNODE*, and *IBNDRY* define a triangulation. $|NEVP|$ is the number of evaluation points. If $NEVP > 0$, *PLTEVL* carries out some relatively expensive initialization and then evaluates the function and gradient. If $NEVP < 0$, *PLTEVL* assumes the initialization has been done on a previous call (with no intervening calls to other routines in the package), and will bypass the initialization. The arrays X and Y are of length $|NEVP|$, with *(X(I),Y(I))* being the Ith evaluation point. The output arrays U, UX, and UY are of size $|NEVP|$, with *U(I)* containing the function value and *(UX(I),UY(I))* the gradient value at the Ith evaluation point. Since the gradient is piecewise constant, it is not uniquely defined along internal triangle edges and at vertices. At such evaluation points a representative (arbitrary) assignment is made from among the possibilities. If a given evaluation point lies outside the domain Ω, the corresponding function and gradient values are set to the minimum value of the function.

The main problem in evaluating a grid function at an arbitrary point (x, y) is determining which element contains the point. Since the meshes in *PLTMG* are generally unstructured and nonuniform, this requires searching and testing lists of elements. *PLTEVL* has an expensive initialization phase where elements are sorted to minimize this searching.

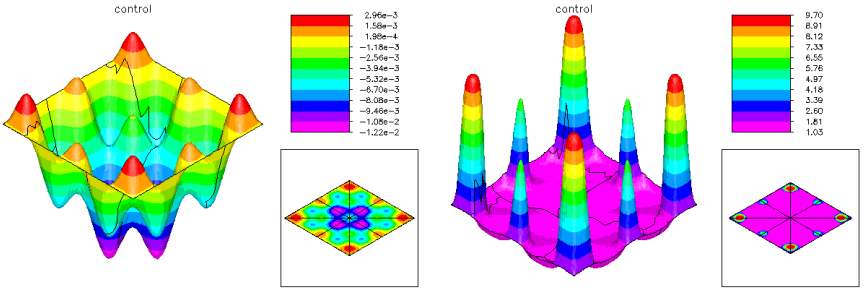
This is done by assigning each triangle to a node in a binary tree. We begin by embedding the entire mesh in a rectangle that becomes the root node of the tree. The root rectangle is then bisected, either horizontally or vertically, by connecting a pair of opposing midpoints. This bisection splits the list of triangles into three groups: those completely in the left (top) rectangle, those completely in the right (bottom) rectangle, and a third group (ideally small) that have nontrivial intersections with both rectangles. The decision whether to divide horizontally, vertically, or not at all depends mainly on the size of this last group relative to the other two. In any event, if a refinement is made, the two new leaves inherit the lists of elements completely contained in their corresponding rectangles, and the third group of elements remains associated with the father element. The leaves (son rectangles) then become candidates for further bisection. The overall process creates the binary



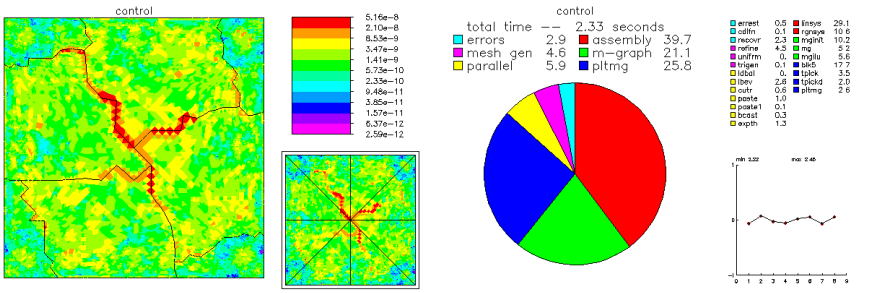
The initial 9×9 *mesh and the load balance* ($NVF = 1000$).



The global refined mesh with $NVG = 16668$ *and the corresponding solution.*



The Lagrange multiplier v *and the control function* λ.



tree, in which each node is a rectangle, and associated with each node is a short list of triangles.

The point $(x, y) \in t$, where t is an element of the triangulation, if and only if all its barycentric coordinates with respect to t are nonnegative (this test is modified slightly for a triangle with a curved boundary edge). The evaluation of the barycentric coordinates requires the assembly and solution of a 3×3 set of linear equations

$$\begin{pmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 1 \\ x \\ y \end{pmatrix},$$

where (x_i, y_i) are the vertices of t.

The evaluation of a function at the point (x, y) uses two different strategies. In the first, we find a triangle t associated with the leaf of the tree whose rectangle contains the point (x, y); this is done by following a path in the binary tree from the root to the desired leaf.

We evaluate the barycentric coordinates of (x, y) with respect to t; if all are nonnegative, we are done. If one (or two) coordinates are negative, we locate the neighbor element to t corresponding to a negative coordinate; this element is closer to (x, y) than t itself. We then replace t by its neighbor and repeat the test on the new element. In this way we map out a fairly direct path from the seed element to the element that contains the point. Since the seed triangle was associated with the leaf of the binary tree containing the point, we expect the path to contain few elements.

This strategy fails if at some step there is no neighbor element, i.e., we arrive at the boundary. If the domain Ω is convex, this implies the point (x, y) is not in Ω. Since we make no convexity assumption on Ω, it could also mean for example, that we have arrived at a crack and the point is in an element on the other side of the crack. Thus, if the first strategy fails, we build a list of all elements that might contain the point. This is done by marching down the binary tree from the root to the leaf containing the point. The lists of triangles associated with all the nodes along this path are joined to form the list for the given point. This list is checked, beginning with those triangles associated with the leaf, and continuing through the tree towards the root. In this process, either we find an element containing the point or we exhaust the list and conclude that (x, y) is not in Ω. In practice, this second strategy is required infrequently, even if Ω is not convex.

Chapter 5
Graphics

5.1 Overview.

The graphics package associated with *PLTMG* is composed of subroutines *TRIPLT*, *INPLT*, *GPHPLT*, and *MTXPLT*. These routines are written in self-contained, portable Fortran, addressing the graphics output device through subroutines *PLINE*, *PFILL*, *PFRAME*, and *PLTUTL*. The specifications for these routines are given in Section 6.12.

Typical graphics output consists of three windows or frames. There is a large square window on the left, and two smaller square windows on the right. The main image typically appears in the large frame, and other useful information (for example, a legend matching colors to function values) appears in the smaller frames. The graphics interface now provides z-buffer information, for use in three dimensional imaging systems such as OpenGL. All the graphics routines are written such that the image appearing in the main window can be animated using such graphics systems when appropriate.

Subroutine *TRIPLT* graphs the solution and various associated functions (e.g., $\dot{u}$, ψ_r, ϵ_t). *TRIPLT* also has options for plotting vector functions (e.g., ∇u_h). Subroutine *INPLT* can display either a triangulation or a skeleton, with elements or regions colored according to various attributes such as the quality of the elements in a triangulation. Subroutine *GPHPLT* displays various graphs and charts containing timings, convergence histories, and other items of interest. Subroutine *MTXPLT* displays several sparse matrices associated with the solution process.

The parameter *MXCOLR* is a device dependent constant, stating the maximum number of colors available for use by the graphics package. We assume that $2 \leq MXCOLR$. While it is possible to make some interesting plots and contour maps with *TRIPLT* using only monochrome devices ($MXCOLR = 2$), *TRIPLT* makes extensive use of available color facilities in producing (shaded) three-dimensional surface plots and vector plots. *GPHPLT*, *MTXPLT*, and *INPLT* also use color, but in less critical ways.

Subroutines *TRIPLT*, *INPLT* and *GPHPLT* offer some capabilities for parallel

processing. In the parallel processing environment, only the master process (corresponding to $IRGN = 1$) makes calls to the graphics interface routines (*PLTUTL*, *PFRAME*, *LINE*, and *PFILL*. However, in the case of *TRIPLT* or *INPLT*, one may wish to plot the solution, error, or some other function in situations where the data is distributed among the processors. If MPI is turned on ($MPISW = 1$), then *TRIPLT* and *INPLT* collect data from all other processors, and draw a composite picture consisting of the union of the refined regions for each processor. If the problem is sufficiently large that it is impossible or inefficient to collect all the data on a single processor, each processor can coarsen its data before sending it to the master process. This coarsening process is controlled by the parameters *ICRSN* and *ITRGT*. If MPI is turned off ($MPISW = -1$), then *TRIPLT* and *INPLT* draw the function on processor one (refined in region one and coarse elsewhere). For some options, *GPHPLT* collects data from all processors when MPI is turned on, for example in presenting timing and load balancing data. Subroutine *MTXPLT* currently has no parallel processing capabilities.

For most of the examples of graphics output, we solved Laplace's equation in a circle of radius one with a crack along the positive x axis. This domain was used to illustrate the triangulation data structure in Section 2.2. Nonhomogeneous Dirichlet boundary conditions were imposed on the circular boundary such that the true solution is $u = r^{1/4}\sin(\theta/4)$, the leading term in the singularity due to the crack tip. Some example output in Section 5.4 came from other problems, in cases where it could not be provided by our simple example.

5.2 Subroutine TRIPLT.

TRIPLT is called using the statement

Call TRIPLT(VX, VY, XM, YM, ITNODE, IBNDRY, IP, RP, SP, W, QXY)

The arrays *VX*, *VY*, *XM*, *YM*, *ITNODE*, and *IBNDRY* should define a triangulation. *TRIPLT* uses several variables from the *IP*, *RP*, and *SP* arrays, as shown in Tables 2.5–2.7. The string variable *FTITLE* is the character string displayed as a label above the graph. Additionally, *TRIPLT* uses the work array *W* and the Fortran subroutine *QXY*. Subroutine *QXY* is documented in Section 2.5. The error flag *IFLAG* is set as in Table 2.8 if there is insufficient storage.

The parameter *IFUN* specifies the function to be plotted. The available options are summarized in Table 5.1. Some of these functions are not defined for all problem types. Although there are many possibilities for *IFUN*, they may be classified as *surface plots* and *vector plots*.

For surface plots, all functions are continuous with the (possible) exceptions of the error, which is piecewise constant on triangles, and *QXY*, which can be multivalued at vertices due to discontinuities in ∇u_h. If desired, a discontinuous function can be mapped to a continuous function using a local averaging technique. This is invoked by setting the switch $ICONT = 1$.

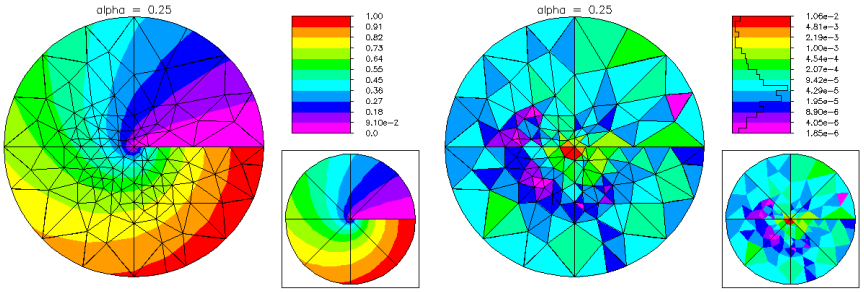


Figure 5.1. *The solution IFUN* $= 0$ *and the error IFUN* $= 5$.

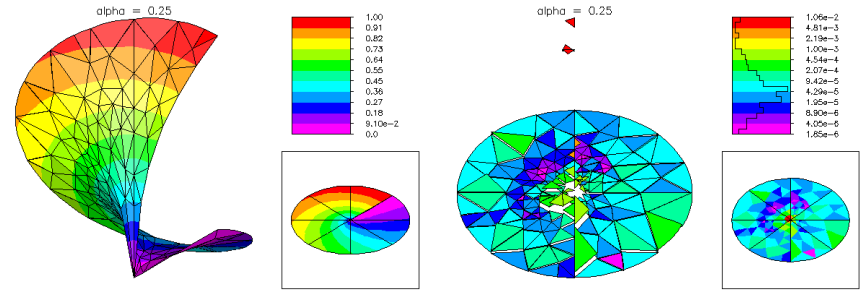


Figure 5.2. *The case IFUN* $= 0$, $(NX, NY, NZ) = (1, -1, -1)$, *and IFUN* $= 5$, $(NX, NY, NZ) = (1, 1, 1)$.

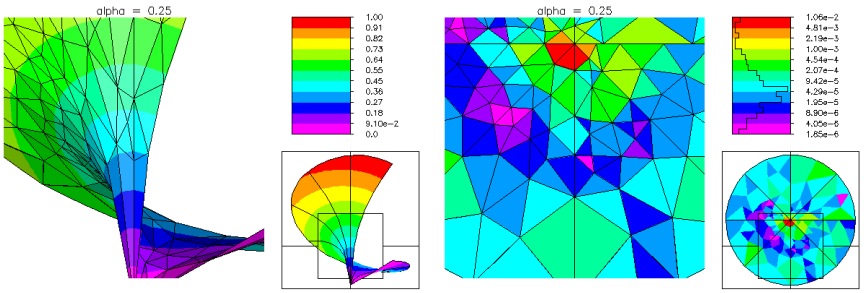


Figure 5.3. *The case IFUN* $= 0$, $(NX, NY, NZ) = (1, -1, -1)$, $RMAG = 2$, $CENX = .5$, $CENY = .3$, *and the case IFUN=5,* $(NX, NY, NZ) = (0, 0, 1)$, $RMAG = 2$, $CENX = .5$, $CENY = .3$.

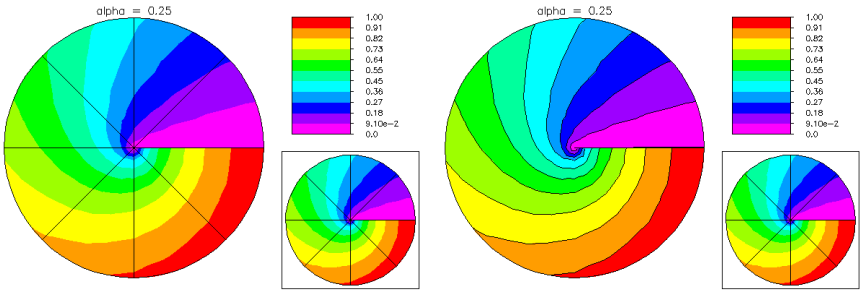


Figure 5.4. *The case LINES* $= 1$ *and the case LINES* $= 3$. *The corresponding picture for LINES* $= 0$ *is in Figure 5.1.*

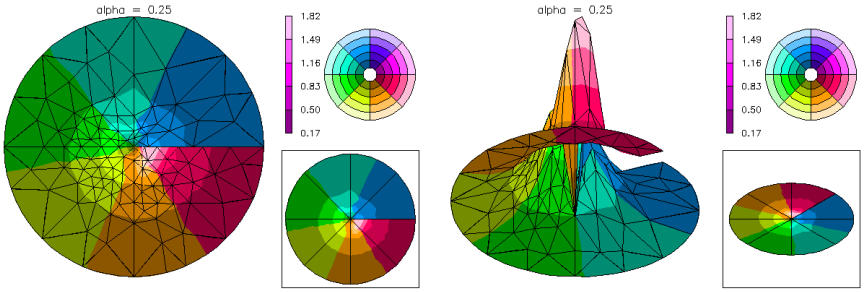


Figure 5.5. *The case IFUN* $= 2$. $(NX, NY, NZ) = (0, 0, 1)$ *and* $(NX, NY, NZ) = (1, -1, -1)$.

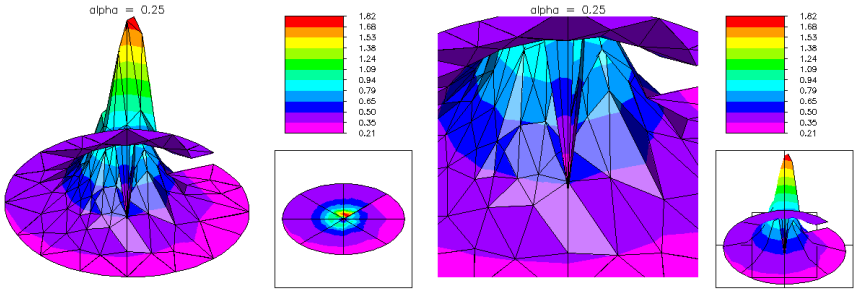


Figure 5.6. *The case IFUN* $= 3$, $(NX, NY, NZ) = (1, -1, -1)$. *In the picture on the right* $RMAG = 2$, $CENX = .5$, *and* $CENY = .3$.

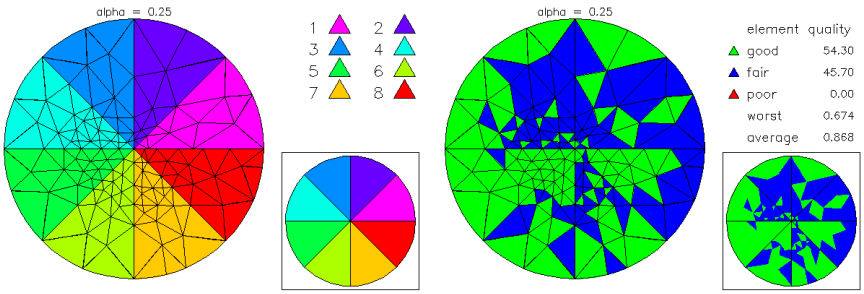


Figure 5.7. *Triangles colored by label (INPLSW* $= 0$*) and by quality (INPLSW* $= 2$*).*

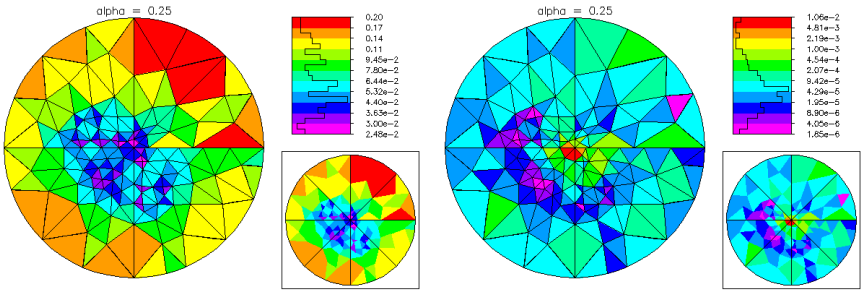


Figure 5.8. *Triangles colored by size INPLSW* $= 5$*) and by error (INPLSW* $= 6$*).*

IFUN	displayed function
0	the solution u_h
1	the scalar function $\mid S^m Q_h \nabla u_h \mid$
2	the vector function $S^m Q_h \nabla u_h$
3	the alternate function QXY
4	the alternate vector function QXY
5	the error estimate $\|\epsilon_t\|_{\mathcal{H}^1(t)}$
6	the tangent function $\dot{u}$
7	the right singular vector ψ_r
8	the left singular vector ψ_ℓ
9	the Lagrange multiplier u_m
10	the control variable λ_h
11	the dual function

Table 5.1. *The values of IFUN.*

5.2.1 Surface Plots.

In the case of surface plots, *NCON* specifies the number of contours (colors) to be used. If $NCON > MXCOLR - 2$, some colors are used for more than one contour. The parameters *SMIN* and *SMAX* can be used to specify the limits of the color scale. If $SMIN < SMAX$, then these values are used as limits, with parts of the function lying outside $(SMIN, SMAX)$ colored white. Otherwise, the largest and smallest values of the displayed function are used as limits.

Each picture consists of three frames; a large plot on the left and a two-part legend on the right. The upper right contains a scale relating colors to function values; three scales are available using the switch *ISCALE* as described in Section 5.2.4. For the case $IFUN = 5$, $ICONT = 0$, a histogram showing the distribution of errors $\|\epsilon_t\|_{\mathcal{H}^1(t)}$ is also provided in this legend. Four line-drawing options using *LINES* and eight labeling options using *NUMBRS* are also available. *RMAG*, *CENX*, and *CENY* provide a zoom-in capability as described in Section 5.2.3. The main image can be animated using three dimensional imaging systems such as OpenGL.

The triple $d = (NX, NY, NZ)$ specifies the viewing perspective. The three-dimensional surface is projected into the plane orthogonal to d, and the function is drawn as it would appear to an observer viewing the surface from a line of sight parallel to d. The vectors (NX, NY, NZ) and $-(NX, NY, NZ)$ cause the same projection to be computed; however, different pictures are generally produced for the two cases. In the former case one observes the projection on the "front" of the plane, and in the latter case one observes the projection on the "back" of the plane. If *MXCOLR* is sufficiently large, the surface will be shaded relative to a light source directly behind the viewer, imparting some additional three-dimensional character to the picture.

The lower right-hand legend provides guidance in understanding three-dimensional surface plots. In this case the legend contains a "flat" version of the main picture, al-

lowing another avenue for orienting oneself with respect to the viewing perspective. Some examples of surface plots are given in Figures 5.1–5.4.

5.2.2 Vector Plots.

Color plays an important role in the vector plots. Different colors correspond to different directions in the vector field. This is illustrated in the color wheel portion of the upper right-hand legend. The number of directions is specified by the parameter *NCON*. Different intensities of the same color correspond to the magnitude of the vector; darker shades correspond to smaller magnitudes, and lighter shades correspond to higher magnitudes. The correspondence between color intensity and vector magnitude is illustrated for an example color in the upper right-hand legend. The parameters *SMIN* and *SMAX* are used to specify the limits of the color intensity scale for the magnitude of the vector. As with surface plots, if $SMIN < SMAX$, then these values are used as limits; otherwise the largest and smallest magnitudes of the vector function are used.

Three scales for the vector magnitude are available using the option switch *ISCALE*. Four line-drawing options using *LINES* and eight labeling options using *NUMBRS* are also available, and *RMAG*, *CENX*, and *CENY* provide zoom-in capabilities. The triple *(NX,NY,NZ)* specifies a direction as in the case of surface plots. In this case the surface plotted is the *linear interpolant* of the magnitude of the vector function.[4] In this case the elements remain colored as in a two dimensional vector plot. Some examples of vector plots are given in Figure 5.5. The vector function $S^m Q_h \nabla u_h$ can also be plotted as a scalar function $|S^m Q_h \nabla u_h|$. Some examples are given in Figure 5.6.

5.2.3 The Parameters RMAG, CENX, and CENY.

The parameters *RMAG*, *CENX*, and *CENY* provide a zoom-in option. *RMAG* is the magnification factor relative to the picture coordinates. For example, if $RMAG = 1$ the whole picture will be drawn; if $RMAG = 2$, the picture is scaled by a factor of 2 in both directions and thus no longer fits on the output device. One must now choose a window and view only a portion of the picture. The fractions $0 \le CENX \le 1$ and $0 \le CENY \le 1$ are used for this purpose. In particular $(CENX, CENY)$ specifies the point that will appear at the center of the magnified window. If $RMAG = 1$, the values of *CENX* and *CENY* are ignored. Some examples are shown in Figure 5.3 and Figure 5.6 (right).

As an aid to understanding, the lower right legend contains a copy of the complete picture (corresponding to $RMAG = 1$). Whenever $RMAG > 1$, a small box is drawn in this legend depicting the portion of the picture appearing in the main graph. The box is supplemented by a crosshair locator, since the box becomes too small to be visible for large magnification factors.

[4] For the actual magnitude, the surface of each triangular element is not necessarily a plane, making the hidden surface problem more difficult.

5.2.4 The Parameters ISCALE, LINES, and NUMBRS.

The parameter *ISCALE* provides three scaling options, summarized in Table 5.2. For linear scaling, drawn contours are equally spaced with respect to the largest and smallest values of the given function $z(x, y)$. If $ISCALE = 1$, then the contours are equally spaced with respect to the largest and smallest values of $\log z$. If $ISCALE = 2$, then the contours are equally spaced with respect to largest and smallest values of the function $\sinh^{-1} z$. The logarithmic scaling clearly requires z to be positive. The $\sinh^{-1}$ scaling is always defined, having a (signed) logarithmic behavior for large $\mid z \mid$ and a linear behavior for small $\mid z \mid$. If $ISCALE = 1$ and $z \leq 0$ at some vertex, then *TRIPLT* defaults to the $\sinh^{-1}$ scaling. In Figure 5.1, the solution u_h was drawn using the linear scale ($ISCALE = 0$), while the error estimate was drawn using the logarithmic scale ($ISCALE = 1$).

ISCALE	scale
0	linear
1	logarithmic
2	$\sinh^{-1}$
LINES	line drawing option
-2	matrix element boundaries
-1	skeleton graphics triangulation
0	all triangle edges
1	boundary/interface edges
2	load balance boundary edges
3	contours
NUMBRS	labeling option
-2	matrix element locations
-1	matrix element values
0	no labels
1	triangles/subregions
2	vertices
3	edges
4	curved edges
5	edge type
6	edge labels
7	processor
8	vertex type
ICRSN	coarsening option
0	no coarsening
1	coarsen global mesh

Table 5.2. *The values of ISCALE, LINES, NUMBRS and ICRSN.*

Three line drawing options are available, specified through the parameter *LINES*, as summarized in Table 5.2. If $LINES = 0$, *TRIPLT* will draw edges

of all triangles in the mesh. If $LINES = 1$, only boundary edges and edges separating triangles from different regions are drawn. The case $LINES = 2$ is similar to the case $LINES = 1$, except that here boundary edges and edges separating triangles from different processors are drawn. When $LINES = 3$ for surface plots, *TRIPLT* draws boundary triangle edges and contour lines separating contours of different colors. This option produces a traditional contour map on monochrome devices and thus is useful when $MXCOLR = 2$. Some examples for $LINES = 1$ and $LINES = 3$ are shown in Figure 5.4. The option *LINES=3* is not implemented for vector plots.

Eight labeling options are available in *TRIPLT*; these are specified through the parameter *NUMBRS*, as summarized in Table 5.2. When $NUMBRS \neq 0$, three-dimensional plotting is disabled; the result will be a "flat" (but labeled) surface. Some examples are shown in Figures 2.1 and 2.2.

5.2.5 The Parameters ICRSN and ITRGT.

When *NVF* becomes very large, the amount of data used to make an image may become too large for animated display systems like OpenGL or for Postscript files of reasonable size.[5] In this situation, one may wish to compress the data and make a lower resolution image. The parameter *ICRSN* indicates whether or not to coarsen the mesh, as indicated in Table 5.2. If $ICRSN = 1$, then the parameter *ITRGT* specifies the target number of vertices for the coarsened mesh. The coarsening option is very much like the mesh coarsening option in *TRIGEN*; many of the same subroutines are used, and the overall coarsening strategy is the same. However, the coarsening criteria is different.

In the case of *TRIPLT*, let p_i be a vertex in the mesh and Ω_i denote the patch of triangles having p_i as one its vertices. For each vertex, we compute the best (least squares) linear polynomial on Ω_i that interpolates the displayed function at p_i. A discrete ℓ^2 norm of the difference between this linear polynomial and the displayed function at the vertices lying on $\partial\Omega_i$ is used as the coarsening criterion. Such a criterion does not directly control the shape regularity or approximation properties of the mesh, but does tend to minimize the visual disruptions caused by deleting p_i and creating a triangulation of Ω_i based on its boundary vertices. All vertices are placed in a heap, and the least disruptive vertex is eliminated until the target *ITRGT* is achieved. When MPI is on ($MPISW = 1$), each processor independently coarsens the mesh for its subregion to a target of $ITRGT/NPROC$ vertices. Thus, when the submeshes are later combined, the global mesh appearing in the image will have at most *ITRGT* vertices.

When the mesh is coarsened, all numbering options are disabled; $NUMBRS = 0$ is always used. The setting $LINES = 0$ is reset to $LINES = 1$, and $ICONT = 1$ is always used.

[5] Raster graphics images like those produced by X-Windows displays and XPM files are largely independent of the size of the underlying data set.

5.2.6 Some Algorithmic Details.

The main algorithms of interest in *TRIPLT* are those for hidden line and surface removal. In the general case of a surface plot, one must make comparisons between various triangles to determine whether a given triangle blocks another with respect to the viewer. Since the triangular mesh is generally unstructured, our goal is to organize the data to minimize the number of comparisons between triangles.

Generally, for surface plots in which $(NX, NY, NZ) \neq (0, 0, 1)$, a partial order is constructed in which elements farthest from the viewer are ordered first, and those closest to the viewer are ordered last. The elements are then drawn and colored in order, with the elements closer to the viewer (possibly) overwriting some elements that are farther away. The notion of distance from the viewer is defined with respect to the x and y coordinates only, so that the same ordering is computed independently of the function being graphed. A typical element is compared only to elements with which it shares a common edge; it is ordered before any edge neighbors closer to the viewer and after any neighbors farther away. Since any element has at most three neighbors, this greatly limits the number of comparisons necessary and completely solves the ordering problem for a convex domain with no holes.

Unfortunately, many domains are not convex and have holes, so that elements with boundary edges must be treated as special cases. Thus we make a list of triangles with boundary edges, sort them with respect to the direction (in the (x, y) plane) perpendicular to the (NX, NY) components of the viewing direction. Boundary edges are also sorted by whether they face "backward" or "forward" with respect to (NX, NY). With these preliminary calculations done, all pairs of relevant triangles that *might* conflict are tested and appropriate ordering constraints imposed. For a mesh with NTF triangles, the number of boundary triangles is $O(\sqrt{NTF})$, so that in the worst case (every boundary element compared with every other boundary element), this will still be only $O(NTF)$ work. Since only $O(NTF)$ work is required for the interior elements, the overall work is still $O(NTF)$.

5.3 Subroutine INPLT.

Subroutine *INPLT* is a graphics routine for displaying the input data defining a triangulation or a skeleton. *INPLT* is called using the statement

Call INPLT(VX, VY, XM, YM, ITNODE, IBNDRY, IP, RP, SP, W)

The arrays *VX*, *VY*, *IBNDRY*, *ITNODE*, *XM*, and *YM* define either a triangulation or a skeleton (*INPLT* uses the value of *ITNODE(3,1)*, which is zero for a skeleton and positive for a triangulation, to distinguish these cases). The string variable *ITITLE* is displayed as a banner above the graph. Variables in the *IP*, *RP*, and *SP* arrays used by *INPLT* are shown in Tables 2.5–2.7. *INPLT* was used to make Figures 3.1 3.4, and 3.6, among others in this manual.

INPLSW	triangulation	skeleton
0	user label	user label
1	load balance	uniform color
2	element quality	subregion
3	largest angle	
4	smallest angle	
5	element size	
6	error estimate	

Table 5.3. *The values of INPLSW.*

5.3.1 Triangle Plots.

For triangle plots, the elements in the triangulation are colored to depict some feature of the mesh. The available options are controlled by the switch *INPLSW* as summarized in Table 5.3.

If $INPLSW = 0$, the elements in the mesh are colored according to the user supplied labels in *ITNODE(5,I)*; all elements with the same label will have the same color. If $INPLSW = 1$, the elements in the mesh are colored according to the load balance (*ITNODE(4,I)*).

For $2 \le INPLSW \le 4$, *INPLT* colors the elements of the triangulation according to their quality, measured by $q(t)$ in (3.1), their largest angle, and their smallest angle, respectively. For each of the three measures, five numbers are printed in the upper right legend. The row labeled "average" refers to the average of that quantity over all elements in the mesh; "worst" reports the smallest value of $q(t)$, largest angle, or smallest angle of all elements. The rows labeled "good," "fair," and "poor" report the percentage of elements in each category and depict the corresponding colors.

For $q(t)$, good means $q(t) \ge \sqrt{3}/2$, fair means $.6 \le q(t) < \sqrt{3}/2$, and poor means $q(t) < .6$. For large angles, good means $A(t) \le \pi/2$, fair means $\pi/2 < A(t) \le 2\pi/3$, and poor means $A(t) > 2\pi/3$ ($A(t)$ is the largest angle). For small angles, good means $\arccos(4/5) \le a(t)$, fair means $\arccos(13/14) \le a(t) < \arccos(4/5)$ and poor means $a(t) < \arccos(13/14)$ ($a(t)$ is the smallest angle). Triangles that are good in terms of $q(t)$ are (necessarily) also good in terms of large and small angles. Those that are fair in terms of $q(t)$ must be good or fair in terms of large and small angles (but not conversely).

When $INPLSW = 5$, *INPLT* produces an image in which each element is colored according to its size. A histogram showing the distribution of element sizes appears in the legend. Although any scaling option available through *ISCALE* can be used, generally the logarithmic scaling ($ISCALE = 1$) produces the most useful image.

When $INPLSW = 6$, *INPLT* produces an image in which each element is colored according to its error $\|\epsilon_t\|_{\mathcal{H}^1(t)}$. This is similar to the case of *TRIPLT* with *IFUN=5* and $ICONT = 0$, except that the *INPLT* image is strictly two dimensional.

This option is included because the coarsening procedure used in *INPLT* with *ICRSN* = 1 assumes discontinuous element data, and generally produces higher quality reduced resolution images than *TRIPLT* for this highly oscillatory function. As in the case *INPLSW* = 5, all scaling options are available, but *ISCALE* = 1 is typically most useful. Some example images made using *INPLT* are shown in Figures 5.7 and 5.8.

The meanings and use of *RMAG*, *CENX*, *CENY*, and *MXCOLR* are identical to *TRIPLT*. Labeling options using *NUMBRS* are summarized in Table 5.2. *INPLT* was used with various *NUMBRS* options to produce Figure 2.1 although the legends on the right-hand sides of the pictures were deleted. For the main graph, three line-drawing options are available using *LINES*, as summarized in Table 5.2.

Subroutine *INPLT* also allows mesh coarsening, but the criterion is different. In *INPLT*, each element is a single color and the images are two dimensional, and the coarsening criterion reflects these differences. If all elements in Ω_i are the same color, then p_i is eliminated. In the case *INPLSW* = 5, 6, if this initial coarsening does not produce a mesh with fewer than *ITRGT* vertices, the elements are scanned, and all elements with two or more neighbors of the same color (different from their color) are switched to that color. This has the effect of smoothing the boundary between regions of different colors. The coarsening process is then applied to the relabeled mesh. This process is repeated as necessary until the target value is exceeded. As with *TRIPLT*, *NUMBRS* = 0 is always specified for a coarsened mesh and *LINES* = 0 is reset to *LINES* = 1.

5.3.2 Skeleton Plots.

As with triangle plots, the subregions of the skeleton are colored according to the option specified by *INPLSW* as summarized in Table 5.3. If *INPLSW* = 0, the subregions are colored according to the user supplied labels in *ITNODE(5,I)*, similar to the case of a triangulation. If *INPLSW* = 1, each subregion is given the same color, while if *INPLSW* = 2, each subregion is given a different color.

Subroutine *INPLT* draws a skeleton by first creating a crude triangulation based on the skeleton, and then drawing the triangulation. Here shape regularity and overall quality of the triangulation is not an issue; rather, keeping the number of elements small and computing the triangulation quickly are important. The option *LINES* = −1 displays the underlying triangulation used in the skeleton plot. It was included mainly for debugging purposes.

The parameters *RMAG*, *CENX*, *CENY*, and *MXCOLR* are the same as for triangle plots. Labeling options using *NUMBRS* are summarized in Table 5.2. There are no coarsening or parallel computation options available for skeleton plots. *INPLT* was used with various *NUMBRS* options to produce Figure 2.2.

5.4 Subroutine GPHPLT.

Subroutine *GPHPLT* displays an assortment of data related to the performance of various algorithms and subroutines in *PLTMG* and *TRIGEN* using a graphical format.

GPHPLT is called using the statement

Call GPHPLT(IP, RP, SP, W)

GPHPLT makes use of the arrays *PATH*, *HIST*, *PSTAT*. *KA*, and *TIME*, initialized by *PLTMG* and *TRIGEN* when $FIRST = 1$ and containing data generated during the solution process. The string variable *GTITLE* is displayed as a banner above the graph. Other variables in the *IP*, *RP*, and *SP* arrays used by *GPHPLT* are shown in Tables 2.5–2.7.

IGRSW	displayed graph
0	Newton iteration convergence history
1	multigraph iteration convergence history
-1	matrix sizes in multigraph iteration
2	individual subroutine timing statistics
-2	time pie chart
3	the continuation path
-3	load balance
4	error estimates for $\mathcal{H}^1$ norm
-4	error estimates for $\mathcal{L}^2$ norm
5	the *IP* array
-5	the *SP* array
6	the *RP* array
-6	the *KA* array

Table 5.4. *The values of IGRSW.*

IGRSW is an integer switch for selecting the displayed graph; the available possibilities are summarized in Table 5.4.

5.4.1 Iteration Information.

For the cases $IGRSW = -1, 0, 1$, information about various iterations is displayed. In all three cases, the same three graphs are drawn. The large main window contains the information indicated in Table 5.4 for the corresponding value of *IGRSW*. The other two graphs appear in the two smaller frames on the right. Examples are shown in Figures 5.9– 5.10.

In the case $IGRSW = 0$, in the main window *GPHPLT* graphs the functions

$$\mathcal{R}_k = \log_{10}\left\{\frac{\|\mathcal{G}_k\|}{\|\mathcal{G}_0\|}\right\} \qquad \text{and} \qquad \mathcal{E}_k = \log_{10}\left\{\frac{\|\delta\mathcal{U}_k\|}{\|\mathcal{U}_k\|}\right\}.$$

$\mathcal{G}_k$ is the residual for the Newton iteration, while δS_k is the incremental change in the solution S_k. The precise meaning of $\mathcal{G}_k$ and S_k varies according to the system of nonlinear equations solved for each problem class addressed by *PLTMG*. Both

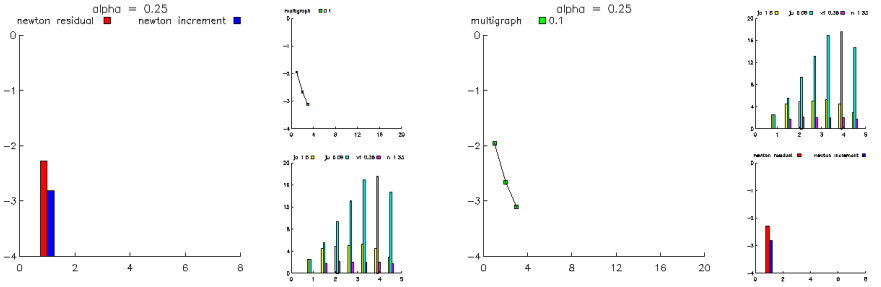


Figure 5.9. *The cases* $IGRSW = 0$ *and* $IGRSW = 1$.

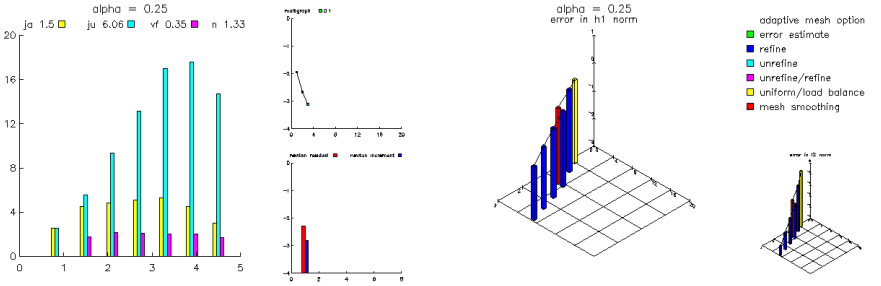


Figure 5.10. *The cases* $IGRSW = -1$ *and* $IGRSW = 4$.

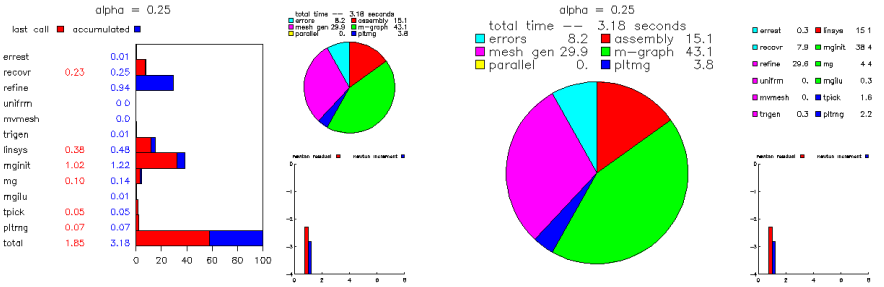


Figure 5.11. *The cases* $IGRSW = 2$ *and* $IGRSW = -2$.

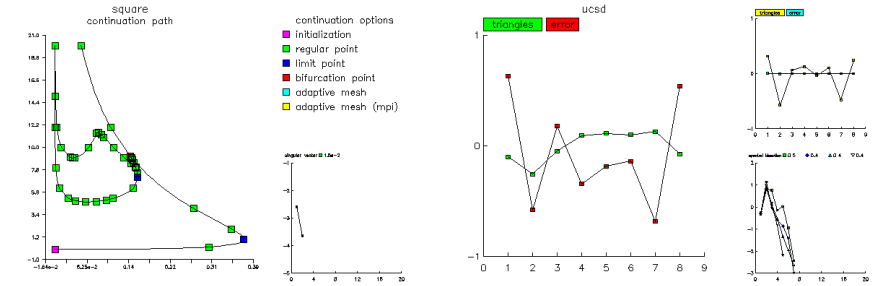


Figure 5.12. *The cases* $IGRSW = 3$ *and* $IGRSW = -3$.

convergence histories are plotted in a bar graph of $\mathcal{R}_k$ and $\mathcal{E}_k$ versus iteration index k.

The convergence history for the most recently solved set of equations is displayed. When this corresponds to a regular (serial) solution ($IPROB > 0$), the relative residuals are red bars, while the solution increments are blue. At most, information about the last twenty Newton iterations is displayed. When $IPROB < 0$, the Newton iteration employs the parallel domain decomposition/multigraph solver in place of the simple multigraph solver. In this case, the residuals are magenta bars, and the solution increments are cyan.

Nominally, the rate of convergence for Newton's method should asymptotically be quadratic; however, the convergence becomes linear when systems of linear equations involving the Jacobian matrix are only approximately solved.

In the case $IGRSW = 1$, in the main window *GPHPLT* graphs the function

$$\mathcal{S}(k) = \log_{10} \left\{ \frac{\|r_k\|}{\|r_0\|} \right\}.$$

Here r_k is the residual of a set of linear equations to be solved by the multigraph method and k is the iteration number. The displayed histories are for linear systems solved in the most recent Newton iteration. Up to four such systems are solved in each Newton step, depending on the value of *IPROB*. In all cases, only information about the last twenty cycles of the most recent iteration is saved and displayed.

Either the composite step conjugate gradient method or composite step biconjugate gradient method is used [7, 6], preconditioned by a multigraph incomplete factorization [21]. Each individual step is marked with a small icon; a color pair (green, red), (blue, yellow), (cyan, magenta), (white, black) is assigned to each history. In each case, for simple steps the icon is colored with the first color (e.g., green), while for composite steps the icon is colored with the second color (e.g., red).

In the case $IGRSW = -1$, in the main window statistics related to the multigraph method are displayed. The horizontal axis is $\log_{10} N$ where N is the order of the matrix. For each level (N_k) three bars are displayed; the heights are proportional to the average number of nonzeros per row in the upper triangular part of the system matrix (yellow), the *ILU* factorization (cyan), and the transfer matrix to the next coarser level (magenta). More detailed information about matrix sizes is available by setting $IGRSW = -6$.

5.4.2 Timing Statistics.

If $IGRSW = 2$, *GPHPLT* prints a summary of timing statistics for *PLTMG* and *TRIGEN*. An example is given in Figure 5.11. Statistics are given both for the total accumulated time since initialization ($IFIRST = 1$) and for the time spent during the last call to *PLTMG* or *TRIGEN*. The timings are itemized with respect to subroutines that carry out major computational tasks in the package. These subroutines are listed in Table 5.5. Depending on the problem, some of these routines may not be called.

subroutine	main function
TGEN	create triangulation from skeleton
REFINE	adaptively refine the triangulation
UNREFN	adaptively unrefine the triangulation
UNIFRM	uniformly refine the triangulation
MVMESH	adaptively smooth the mesh points
ERREST	compute error estimates for u_h
CDLFN	Compute dual function
RGEN	create skeleton from triangulation
LDBAL	compute a load balance
LDEV	solve eigenvalue subproblem in load balance
CUTR	reorganize data structures for reconciling mesh
PASTE	reconcile mesh along interface of *IRGN*
PASTE1	reconcile mesh along interface not part of *IRGN*
BCAST	broadcast mesh to all processors
COLLCT	gather mesh from all processors
EXPTH	exchange boundary interface data
TRIGEN	all other time spent in *TRIGEN*
MGINIT	initialize multigraph data structures
MG	solve equations using multigraph iteration
MGILU	compute sparse *ILU* factorizations
CEV	compute the singular value μ and vectors ψ_r and ψ_ℓ
LINSYS	compute the stiffness matrix and right-hand side
PREDCT	compute the steplength σ for continuation
BLK3	block elimination for $IPROB = 3$
BLK4	block elimination for $IPROB = 4$
BLK5	block elimination for $IPROB = 5$ and $IPROB = -5$
SWBRCH	switch branches at a bifurcation point
TPICK	line search for Newton iteration
BLK3DD	block elimination for $IPROB = -3$
RGNSYS	assemble linear system for domain decomposition
TPICKD	line search for Newton/DD iteration
BLK4DD	block elimination for $IPROB = -4$
RECOVR	Compute $S^m Q_h \nabla u_h$
PLTMG	all other time spent in *PLTMG*

Table 5.5. *Subroutines timed by GPHPLT.*

A bar graph is drawn illustrating the percentage of time spent in each routine. Each bar in the graph is partitioned into a part corresponding to the last call to *PLTMG* (red) and a part corresponding to all preceding calls (blue). The timing pie graph described below appears in the upper right frame.

If $IGRSW = -2$, *GPHPLT* displays a pie graph summarizing the same information. Each routine in Table 5.5 is assigned to one of six categories: linear system

assembly (red), multigraph solver (green), mesh generation (magenta), a posteriori error estimation (cyan), parallel processing routines (yellow), and other *PLTMG* routines (blue). A pie graph showing the fraction of total time spent in each of the six categories is drawn in the main frame. Details of individual contributions from the subroutines listed in Table 5.5 are summarized in the upper right frame. Sample output is shown in Figure 5.11.

When $MPISW = 1$, the times displayed for $IGRSW = \pm 2$ are time averaged across all processors. In this case, in the lower right frame, a graph displaying the deviation from the average time for each processor is drawn.

5.4.3 Continuation Path.

When $IGRSW = 3$, *GPHPLT* displays the continuation path generated by the continuation procedure $IPROB = 3$. Target points are marked by small boxes, generally using different colors for different values of *ITASK*. A legend appears in the upper right frame summarizing the possibilities. Up to one hundred target points generated by calls to *PLTMG* are saved and displayed. Successive points are interpolated using parabolic arcs matching the values of (λ, ρ) and the tangent vectors $(\dot\lambda, \dot\rho)$. In the lower right frame appears a convergence history for the most recent singular vector computation. Sample output is shown in Figure 5.12.

5.4.4 Parallel Statistics

When $IGRSW = -3$, *GPHPLT* plots the functions

$$T_k = \log_2 \left\{ \frac{NPROC \cdot NTF(\Omega_k)}{\sum_k NTF(\Omega_k)} \right\} \quad \text{and} \quad E_k = \log_2 \left\{ \frac{NPROC \|\epsilon_t\|_{\mathcal{H}^1(\Omega_k)}}{\sum_k \|\epsilon_t\|_{\mathcal{H}^1(\Omega_k)}} \right\}$$

where $1 \le k \le NPROC$. Both curves appear in the large frame. When $MPISW = 1$ the information from all processors is obtained by an exchange of data using the MPI library. This is the most useful situation. When $MPISW = -1$, the same graph is made using local data on the given processor; this case is typically not interesting. In the upper right frame is a similar graph for the distribution of error and elements following the initial load balancing step ($IADAPT = 7$). In the lower right frame appear convergence histories for eigenvalue computations in the load balancing phase. Convergence histories are shown for the four most recent problems. Sample output is shown in Figure 5.12.

5.4.5 Error Estimates.

In the case $IGRSW = 4$, *GPHPLT* graphs the function

$$\mathcal{F}_1(NVF, ICALL) = \log_{10} \left\{ \frac{\|\epsilon_t\|_{\mathcal{H}^1(\Omega)}}{\|u_h\|_{\mathcal{H}^1(\Omega)}} \right\},$$

and in the case $IGRSW = -4$, *GPHPLT* graphs the function

$$\mathcal{F}_0(NVF, ICALL) = \log_{10} \left\{ \frac{\|\epsilon_t\|_{\mathcal{L}^2(\Omega)}}{\|u_h\|_{\mathcal{L}^2(\Omega)}} \right\}.$$

Here ϵ_t is the computed approximation of the error $u - u_h$. While it is hoped that these approximations accurately reflect the true state of affairs, the estimates are based on a posteriori calculations involving only the computed solution. Some judgment of the validity of such computations may be required. An example is shown in Figure 5.10.

Since there are several adaptive options in *TRIGEN* that do not involve a change in *NVF*, error estimates are plotted as a function of both *NVF* and *ICALL*. In particular, $\mathcal{F}_j$ is graphed versus $\log_{10} NVF$ and *ICALL* in a three-dimensional graph. All data points (up to the 20 most recent) for which error estimates are available are marked with rectangular cylinders of different colors. A legend appears in the upper right frame summarizing the possibilities. In the case $IGRSW = 4$, the plot of $\mathcal{F}_0$ appears in the lower right frame; if $IGRSW = -4$, the plot of $\mathcal{F}_1$ in the lower right frame.

The triple $d = (MX, MY, MZ)$ specifies the viewing perspective for these graphs in a fashion similar to *(NX,NY,NZ)* for surface plots. The choice $(1, 1, 1)$ is a reasonable default. The choice $(0, -1, 0)$ yields a traditional two-dimensional graph of $\log_{10} \mathcal{F}_j$ versus $\log_{10} NVF$, and is useful for situations where only refinement options are used in *TRIGEN*. The choice $(1, 0, 0)$ yields a two-dimensional graph of $\log_{10} \mathcal{F}_j$ versus *ICALL* and is useful when only mesh smoothing options are employed. The main image can be animated using three dimensional imaging systems such as OpenGL.

5.4.6 Displaying Data Arrays.

The options $| IGRSW | \geq 5$, *GPHPLT* displays the *IP*, *RP*, *SP*, or *KA* arrays. Unlike other graphics options, here the entire graphics window is treated as a single frame. In the case of the *IP* and *RP* arrays, all 100 entries, their names, and their current values are displayed. Entries that can be interactively reset in the *ATEST* driver are colored red, unused entries appear in black, and all other entries are colored blue. This situation is similar for the *SP* array, except only the first 50 entries are displayed (the remainder are all presently unused). When $IGRSW = -6$, information about the multigraph solver stored in the *KA* array is displayed. This includes the order and number of nonzeros in the system matrix and *ILU* factorization for all of the levels, as well as the sizes of the inter-level transfer matrices. Examples are shown in Figures 5.13–5.14.

5.5 Subroutine MTXPLT.

Subroutine *MTXPLT* displays the sparsity structure of the stiffness matrix A, the LDU factors from the ILU factorization, or the error matrix $E = LDU - A$. *MTXPLT* is called using the statement

Call MTXPLT(JA, A, IP, RP, SP, VX, VY, W)

The arrays *JA* and *A* should contain the matrices generated through an appropriate call to *PLTMG*. *MTXPLT* uses several variables from the *IP*, *RP*, and

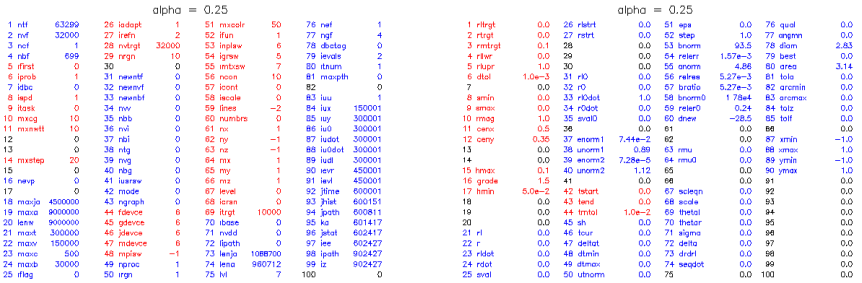


Figure 5.13. *The cases IGRSW* = 5 *and IGRSW* = 6.

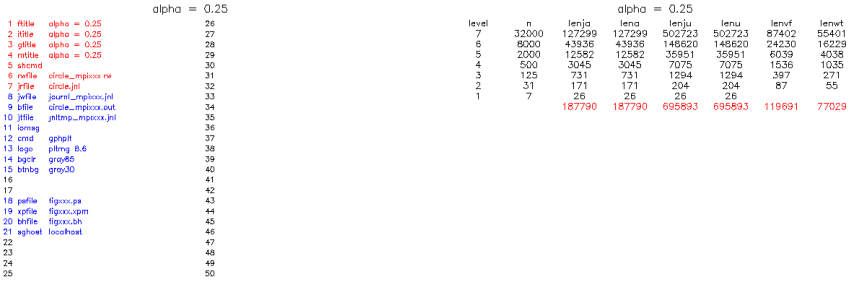


Figure 5.14. *The cases IGRSW* = −5 *and IGRSW* = −6.

SP arrays, as shown in Tables 2.5–2.7. The *VX* and *VY* arrays are used for options that display matrix graphs. The string variable *MTITLE* is the character string displayed as a label above the graph. The error flag *IFLAG* is set as in Table 2.8.

The parameter *IMTXSW* specifies the matrix to be plotted. The available options are summarized in Table 5.6. The parameter *LEVEL* indicates the multigraph level of the matrix to be drawn. If *LEVEL* < 1 or *LEVEL* > *LVL* then the matrix from the finest level *LVL* is displayed.

IMTXSW	displayed matrix
±1	*LDU* colored by element type
±2	*LDU* colored by element size
±3	*A* colored by element type
±4	*A* colored by element size
±5	*E* colored by element type
±6	*E* colored by element size
±7	matrix graph, colored by level

Table 5.6. *The values of IMTXSW.*

The main picture in divided into an $N \times N$ square grid. Grid cell (i, j) corresponds to matrix element (i, j). For the case of *LDU* factorizations, the strictly lower triangular part of L, the diagonal D, and the strictly upper tri-

angular part of U are displayed (L and U have unit diagonal entries). Matrix elements stored in sparse matrix data structures are colored according to type or size. If $IMTXSW > 0$, then matrix element magnitudes are displayed. If $IMTXSW < 0$, then (signed) matrix element values are displayed. For the cases $\mid IMTXSW \mid = 2, 4, 6$, the parameters *NCON* and *ISCALE* are used to determine the color scale in a fashion similar to *TRIPLT*. In the case $IMTXSW = 7$, the graph of the matrix is drawn, with vertices of the graph colored by level. When $IMTXSW = -7$, just the vertices in the matrix graph are drawn. Some examples are shown in Figures 5.15 and 5.16. The parameters (MX, MY, MZ) can be used to set a viewing perspective in a fashion similar to *GPHPLT*. In perspective views, matrix elements are displayed as rectangular cylinders with height proportional to element value or magnitude. *LINES* and *NUMBRS* can be set as indicated in Table 5.2. Similar to *TRIPLT*, if $NUMBRS \neq 0$ and $(MX, MY, MZ) \neq (0, 0, 1)$, the picture will be drawn on a "flat" surface. The parameters *RMAG*, *CENX*, and *CENY* may be used as in Section 5.2.3 to provide zoom-in capabilities. The main image can be animated using three dimensional imaging systems such as OpenGL. Some examples are shown in Figures 5.17 and 5.18.

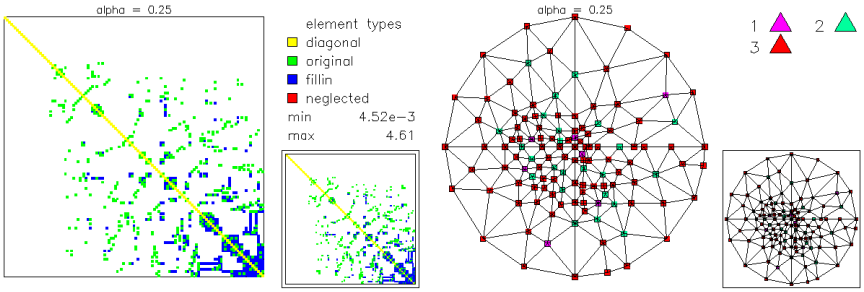


Figure 5.15. *The case* $IMTXSW = 1$ *(left) and* $IMTXSW = 7$ *(right).*

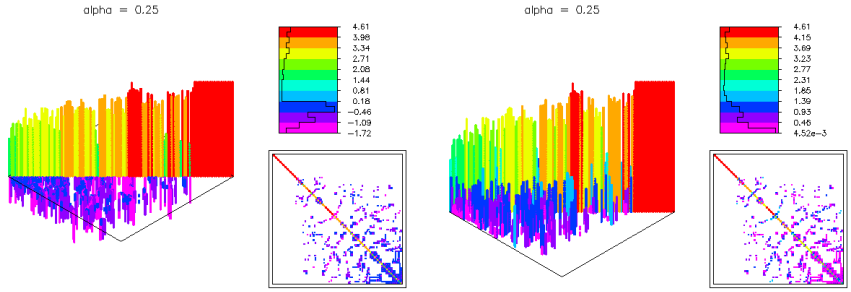


Figure 5.16. *The cases* $IMTXSW = -2$ *(left) and* $IMTXSW = 2$ *(right).* $(MX, MY, MZ) = (1, 1, 1)$ *in both images.*

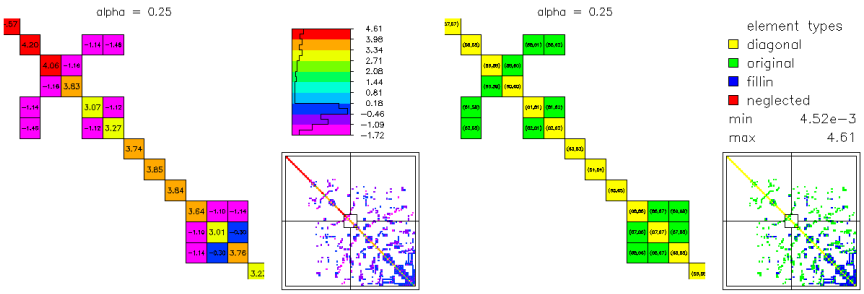


Figure 5.17. $RMAG = 10$, $NUMBRS = -1$ *(left) and* $NUMBRS = -2$ *(right).*

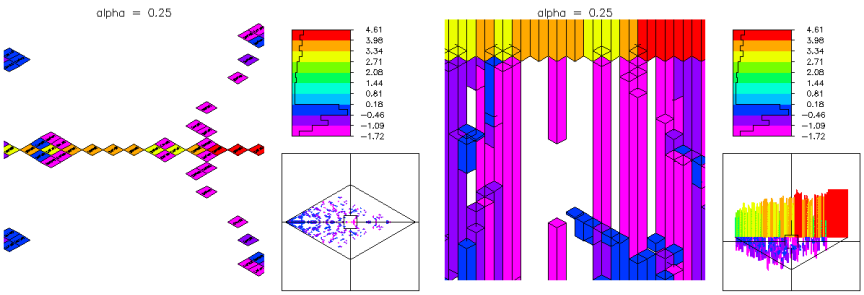


Figure 5.18. $RMAG = 10$, $(MX, MY, MZ) = (1, 1, 1)$. $NUMBRS = -2$ *(left) and* $NUMBRS = 0$ *(right).*

Chapter 6

Test Driver

6.1 Overview.

Program *ATEST* is the test driver used in the development and testing of the *PLTMG* package. *ATEST* is a flexible program in that it accepts simple command strings directing it to call subroutines or perform other tasks. It is not limited to a fixed sequence of tasks on a particular run; any routine can be called as often as desired, with certain parameters reset for each call at the discretion of the user.

The program *ATEST* can operate in four modes, governed by the switch *MODE*. If $MODE = -1$, *ATEST* runs as an interactive program, accepting commands from the user via a terminal window. If $MODE = 1$, *ATEST* runs interactively, accepting commands from the user via an X-Windows interface. This interface is based on the Motif widget set and can be used only in environments supporting X-Windows. If $MODE = 1$, *ATEST* runs as a batch program, reading commands from a journal file and sending all output to appropriate output files. Finally, if $MODE = -2$, *ATEST* runs as a slave mode under MPI; this mode cannot be directly set by the user, but is set by *ATEST* if it determines that it is a slave node in a parallel computation. In this situation, the user specifies *MODE* only for the master node, which can be any of the three other options.

A common command syntax is used for all modes. This is described first for the case $MODE = -1$ in Section 6.2. The extensions used in the X-Windows interface are described in Section 6.3.

Several files are written by *ATEST*. The file *BFILE* contains a complete record of all commands and printed output produced during the session. The file *JWFILE* contains a record of all commands read and processed during the session, formatted as a journal file. See Section 6.8 for a discussion of journal files. *ATEST* also creates a temporary file *JTFILE* used in connection with the journal command. While most commands invoke one of the major routines in the package, there are a few utility routines (e.g. for reading and writing files) which are documented in Sections 6.7–6.10.

6.2 Terminal Mode.

In terminal mode, commands are entered from a terminal window in character strings of 80 characters, counting blanks. The syntax of a command can take several forms, but the root command is always a single letter. The commands that are currently recognized by *ATEST* are summarized in Table 6.1.

Command	Action
s	call *PLTMG*
t	call *TRIGEN*
f	call *TRIPLT*
g	call *GPHPLT*
i	call *INPLT*
m	call *MTXPLT*
w	write data set to a file
r	read data set from a file
j	read journal file
k	execute shell command
u	call *USRCMD*
p	MPI toggle
q	quit

Table 6.1. *Available commands for ATEST.*

The terminal window prompt is the string *command:*. At this prompt, one can enter a command string (e.g., s), reset parameters as described below, or enter a blank line to see a list of the available commands. In this latter case the terminal window will appear as follows.

```
command:
trigen t    pltmg  s    triplt f    gphplt g    inplt  i    mtxplt m
read   r    write  w    usrcmd u    journl j    shell  k    mpi    p
quit   q

command:
```

A syntax error in a given command string causes the entire string to be ignored. *ATEST* will display the string *command error* and present the command prompt for a new input string.

The most simple commands are just single lower case letters as shown in Table 6.1. However, associated with most commands are various parameters which can be reset before calling the given routine. To see a listing of the parameters associated with a given command and their current values, without executing the command itself, enter the command in upper case at the command prompt. For example, the command F will display the parameters which can be interactively reset in connection with *TRIPLT*.

```
command:F
ifun   f  0         iscale s  0         lines  l  0         numbrs n  0
fdevce d  0         nx     nx 0         ny     ny 0         nz     nz 1
ncon   c  11        icont  ic 0         icrsn  cr 0         itrgt  it 10000
mxcolr mc 100       smin   sn 0.0       smax   sx 0.0       rmag   m  1.0
cenx   cx 0.5       ceny   cy 0.5
ftitle t  "circle"

command:
```

There are thirteen integer parameters, five real parameters, and one string parameter affecting subroutine *TRIPLT* that can be interactively reset by the user. To the right of each parameter is a one- or two-letter alias (to avoid typing long names), followed by the current value.

To reset some parameters associated with a command c (c = *s, f, g,* etc.), without invoking the command itself, one can type a string of the form

```
command:C name1=value1, name2=value2, ... , namek=valuek
```

Note that the root command appears in upper case. The *namek* refer to variable names or their aliases, and *valuek* refer to integer, real, or string values. Several parameters can be reset, with different entries separated by commas. Values for integer parameters should be integers, while values for real parameters can be specified using integer, fixed point, or exponential notation. There are three types of string parameters: *short*, *long* and *file*. Short strings are typically single words and can not contain any blank characters. Files are typically file names, and they also can not contain any blank characters. All other strings are long, and can contain any printable ASCII characters other than double quotes. Values of long string parameters should appear within double quotes. Short and file string parameters are not enclosed with double quotes. Blank spaces are ignored everywhere but within the value field of a long string parameter. A syntax error in the input line (e.g., a misspelled variable name) causes the entire command to be ignored and no variables to be reset. *ATEST* will respond *command error* and then ask for the next command. For example, here we reset $ISCALE = 1$, $NCON = 20$, $CENX = .3$, $RMAG = 10$, and *FTITLE* = *A new title for circle*. Subroutine *TRIPLT* is not called, but the parameters are updated and redisplayed as

```
command:F s=1, ncon=20, cenx=.3, rmag=1.e1, t="A new title for circle"
ifun   f  0         iscale s  1         lines  l  0         numbrs n  0
fdevce d  0         nx     nx 0         ny     ny 0         nz     nz 1
ncon   c  20        icont  ic 0         icrsn  cr 0         itrgt  it 10000
mxcolr mc 100       smin   sn 0.0       smax   sx 0.0       rmag   m  10.0
cenx   cx 0.3       ceny   cy 0.5
ftitle t  "A new title for circle"

command:
```

One can reset some parameters for a given command c, and then invoke the command itself, using a string of the form

```
command:c name1=value1, name2=value2, ... , namek=valuek
```

Note that the only difference is that the root command now appears in lower case rather than upper case. Thus

```
command:f s=1, ncon=20, cenx=.3, rmag=1.e1, t="A new title for circle"
```

resets the indicated parameters as in the previous example. However, instead of displaying the updated values, subroutine *TRIPLT* is called.

Finally, the graphics and MPI commands (f, i, g, m and p) have a short form allowing one crucial parameter (*IFUN*, *INPLSW*, *IGRSW*, *IMTXSW*, and *MPISW*, respectively) to be reset without typing even the alias. For example,

```
command:f5
```

is the short form for

```
command:f ifun=5
```

The short and long forms of these commands cannot be mixed. Thus

```
command:f5, ncon=10
```

is not valid.

6.3 X-Windows Mode.

When $MODE = 0$, the driver *ATEST* creates an X-Windows interface for the *PLTMG* package. The functional capabilities are the same as for the terminal window mode, but the possibilities for data entry are more varied. An example of the X-Windows interface appears in Figure 6.1.

The main display contains three elements. The upper portion of the display contains *command buttons*. Below the command buttons is a one line *command window*. The bottom portion of the display is the *history window*. The interface supports up to ten graphics displays.

The command buttons stand in one to one correspondence with the basic *ATEST* command set shown in Table 6.1. In particular, clicking the left mouse button (button one) with the pointer over a command button is equivalent to the typed lower-case version of that command. For example, clicking mouse button one on the *TRIPLT* command button causes subroutine *TRIPLT* to be called as in the command f. On the other hand, clicking on the right mouse button (button three) with the pointer over a command button is equivalent to the upper case version of the command. Clicking mouse button three on the *TRIPLT* command button causes the parameters for the *TRIPLT* command to be displayed in a popup reset window, as in the typed command F. This is shown is figure 6.2.

The parameters associated with a given command are displayed in the reset window in a format similar to terminal mode. However, each parameter value is

Figure 6.1. *The X-Windows interface.*

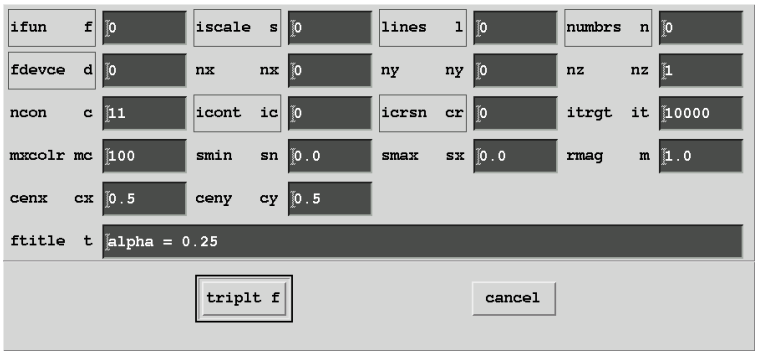


Figure 6.2. *An example reset window.*

displayed in one line text-editing window, and can be reset by typing in the new value. For some parameter names (e.g., *IFUN* in Figure 6.2), the name has a colored border. Clicking on the name causes a display of radio buttons, listing available options for the given parameter, to pop up. Clicking on the appropriate option causes the parameter to be reset to the corresponding value. The radio button popup associated with the parameter *IFUN* appears in Figure 6.3.

For file selection commands (*READ*, *WRITE*, and *JOURNL*), the generic reset window is replaced by the Motif file-selection widget. The file-selection popup

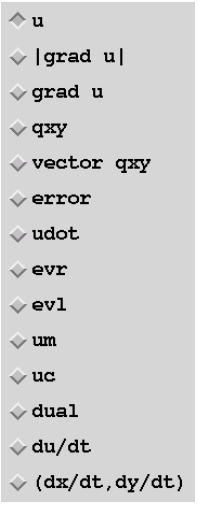


Figure 6.3. *An example radio buttons popup.*

for the *JOURNAL* command is shown in Figure 6.4.

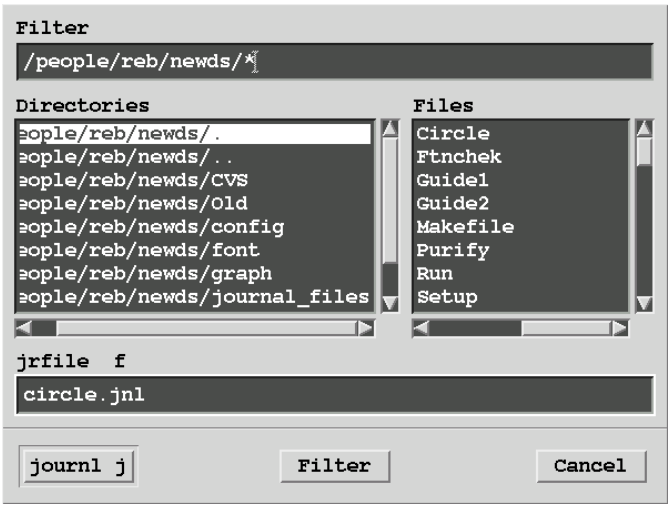


Figure 6.4. *An example file selection popup.*

The X-Windows interface also allows commands to be entered from the keyboard as in terminal mode. In particular, one can place the pointer in the command window, and enter a command from the keyboard as in terminal mode.

The history window displays the contents of the output file, *BFILE*, as it is created. If the file becomes sufficiently large, only the tail of the file is displayed.

The X-Windows driver also supports ten graphics displays (numbered 0-9).

The parameter *NGRAPH*, $1 \leq NGRAPH \leq 10$, states the number of windows to create initially. Graphics displays can be dismissed and recreated as necessary. These windows use only X-Windows primitives, and display static images which cannot be manipulated (e.g. rotated) with the mouse. Graphics popups can be resized in the usual way, but maintain a 3/2 aspect ratio. Also, any existing image is erased upon resize, and must be redrawn.

When executing a journal file in X-Windows mode, if a graphics command is executed, depending on the graphics device selected, *ATEST* can pause after the picture is drawn, and create a small popup *continue* button. In this case, *ATEST* waits until the user dismisses the *continue* popup before continuing to execute the journal file. This allows time for the user to view the picture before processing the next command in the journal file.

The X-Windows display can be interactively resized in the usual way. However, *ATEST* will adjust the user-specified resizing such that an overall aspect ratio of 3/2 is maintained. *ATEST* also imposes a minimum size requirement on the main window.

The string parameters *BGCLR* and *BTNBG* allow the user to specify the background and button background colors for the main display. Motif automatically defines the remaining colors used in the display. These parameters can be given any of the named colors supported by X-Windows. The string parameter *LOGO* is provided to X-Windows for use as titlebars and other identifiers.

Finally, we remark that the X-Windows interface does not follow the pattern of many X-Windows programs, in that the *PLTMG* package was not integrated into the X-Windows system with the X-Windows interface serving as the main routine. Indeed, the X-Windows interface is realized as a collection of C language subroutines called by a Fortran driver. These routines use the same database of Fortran character strings as the terminal window interface to define their displays, and return command strings of the same type described in the terminal windows interface. Both the X-Windows interface and the terminal window interface are quite generic, in that neither contains direct links to any of the main routines in the package. Thus changes in the behavior of routines comprising the package have no impact on the interface routines and at most modest impact on the database of character strings that define the displays.

6.4 Batch Mode.

When $MODE = 1$, the *ATEST* driver runs as a batch program. All commands are read from the journal file specified in *JRFILE*. Graphics output should be directed to files (BH, Postscript, and XPM) rather than to interactive displays.

6.5 Parallel Processing

When run as a parallel program using *NPROC* processors, *ATEST* uses a master-slave model. One process, the master process, runs in terminal, X-Windows, or batch mode, and the remaining $NPROC - 1$ slave processes all run with $MODE =$

-2. Slave nodes receive command strings from the master node via MPI communication. At any given time, the parallel computation is in one of two possible states that specify how slave nodes should process commands. Somewhat arbitrarily, the two states are denoted "off" and "on". When MPI is on, all processors execute all commands from the user, whether entered interactively or through a journal file. When MPI is off, only the master process executes most commands. Slave nodes remain active and still receive and evaluate the command strings they receive. Some commands (namely p and q) continue to be executed and some parameter updates continue on all processors in the off state. However, in the off state, slave nodes are mainly waiting for MPI to be turned on again.

The p command is used to switch between the on and off states of MPI. When $MPISW = 1$, MPI is on, and when $MPISW = -1$, MPI is off. The p command is unusual in that it can behave as a toggle; executing p with no argument switches the MPI state. The p command can also be employed in the usual way to explicitly set the MPI state using the parameter *MPISW* (e.g., $p1$ turns on MPI, while $p-1$ turns off MPI). When the master process is running in X-Windows mode, a small icon appears to the right of the command window when MPI is turned on, and disappears when MPI is turned off.

A common and effective way to use MPI is to create a journal file that contains a script for the entire computation (including p commands) The j command issued in the MPI on state directs all processors to run the journal file. The master process will then execute the entire script, while the slave nodes execute the parts of the journal file that correspond to the on state.

An issue with respect to file names arises in the context of parallel processing. Some files, for example a journal file, are intended to be read by all processors. In other situations, for example writing data files, each processor is intended to process its own version of the file. Then name conflicts can potentially become catastrophic if all nodes read and write files on the same file system. To resolve this conflict in a simple way that allows the user to easily specify on a case-by-case basis if the file is a single file or a file with distinct copies on each node, *ATEST* scans all file names, looking for the characteristic string *MPIXXX*. If found, this string is replaced by *MPI001*, *MPI002*, etc, where the integer part denotes the processor. Thus, for example if one sets

JRFILE = MYFILE.JNL

all nodes process the same file with the name *MYFILE.JNL*. If one sets

RWFILE = MYFILE_MPIXXX.RW

node one would process the file *MYFILE_MPI001.RW*, node two would process the file *MYFILE_MPI002.RW*, and so on.

6.6 Array Dimensions and Initialization.

ATEST has six labeled *common* blocks:

```
common /atest1/ip(100),rp(100),sp(100)
common /atest2/iu(100),ru(100),su(100)
common /atest3/mode,jnlsw,jnlr,jnlw,ibatch
common /atest4/jcmd,cmdtyp,list
common /atest5/idevce
common /atest6/nproc,myid,mpisw
```

The *IP*, *RP*, and *SP* arrays are described in Section 2.4. The arrays *IU*, *RU*, and *SU* are not directly used by *ATEST* or any of the other routines. They are provided to the user for storing integer, real, and string parameters associated with a particular problem. The advantages in using these arrays are that they are saved and read in the *w* and *r* commands; the common block *ATEST2* can be included in subroutines *A1XY*, *A2XY*, etc., where the parameters may be needed; and they can form part of the interface for resetting problem parameters using *USRCMD*. *ATEST3* contains internal control parameters used by *ATEST*; each has a corresponding location in the *IP* array. *ATEST4* contains string and integer variables that are used for internal communication among the user interface routines. The block *ATEST5* contains an integer specifying the current graphics output device, while *ATEST6* contains parameters relevant to MPI.

The input data arrays *ITNODE(5,MAXT)*, *IBNDRY(6,MAXB)*, *VX(MAXV)*, *VY(MAXV)*, *XM(MAXC)*, *YM(MAXC)*, *JA(MAXJA)*, *A(MAXA)*, and the work array $W(LENW)$ are declared at the beginning of *ATEST*. The sizes of the arrays, *MAXT*, *MAXV*, *MAXC*, *MAXB*, *MAXJA*, *MAXA*, and *LENW*, are specified at the beginning of *ATEST* using a *parameter* statement; changing sizes to suit a particular computing environment or problem is thus a simple matter.

To use *ATEST*, the user must provide Fortran subroutines *A1XY*, *A2XY*, *FXY*, *GNXY*, *GDXY*, *P1XY*, *P2XY*, and *QXY*. Subroutine *USRCMD* should be provided, if only as a dummy routine. The user must also supply subroutine *GDATA*, in which the input arrays *VX*, *VY*, *XM*, *YM*, *ITNODE*, and *IBNDRY* are specified, along with some parameters in *IP*, *RP*, *SP*, and possibly *IU*, *RU*, and *SU*. Other entries of the *IP*, *RP*, and *SP* arrays not required to be provided by the user through *GDATA* are given default values at the beginning of *ATEST*, but can be reset by the user as desired.

6.7 Reading and Writing Files.

The *w* and *r* commands are used to save and restore data sets. The arrays *IP*, *RP*, *SP*, *IU*, *RU*, *SU*, *VX*, *VY*, *XM*, *YM*, *IBNDRY*, and *ITNODE*, *JA*, *A*, and portions of *W* corresponding to the current state of the calculation are written to (*w* command) or read from (*r* command) the file *RWFILE*. Data files are formatted as machine independent binary files using the XDR protocol. The *w* and *r* commands can be used with both the triangulation and skeleton data structures.

One can use the w and r commands to save and restore the solution at various points along a continuation path. One can also save solutions in the current run for post processing (graphics, etc.), which can then occur in a later run.

6.8 Journal Files.

The j command causes *ATEST* to read its command strings from the file *JRFILE*, rather than accepting them interactively from the user. It is the only option available in batch mode. A journal file is an ASCII file containing a sequence of command strings as described in Section 6.2. The symbol # appearing as the first character in a line causes that line to be interpreted as a comment. When the end of the file is reached *ATEST* returns to terminal or X-Windows mode and again accepts commands interactively. If a q command is encountered in a journal file, *ATEST* will exit.

6.9 Shell Command.

The k command causes the string stored in the variable *SHCMD* to be executed by the user's shell. It is included mainly as a convenience, in particular as a means to include system file manipulation commands within journal files.

6.10 Subroutine USRCMD.

The u command is used to call the user supplied routine *USRCMD*.

Call USRCMD

This routine is written by the user to perform any tasks not covered by other commands. In our experience, the most frequent use of *USRCMD* has been to reset parameters unique to a particular problem.

USRCMD is affected by the variable *IUSRSW*. If $IUSRSW = 0$, the return from *USRCMD* causes *ATEST* to present the command prompt. If $IUSRSW \neq 0$, the return from *USRCMD* results in a branch to the user supplied routine *GDATA* before presenting the command prompt. This switch is useful if modified parameters affect the geometry of the region, boundary conditions, etc., requiring modifications of the input arrays.

Since the most frequent use of *USRCMD* is to modify problem dependent parameters, we now describe how to build an interface within *USRCMD* allowing one to reset parameters in a fashion similar to the other commands. This is done via subroutine *USRSET*, which is called as follows:

Call USRSET(FILE, LEN, IU, RU, SU)

IU, *RU*, and *SU* are integer, real, and *CHARACTER*80* arrays, respectively, of size 100 containing the parameters to be reset. It is often convenient to use the

IU, *RU*, and *SU* arrays provided by *ATEST* in common block *ATEST2* for this purpose. *FILE* is a *CHARACTER*80* array of length *LEN*, described below. In terminal mode, the command *u* creates a display listing the user parameters and their current values, similar to the upper case form of other commands. Commands of the form

```
command:u name1=value1, name2=value2, ... , namek=valuek
```

reset the indicated parameters and then display the updated values. In X-Windows mode, pressing the *USRCMD* command button with mouse button one pops up a reset window, similar to pressing mouse button three for the other commands.

The array *FILE* contains a list of commands that define the variables to be reset, and characterize the reset display. The commands in *FILE* have a syntax similar to the basic scripting language we have defined for *ATEST* itself. However, in this case there are just two basic commands: *n* (name variable) and *s* (string for radio button). These are summarized in Table 6.2.

Parameters associated with *n* command			
Name	Alias	Type	Value
vname	n	short	maximum of 6 characters
alias	a	short	maximum of 2 characters
vtype	t	short	i (int), r (real), s (short), l (long), f (file)
index	i	int	pointer to *IU*, *RU*, *SU*
Parameters associated with *s* command			
Name	Alias	Type	Value
vname	n	short	variable name
value	v	-	depends on vname
label	l	long	label associated with value in radio buttons

Table 6.2. *Command syntax for USRSET.*

Note that integer variables are stored in the *IU* array, real variables in the *RU* array, and short, long and file strings are all stored as entries in the *SU* array. In order to correctly define the reset window, all four variables associated with the *n* command should be defined in each *n* command. Similarly, the three variables associated with the *s* command should all be defined in each *s* command. Otherwise, the syntax for each command follows the usual rules of the scripting language. Below is an example code fragment that could define a simple *FILE* array.

FILE(1) = 'N I=1, N=NTRI, A=NT, T=I'
FILE(2) = 'N I=2, N=IBC , A=BC, T=I'
FILE(3) = 'S N=IBC, V=1, L="NEUMANN BC"'
FILE(4) = 'S N=IBC, V=2, L="DIRICHLET BC"'
LEN = 4

The first two lines are n commands that define two integer variables. The first line defines a variable with name *NTRI*, alias *NT*, that is stored as *IU(1)*. The second defines a variable *IBC*, alias *BC*, that is stored as *IU(2)*. The variable *IBC* can take on two values, 1 and 2, that are associated with Neumann and Dirichlet boundary conditions, respectively. The third and fourth lines above are s commands that define the structure of a radio box associated with the *IBC* name in the X-Windows popup. Note that since the *LABEL* is a long string, its value must be enclosed in double quotes.

6.11 Subroutine GDATA.

The user provides subroutine *GDATA*, which defines the region through an initial triangulation or a skeleton. A call to *GDATA* is among the first executable statements in *ATEST*.

Call GDATA(VX, VY, XM, YM, ITNODE, IBNDRY,
IP, RP, SP, IU, RU, SU, W)

Through this call the user is minimally expected to supply values for *NTF*, *NVF*, *NCF*, and *NBF* in the *IP* array, as well as the relevant values for the input arrays *VX*, *VY*, *XM*, *YM*, *ITNODE*, and *IBNDRY*. Entries in *RP*, *SP*, *IU* and *RU*, as well as parameters in *IP* other than those mentioned above, may be optionally specified in *GDATA*.

6.12 Machine Dependent Routines.

During the initial installation of the package, the user must provide several machine dependent routines associated with timing and graphics. Default versions of these routines are provided with the package, which should work without modification in many environments, and in any event can serve as a model for a new implementation. The timing routine *TIMER* is used by *PLTMG* and *TRIGEN*. The graphics routines *TRIPLT*, *GPHPLT*, *INPLT*, and *MTXPLT* address the graphics output device through the routines *PLTUTL*, *PFRAME*, *PLINE*, and *PFILL*. These routines are documented in detail below.

6.12.1 Timing Routine.

Subroutine *TIMER* has the calling sequence

Subroutine TIMER(TIME, ISW)

Here *TIME* is a 3×50 real array and *ISW* is an integer. The array *TIME* records the time spent in major subroutines called by *PLTMG* and *TRIGEN*. Timer should call an appropriate system routine to determine the current time each time it is entered, and then take various actions depending on the value of *ISW*. The cases $ISW = -2$ and $ISW = -1$ request initialization of the *TIME* array, while

$1 \leq ISW \leq 50$ request an individual entry in the *TIME* array be updated. The current time is saved as it is needed for the next call to *TIMER*. Subroutine *TIMER* is machine independent except for the call to the system clock. An example of *TIMER*, calling the Unix function *ETIME*, is given below.

```
      subroutine timer(time,isw)
c
            implicit real (a-h,o-z)
            implicit integer (i-n)
            real
     +          time(3,*)
            real temp(2),etime
            save tx,len
            data tx/0.0e0/
            data len/50/
c
c       call the clock and return the time in seconds
c       (time differences are used to compute the elapsed time)
c
        ty=tx
        tx=etime(temp)
c
c       udpate time array (1.0e-10 is below resolution of timer)
c
        if(isw.gt.0) then
            dt=amax1(tx-ty,1.0e-10)
            time(1,isw)=time(1,isw)+dt
            time(2,isw)=time(2,isw)+dt
        else if(isw.eq.-1) then
            do i=1,len
                time(1,i)=0.0e0
            enddo
        else if(isw.eq.-2) then
            do i=1,len
                time(1,i)=0.0e0
                time(2,i)=0.0e0
                time(3,i)=0.0e0
            enddo
        endif
        return
        end
```

6.12.2 Graphics Interface.

The four device dependent routines in the graphics package are

Subroutine PLTUTL(NCOLOR, RED, GREEN, BLUE)
Subroutine PFRAME(IFRAME)
Subroutine PLINE(X, Y, Z, N, ICOLOR)
Subroutine PFILL(X, Y, Z, N, ICOLOR)

Subroutine *PLTUTL* takes various actions depending on the value of the integer *NCOLOR*. $NCOLOR > 0$ specifies initialization; *NCOLOR* denotes the number

of colors to be used and satisfies $2 \le NCOLOR \le MXCOLR$. *RED*, *GREEN*, and *BLUE* are vectors of length *NCOLOR*. The entries *RED(i)*, *GREEN(i)*, and *BLUE(i)*, $1 \le i \le NCOLOR$, are floating point numbers on the interval $[0, 1]$, corresponding to *rgb* values for the ith color. Color number 1 is always white ($RED(1) = GREEN(1) = BLUE(1) = 1.0$), and color number 2 is always black ($RED(2) = GREEN(2) = BLUE(2) = 0.0$). The *rgb* values of the remaining entries depend on the picture to be drawn and the value of *MXCOLR*. *PLTUTL* should create a color map with the required colors, as these will be referenced in future calls to *PLINE* and *PFILL*. If *PLTUTL* is called with $NCOLOR < 0$, the drawing is complete and any necessary post processing should be carried out (e.g., close the plot file).

The drawing space used by the graphics routines is always assumed to be either the unit square $(0, 1) \times (0, 1)$ or the rectangle $(0, 1.5) \times (0, 1)$. For devices that have a so-called *Z*-buffer, the drawing space is either the unit cube $(0, 1) \times (0, 1) \times (0, 1)$ or the brick $(0, 1.5) \times (0, 1) \times (0, 1)$. The graphics display itself is always viewed as rectangular with aspect ratio 3/2, which is either a single rectangular frame or three square frames. These frames are numbered 1 to 4 as illustrated in Figure 6.5. The graphics routines write their output to various *lists*. A list consists of a frame, and certain attributes (rotating/non-rotating, lighted/non-lighted). Some attributes may not have realizations for certain graphics devices. The nine available lists are summarized in Table 6.3.

When graphics is initiated for a certain list, say list k, subroutine *PFRAME(k)* is called to indicate that subsequent calls of *PLINE* and *PFILL* contain data to be written to list k. *PFRAME*$(-k)$ indicates that the output to the given list should be terminated. By convention, graphics routines are allowed only one open list at a time. Therefore, when *PFRAME* is invoked with a positive argument, the given list should be opened and the mapping from the unit cube or brick to the actual device coordinates for the given list should be computed. If rotation or lighting attributes are available, these should be set as specified in Table 6.3. When *PFRAME* is invoked with a negative argument, the given list should be closed.

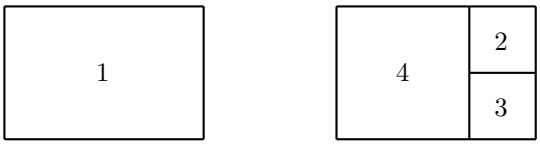


Figure 6.5. *Frame definitions.*

Subroutine *PLINE* has arguments *X*, *Y*, *Z*, *N*, and *ICOLOR*. *X*, *Y*, and *Z* are vectors of length $N \ge 2$. The points *(X(i),Y(i),Z(i))* lie in the unit cube or the brick $(0, 1.5) \times (0, 1) \times (0, 1)$. The *Z* coordinate is useful only for devices that have a *Z*-buffer, and can be ignored in other cases. *ICOLOR* is an integer between 1 and *NCOLOR*, where *NCOLOR* was the argument that initialized *PLTUTL*, indicating

list	frame	rotating	lighted
1	1	no	no
2	2	no	no
3	3	no	no
4	4	no	no
5	4	yes	no
6	4	yes	no
7	4	yes	yes
8	4	yes	yes
9	4	no	yes

Table 6.3. *list specifications for pframe.*

the color to be used. *PLINE* should draw the given polyline *(X(i),Y(i),Z(i))* to *(X(i+1),Y(i+1),Z(i+1))*, $1 \leq i \leq N-1$, with the specified color in the proper frame.

Subroutine *PFILL* has arguments *X*, *Y*, *Z*, *N*, and *ICOLOR*. *X*, *Y*, and *Z* are vectors of length $N \geq 3$. The points *(X(i),Y(i),Z(i))* lie in the unit cube or the brick $(0, 1.5) \times (0, 1) \times (0, 1)$, and define an *N*-sided (planar) polygonal region with sides *(X(i),Y(i),Z(i))* to *(X(i+1),Y(i+1),Z(i+1))* for $1 \leq i \leq N-1$, and *(X(n),Y(n),Z(n))* to *(X(1),Y(1),Z(1))*. *ICOLOR* is an integer between 1 and *NCOLOR*, where *NCOLOR* was the argument that initialized *PLTUTL*, indicating the color to be used. *PFILL* should color the specified polygon with the specified color in the proper frame.

IDEVCE	output driver
0–3	*SG* sockets 0–3
4	BH file
5	Postscript file
6	XPM file
7–10	X-Windows display 0-3

Table 6.4. *Default graphics devices.*

The default installation of the package includes several standard output graphics devices. These are described in Table 6.4. *SG* is an OpenGL program written by Mike Holst that is available separately. It can receive input from a specified INET socket. *ATEST* allows up to four *SG* displays to be accessed. Because it is socket based, *SG* and *ATEST* can be running on different computers; the parameter *SGHOST* is the name of the host computer running *SG*. Since it is based on OpenGL the graphics displays are animated, and images can be manipulated with the mouse.

BH is the protocol developed for communication between *ATEST* and *SG*. BH

files are essentially file versions of *SG* images. The parameter *BHFILE* gives the file name. The parameter *BHFILE* is scanned for the string *FIGXXX*. If found, this string is replaced by *FIG001*, *FIG002*, etc, with the counter incremented for each image. This allows the single parameter *BHFILE* to specify a family of separate BH files. The parameter *BHFILE* is also scanned for the string *MPIXXX*. If found, this string is replaced by *MPI001*, *MPI002*, etc, where the integer part denotes the processor. This avoids potential name conflicts when running *ATEST* as a parallel program. The BH file itself is a device independent binary file written using XDR. These files can be saved and later displayed using the *SG* program.

If the *SG* interface is not available or not desired, an alternate interface composed of stub routines is provided with the default installation of the program. The alternate interface has the same routines as the regular *SG* interface, but with all calls to routines and functions in the *MALOC* library deleted. Using the stub routines, an executable can be created without loading the *MALOC* library to resolve external references. However, if the stub routines are used, the *SG* and BH graphics options are disabled.

Postscript and XPM are both ASCII files. The parameters *PSFILE* and *XPFILE* specify the file names. These names are scanned for the strings *FIGXXX* and *MPIXXX*, that are replaced if found as described above in the case of *BHFILE*. Devices 7–10 refer to X-Windows graphics displays. Up to four such displays may be used (although the *ATEST* driver itself allows up to ten). These graphics windows display static pixmaps (raster images similar to XPM files) that cannot be animated or manipulated, other than resizing the window. X-Windows graphics displays are only available when $MODE = 0$.

6.12.3 X-Windows Interface.

The X-Windows interface uses several X-Windows libraries, as well as the Motif widget set, and thus can be used only in environments that support the X-Windows system. It is based on the release X11R6. Our intent was to make the interface as generic and simple as possible. Since the *PLTMG* package is constantly evolving, the interface is structured to run arbitrary Fortran programs, so that in the future, large changes in the package need not cause correspondingly large changes in the interface. The X-Windows interface is written in C.

If the X-Windows libraries that support the X-Windows interface are not available, one can use substitute stub routines in place of the regular interface. These alternative stub routines are supplied with the default installation of the package, and are similar to those in the regular X-Windows interface, except that all calls to routines and functions in the X-Windows libraries have been deleted. Using the stub routines, an executable can be created without the need to load X-Windows libraries to resolve external references. However, in this case the X-Windows interface ($MODE = 0$) is completely disabled. This includes X-Windows graphics options ($7 \leq IDEVCE \leq 10$).

6.12.4 MPI Interface

The communication used in parallel processing is provided by calls to the MPI library. This library is not provided as part of the *PLTMG* package. The calls to the MPI library are all made from Fortran, and we have concentrated all calls into just a few subroutines. Thus the vast majority of the code comprising the main *PLTMG* routines is self-contained. If the MPI library is not available, one can use substitute stub routines supplied with the default installation in place of the regular interface. The stub routines are similar to the those in the regular interface, except that all calls to routines and functions in the MPI library have been deleted. Using the stub routines, an executable can be created without the need to load the MPI library to resolve external references. In this case, all the parallel computing options provided by *PLTMG*, *TRIGEN*, and the graphics routines are disabled.

Chapter 7
Test Problems

7.1 Overview.

In this chapter, we briefly document the test problem data sets included with the *PLTMG* source code. These problems encompass a variety of applications and exercise most features of the package. Each data set minimally consists of functions *A1XY*, *A2XY*, *FXY*, *GNXY*, *GDXY*, *P1XY*, *P2XY*, and *QXY* and subroutines *USRCMD* and *GDATA*. Problem specific routines are also included.

7.2 Test Problem CIRCLE.

In this problem, we solve the equation

$$-\nabla \cdot (a \nabla u) = 0,$$

where Ω is the unit circle with a crack along the positive x axis. Homogeneous Dirichlet boundary conditions are imposed on the top of the crack, and homogeneous Neumann boundary conditions are imposed below the crack. The coefficient $a \equiv a_k$ is piecewise constant in the eight sectors

$$\Omega_k = \{(r, \theta) |\, 0 \le r \le 1,\ (k-1)\pi/4 \le \theta \le k\pi/4\}.$$

The domain Ω is defined by a triangulation consisting of eight similar triangles, shown in Figure 7.1, that correspond to the eight sectors of constant a. On the boundary of the circle, nonhomogeneous boundary conditions are imposed such that the true solution in sector Ω_k is given by

$$u = r^{\alpha}(\beta_k \sin \alpha\theta + \gamma_k \cos \alpha\theta). \tag{7.1}$$

The exponent α is chosen to correspond to the leading singularity arising from the geometry, change of boundary conditions, and coefficient jumps at the origin. The coefficients β_k and γ_k are chosen to insure continuity of the solution u and the normal component of the flux $a\nabla u \cdot n$ across the interfaces, and to satisfy the

boundary conditions along the crack. For example, in the case $a_k = 1$ for all k, $\alpha = 1/4$ and

$$u = r^\alpha \sin \alpha\theta.$$

The *USRCMD* for this test problem has ten parameters that can be set. *IBC* determines the boundary conditions. If $IBC = 2$, the boundary conditions on the outer boundary of the circle are nonhomogeneous Dirichlet chosen such that (7.1) is the exact solution; if $IBC = 1$, nonhomogeneous Neumann boundary conditions are imposed on the circular part of the boundary in a similar fashion. One can also alter the geometry of the domain using the parameter *NTRI*, where $1 \le NTRI \le 8$. If $NTRI = 8$ the entire circle is used as the domain; if $NTRI < 8$, only the first *NTRI* sectors are used. Some examples are shown in Figure 7.1. Eight parameters,

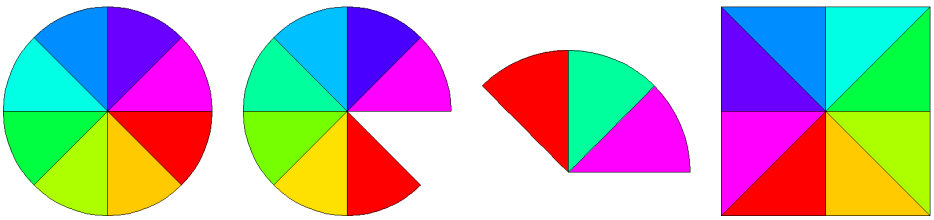


Figure 7.1. *On the far right is the square domain for problems SQUARE, OB and CONTROL. The remaining domains are for test problem CIRCLE with* $NTRI = 8$, $NTRI = 7$ *and* $NTRI = 3$.

A1, *A2*,...,*A8* define the coefficients a_k. Given the a_k and *NTRI*, the values of α, β_k and γ_k are computed in *GDATA* by solving appropriate nonlinear equations. Since the exact solution is known, we can compute the exact error. For this test problem, the function *QXY* is defined to be the exact error for graphics options and the true solution (7.1) otherwise.

7.3 Test Problem SQUARE.

In this test problem, a complicated equation is solved on a simple domain. The domain is always the unit square shown in Figure 7.1; boundary conditions on each side of the square can be independently specified as Dirichlet or natural, or pairs of opposite sides can be specified as periodic. The region is specified as a triangulation.

The coefficient functions are defined by

$$
\begin{aligned}
a_1 &= A1X\,\frac{\partial u}{\partial x} + A1Y\,\frac{\partial u}{\partial y} + A1U\,u,\\
a_2 &= A2X\,\frac{\partial u}{\partial x} + A2Y\,\frac{\partial u}{\partial y} + A2U\,u,\\
f &= -BUX\,\frac{\partial u}{\partial x} - BUY\,\frac{\partial u}{\partial y} - F0(y-x) - CU0 - CU1\,u - CU2\,u^2\\
&\quad - CIR\left(\frac{\partial u}{\partial x}(y-.5) - \frac{\partial u}{\partial y}(x-.5)\right) - CEXP\,e^{u(1+EPS\,u)^{-1}} - CSIN\,\sin u,\\
g_1 &= -DU0 - DU1\,u,\\
g_2 &= -EU0,
\end{aligned}
$$

and the functional ρ is defined by

$$
\begin{aligned}
p_1 &= u^2,\\
p_2 &= 0.
\end{aligned}
$$

Any of these nineteen parameters can be set using *USRCMD*, and any can be used as the continuation parameter λ by specifying the parameter *ICONT* in *USRCMD* as in Table 7.1. With this variety of nonlinearities, one can exercise most continuation features of *PLTMG*.

ICONT	λ	*ICONT*	λ
0	none	10	*CU1*
1	*A1X*	11	*CU2*
2	*A1Y*	12	*CIR*
3	*A1U*	13	*CEXP*
4	*A2X*	14	*EPS*
5	*A2Y*	15	*CSIN*
6	*A2U*	16	*DU0*
7	*BUX*	17	*DU1*
8	*BUY*	18	*EU0*
9	*CU0*	19	*F0*

Table 7.1. *Possible settings for ICONT.*

If $ICONT = 0$, then none of the parameters is regarded as λ, and one should set $IPROB = 1$ to signify that the problem does not involve continuation.

One can also set the integer parameters *LEFT*, *RIGHT*, *TOP*, and *BOTTOM* in *USRCMD*. These refer to the four sides of the square in an obvious fashion and can be individually set to 2 for Dirichlet boundary conditions or to 1 for natural boundary conditions for the given side of the square. A pair of opposite edges can be set to 0 (e.g., $TOP = BOTTOM = 0$), and *IBNDRY* will then be set for periodic boundary conditions.

7.4 Test Problem DOMAINS.

In this test problem, a simple equation is solved on a variety of complicated domains. This test problem was designed mainly to exercise *TRIGEN*.

The problem to be solved is the linear partial differential equation

$$a_1 = A1X\,\frac{\partial u}{\partial x} + A1Y\,\frac{\partial u}{\partial y} + A1U\,u,$$
$$a_2 = A2X\,\frac{\partial u}{\partial x} + A2Y\,\frac{\partial u}{\partial y} + A2U\,u,$$
$$f = -BUX\,\frac{\partial u}{\partial x} - BUY\,\frac{\partial u}{\partial y} - CU1\,u - CU0$$

with a combination of homogeneous Dirichlet, homogeneous Neumann, and periodic boundary conditions. The parameters *A1X*, *A1Y*, *A1U*, *A2X*, *A2Y*, *A2U*, *BUX*, *BUY*, *CU0*, and *CU1* can all be set in *USRCMD*. The parameter *DOMAIN*, satisfying $1 \le DOMAIN \le 20$, specifies the domain to be used. The various possibilities are shown in Figure 7.2. All domains are defined by skeletons, so *TRIGEN* must be called to generate a triangulation.

7.5 Test Problem NACA.

Test problem *NACA* solves the equation of potential flow in one of several domains. The equation is of the form

$$-\nabla \cdot \rho(\nabla u)\nabla u = 0,$$

where

$$\rho(\nabla u) = (1 - u_x^2 - u_y^2)^{\frac{1}{\gamma-1}}$$

and $\gamma = 1.4$. The local Mach number is computed in *QXY* and is given by

$$q = \sqrt{\frac{2c}{\gamma - 1}},$$
$$c = \frac{1}{1 - u_x^2 - u_y^2} - 1.$$

There are seven domain options, chosen using the parameter *DOMAIN* in *USRCMD*. These domains are shown in Figure 7.3. All regions are defined as skeletons, so *TRIGEN* must be used to generate a triangulation.

Neumann boundary conditions are imposed everywhere so each domain has $IDBC \neq 0$. There are several parameters in *USRCMD* that affect these problems. The parameter *MINF*, specifying the Mach number at infinity M_∞, sets the boundary conditions on the outer boundary and is also the continuation parameter λ for these problems. The parameter *ANGLE* specifies the angle of attack (in degrees). The parameter *SIZE* sets the radius of the outer boundary. The parameter *RATIO* affects only the case $DOMAIN = 6$ and is used to set the ratio for the principal axes of the ellipse ($RATIO = 1$ corresponds to a circle).

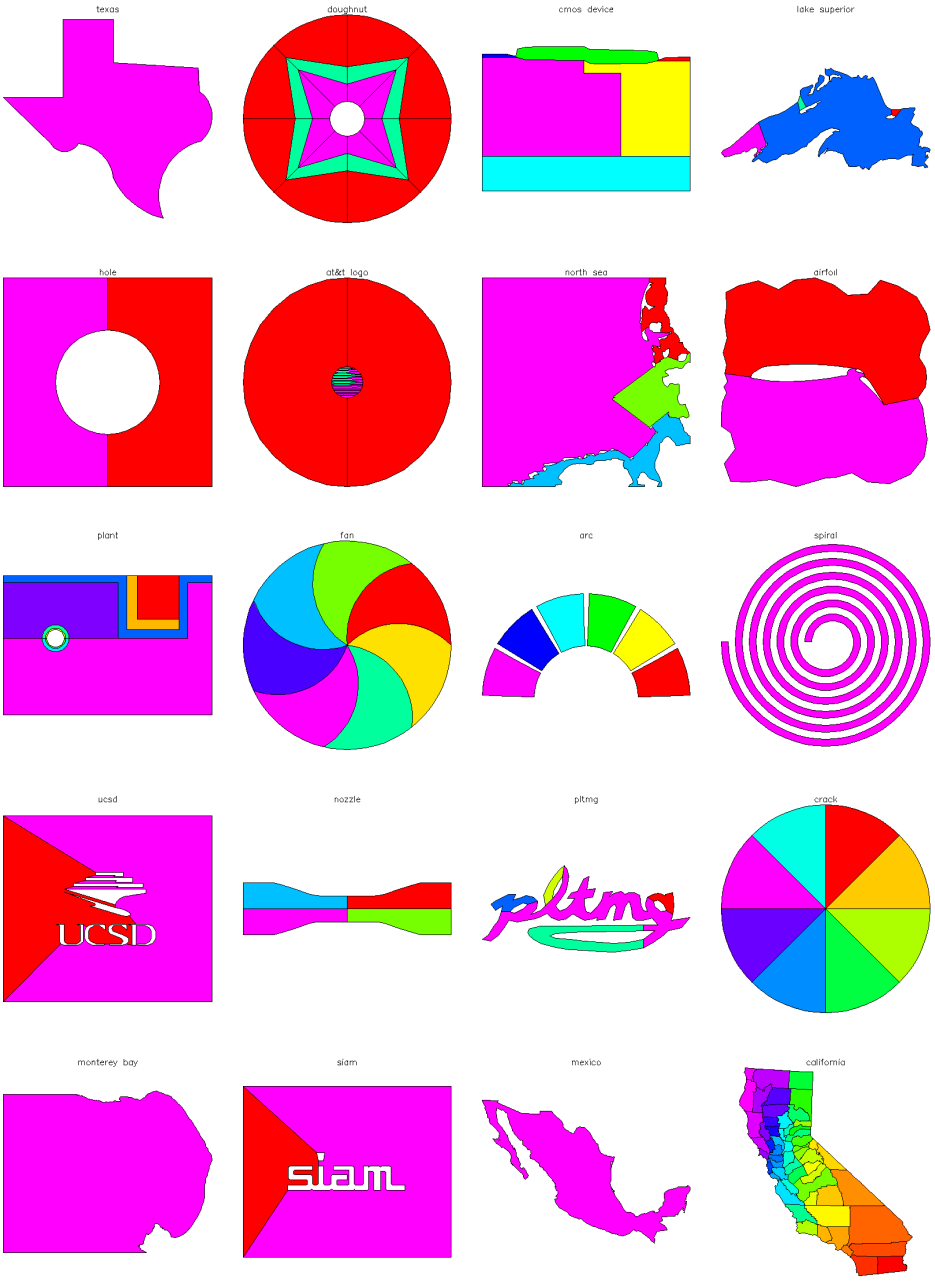


Figure 7.2. *The domains for* $DOMAIN = i$, $1 \leq i \leq 20$.

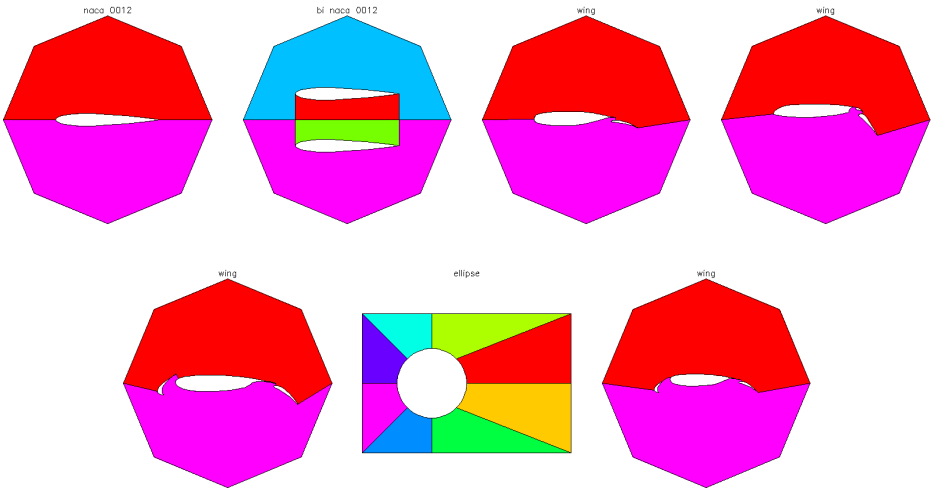


Figure 7.3. *The domains for DOMAIN* $= i$, $1 \le i \le 7$, *with SIZE* $= 1$.

When the local Mach number is less than one the flow is subsonic; *PLTMG* will work well in regions where the flow is entirely subsonic. As the M_∞ is increased, the solution will begin to develop regions of supersonic flow near the airfoils; *PLTMG* will continue to work as these regions are forming, but eventually will fail, as the underlying discretization used by *PLTMG* is not really appropriate for hyperbolic problems.

7.6 Test Problem JCN.

Test problem *JCN* solves the convection diffusion equation

$$-\nabla \cdot (\nabla u + \beta u) = 0,$$

where β is piecewise constant. The region is shown in Figure 7.4. The domain is specified by skeleton, so *TRIGEN* must be used to generate a triangulation.

This problem is an idealized model of the current continuity equation from the semiconductor device model that we have used to study the stability of discretizations used in device simulation. The problem has seven regions; $\beta = 0$ in regions one and seven. In the other five regions it has a magnitude of approximately 10^4 and is directed radially in each of the five subregions. The solution develops steep gradients at the junction between region seven and the five adjoining subregions.

Constant nonhomogeneous Dirichlet boundary conditions are specified along the bottom of the domain and on the left-hand portion of the top of the domain. Homogeneous Neumann boundary conditions are imposed elsewhere. The parameters *TOP* and *BOTTOM* in *USRCMD* can be used to reset the Dirichlet boundary conditions on the top and bottom of the domain. The parameter *DU* can be used to adjust the size of β in regions 2–5; in particular, the magnitude of β in these five

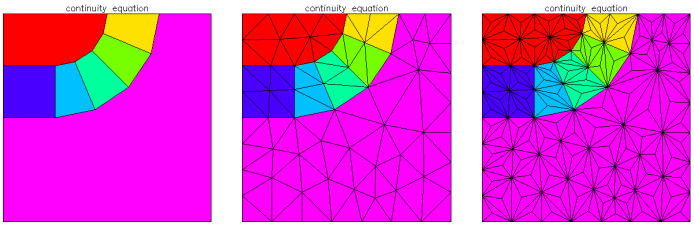


Figure 7.4. *The domain for test problem JCN (left), a triangulation produced by TRIGEN (middle), and the corresponding triangulation after a call to USRCMD with OBTUSE* $= 1$ *(right).*

regions is proportional to *DU*.

Our original purpose in constructing this example was to test the sensitivity of various upwinding techniques [4] to poor element geometries. Since the goal of *TRIGEN* is to produce elements with good geometries, the *USRCMD* for this problem includes a procedure for systematically degrading the quality of the triangulation by introducing new elements with obtuse angles. If *OBTUSE* $= 1$ in *USRCMD*, then each triangle in the current mesh is divided into three new triangles by connecting its barycenter to its vertices. An example is shown in Figure 7.4. Repeated application of this procedure will produce triangulations with interior angles arbitrarily close to π.

7.7 Test Problem OB.

Test problem *OB* solves the a simple obstacle problem, with coefficient functions defined by

$$
\begin{aligned}
p_1 &= AX\left(\frac{\partial u}{\partial x}\right)^2 + AY\left(\frac{\partial u}{\partial y}\right)^2 + CU\,u^2 - 2su,\\
s &= \big(AX(IX\pi)^2 + AY(IY\pi)^2 - CU\big)\sin(IX\pi x)\sin(IY\pi y),\\
\underline{u} &= BDLW + CFLW\sin(IXL\pi x)\sin(IYL\pi y),\\
\overline{u} &= BDUP + CFUP\sin(IXU\pi x)\sin(IYU\pi y),\\
g_1 &= 0.
\end{aligned}
$$

The domain Ω is the unit square with homogeneous Dirichlet boundary conditions. The input data structure is a triangulation consisting of eight right triangles, shown in Figure 7.1. The parameters *AX*, *AY*, *CU*, *BDLW*, *BDUP*, *CFLW*, *CFUP* and the integers *IX*, *IY*, *IXL*, *IYL*, *IXU*, *IYU* can all be set by the user in *USRCMD*. The exact solution to this problem in the absence of the obstacle is $u = \sin(IX\pi x)\sin(IY\pi y)$. This problem is mainly designed to test the cases *IPROB* $= \pm 2$ in *PLTMG*.

7.8 Test Problem MNSURF.

Test problem *MNSURF* solves the a simple minimal surface problem with an obstacle. The coefficient functions are given by

$$p_1 = \sqrt{1 + \left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2}$$

$$\underline{u} = \begin{cases} 1 & \text{in } \Omega_1 \\ -1 & \text{in } \Omega_2 \\ \ell & \text{in } \Omega_I \end{cases}$$

$$\overline{u} = 1.5$$

$$g_1 = 0,$$

$$g_2 = 0.$$

The domain Ω is the unit square with a mixture of homogeneous Dirichlet and Neumann boundary conditions. The domain is given as a skeleton, and is shown in Figure 7.5. The region Ω_1 is the inner square with side $1/2$, and Ω_2 is the outer region. The region Ω_I is the small band separating Ω_1 and Ω_2, consisting of four narrow trapezoids. In each of the four trapezoids, $\underline{u}$ is a linear polynomial in x or y that interpolates between -1 and 1, insuring continuity of $\underline{u}$. The parameter *THETA*, which can be set in *USRCMD*, controls the width of the band. The upper bound $\overline{u}$ is chosen such that it does not affect the solution. As with test problem *OB*, this problem is mainly designed to test the cases *IPROB* $= \pm 2$ in *PLTMG*.

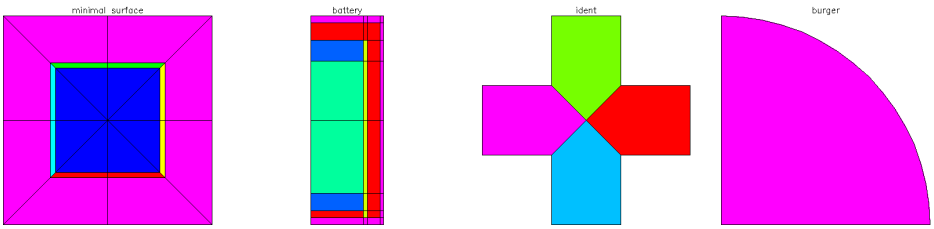


Figure 7.5. *The domains for test problems MNSURF, BATTERY, IDENT, and BURGER (left to right).*

7.9 Test Problem BURGER.

Test problem *BURGER* solves the nonlinear convection dominated flow

$$-\epsilon \Delta u + u_y + u u_x = 0.$$

The small parameter $\epsilon > 0$ and can be set in *USRCMD*. If $\epsilon = 0$, this is the one dimensional Burger's equation with y playing the role of time. The domain Ω is

the quarter circle shown in Figure 7.5, and is specified as a skeleton. Homogeneous Neumann boundary conditions are applied along the circular arc, while Dirichlet boundary conditions are specified on the left side ($x = 0$) and the bottom ($y = 0$) as

$$g_2 = \begin{cases} 1 & 0 \le x \le 1/4 \\ 3/2 - 2x & 1/4 \le x \le 3/4 \\ 0 & 3/4 \le x \le 1 \end{cases}.$$

This combination of boundary conditions gives rise to a solution similar to the so-called "λ shock" of Burger's equation.

7.10 Test Problem BATTERY.

In this test problem we solve the linear elliptic problem

$$-a_1 u_{xx} - a_2 u_{yy} - f = 0$$

where the piecewise constant values of the coefficients are given in Table 7.2. The

Region	a_1	a_2	f	side	c	α
1	25	25	0	left	0	0
2	7	0.8	1	top	1	3
3	5.0	10^{-4}	1	right	2	2
4	0.2	0.2	0	bottom	3	1
5	0.05	0.05	0			

Table 7.2. *Coefficient definitions.*

domain Ω is shown in Figure 7.5 and is specified as a skeleton. The five subregions are given labels in *ITNODE(5,*)*, allowing us to conveniently define the coefficient functions. The boundary conditions are natural boundary conditions of the form

$$g_1 = c - \alpha u.$$

Here c and α are piecewise constant functions defined using *IBNDRY(6,*)*, as indicated in Table 7.2. The data for this problem was supplied by Leszek Demkowicz.

7.11 Test Problem CONTROL.

This problem tests the cases $IPROB = \pm 5$. The differential equation (constraint) is

$$-\Delta u = \lambda$$

and the objective function ρ is given by

$$\rho(u, \lambda) = \int_\Omega \beta(u - u_0)^2 + \gamma\lambda^2 \, dx.$$

Ω is the unit square, defined as a triangulation similar to test problem *SQUARE*. see Figure 7.1. The function u_0 and the bounds on λ are given by

$$\begin{aligned} u_0 &= \sin(IX\pi x)\sin(IY\pi y), \\ \underline{\lambda} &= BDLW + CFLW\sin(IXL\pi x)\sin(IYL\pi y), \\ \overline{\lambda} &= BDUP + CFUP\sin(IXU\pi x)\sin(IYU\pi y). \end{aligned}$$

Homogeneous Dirichlet boundary conditions are applied ($g_2 = 0$). The constants $BETA = \beta$. $GAMMA = \gamma$, *BDLW*, *BDUP*, *CFLW*, and *CFUP*, and the integers *IX*, *IY*, *IXL*, *IYL*, *IXU*, and *IYU* can all be reset in *USRCMD*.

7.12 Test Problem IDENT.

This problem tests the cases $IPROB = \pm 4$. The differential equation is

$$-(1 + A^2)\Delta u + C2\,u^2 + C1\,u - C0\ = 0.$$

The domain Ω is specified as a skeleton, and is shown in Figure 7.5. The boundary conditions are a combination of homogeneous Neumann and Dirichlet, except for the vertical edge on the right where the (possibly) inhomogeneous Dirichlet boundary condition

$$u = D$$

is imposed. The five parameters A, $C0$, $C1$, $C2$, and D can be set in *USRCMD*, and any can be used as the scalar parameter λ in the optimization problem. This is done setting the parameter *IRL* (also set in *USRCMD*) as indicated in Table 7.3.

IRL	λ
1	A
2	$C0$
3	$C1$
4	$C2$
5	D

Table 7.3. *Possible settings for IRL.*

The objective function ρ is given by

$$\rho(u, \lambda) = \int_\Omega e^{-20(x^2+y^2)}(u-1)^2\,dx,$$

which tries to make the solution $u = 1$ near the origin, located at the center of Ω. Upper and lower bounds for λ are set using the parameters *RLLWR* and *RLUPR* in the *RP* array.

7.13 Test Problem MESSAGE.

In this test problem, a simple equation is solved on a domain consisting of a message with up to ten lines. This test problem was designed mainly for fun, and to make software demonstrations more interesting.

The problem to be solved is the linear partial differential equation

$$a_1 = A1X \frac{\partial u}{\partial x} + A1Y \frac{\partial u}{\partial y} + A1U\, u,$$
$$a_2 = A2X \frac{\partial u}{\partial x} + A2Y \frac{\partial u}{\partial y} + A2U\, u,$$
$$f = -BUX \frac{\partial u}{\partial x} - BUY \frac{\partial u}{\partial y} - CU1\, u - CU0$$

with homogeneous Dirichlet boundary conditions. The parameters *A1X*, *A1Y*, *A1U*, *A2X*, *A2Y*, *A2U*, *BUX*, *BUY*, *CU0*, and *CU1* can all be set in *USRCMD*. String parameters *LINE0*, *LINE1*, ... , *LINE9* can be set in *USRCMD* to a user specified message. Upper case and lower case letters, numbers, and several symbols found on a standard keyboard are available. Two possible domains are shown in Figure 7.6. All domains are defined by skeletons, so *TRIGEN* must be called to generate a triangulation.

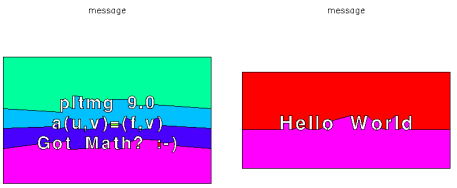


Figure 7.6. *Sample domains for test problem MESSAGE.*

Bibliography

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

[2] I. Babuška and T. Strouboulis, *The finite element method and its reliability*, Numerical Mathematics and Scientific Computation, The Clarendon Press Oxford University Press, New York, 2001.

[3] R. E. Bank, *Multigraph users' guide - version 1.0*, tech. report, Department of Mathematics, University of California at San Diego, 2001.

[4] R. E. Bank, J. F. Bürgler, W. Fichtner, and R. K. Smith, *Some upwinding techniques for finite element approximations of convection-diffusion equations*, Numer. Math., 58 (1990), pp. 185–202.

[5] R. E. Bank and T. F. Chan, *PLTMGC: A multi-grid continuation program for parameterized nonlinear elliptic systems*, SIAM J. Sci. and Stat. Computing, 7 (1986), pp. 540–559.

[6] ——, *An analysis of the composite step biconjugate gradient method*, Numer. Math., 66 (1993), pp. 295–319.

[7] ——, *The composite step biconjugate gradient algorithm for nonsymmetric linear systems*, Numerical Algorithms, 7 (1994), pp. 1–16.

[8] R. E. Bank, W. M. Coughran, and L. C. Cowsar, *Analysis of the finite volume Scharfetter-Gummel method for steady convection diffusion equations*, Computing and Visualization in Science, 1 (1998), pp. 123–136.

[9] R. E. Bank, P. E. Gill, and R. F. Marcia, *Interior methods for a class of elliptic variational inequalities*, in Large-scale PDE-constrained Optimization, L. T. Biegler, O. Ghattas, M. Heinkenschloss, and B. van Bloemen Waanders, eds., vol. 30 of Lecture Notes in Computational Science and Engineering, Berlin, Heidelberg and New York, 2003, Springer-Verlag, pp. 218–235.

[10] R. E. Bank and M. J. Holst, *A new paradigm for parallel adaptive meshing algorithms*, SIAM J. on Scientific Computing, 22 (2000), pp. 1411–1443.

[11] ——, *A new paradigm for parallel adaptive meshing algorithms*, SIAM Review, 45 (2003), pp. 292–323.

[12] R. E. Bank and P. K. Jimack, *A new parallel domain decomposition method for the adaptive finite element solution of elliptic partial differential equations*, Concurrency and Computation: Practice and Experience, 13 (2001), pp. 327–350.

[13] R. E. Bank, P. K. Jimack, S. A. Nadeem, and S. V. Nepomnyaschikh, *A weakly overlapping domain decomposition preconditioner for the finite element solution of elliptic partial differential equations*, SIAM J. on Scientific Computing, 23 (2002), pp. 1817–1841.

[14] R. E. Bank and S. Lu, *A domain decomposition solver for a parallel adaptive meshing paradigm*, SIAM J. on Scientific Computing, (to appear).

[15] R. E. Bank and H. D. Mittelmann, *Continuation and multigrid for nonlinear elliptic systems*, in Multigrid Methods II: Proceedings, Cologne 1985, vol. 1228 of Lecture Notes in Mathematics, Springer-Verlag, Heidelberg, 1986, pp. 24–38.

[16] ——, *Stepsize selection in continuation procedures and damped Newton's method*, J. Comput. Appl. Math., 26 (1989), pp. 67–78.

[17] R. E. Bank and D. J. Rose, *Global approximate Newton methods*, Numer. Math., 37 (1981), pp. 279–295.

[18] ——, *A multi-level Newton method for nonlinear finite element equations*, Math. Comp., 39 (1982), pp. 453–465.

[19] R. E. Bank and R. K. Smith, *Mesh smoothing using a posteriori error estimates*, SIAM J. Numer. Anal., 34 (1997), pp. 979–997.

[20] R. E. Bank and R. K. Smith, *Multigraph algorithms based on sparse Gaussian elimination*, in Thirteenth International Symposium on Domain Decomposition Methods for Partial Differential Equations, Domain Decomposition Press, Bergen, 2001, pp. 15–26.

[21] ——, *An algebraic multilevel multigraph algorithm*, SIAM J. on Scientific Computing, 25 (2002), pp. 1572–1592.

[22] R. E. Bank and J. Xu, *Asymptotically exact a posteriori error estimators, part I: Grids with superconvergence*, SIAM J. Numerical Analysis, (to appear).

[23] ——, *Asymptotically exact a posteriori error estimators, part II: General unstructured grids*, SIAM J. Numerical Analysis, (to appear).

[24] M. W. Beall and M. S. Shephard, *A general topology-based mesh data structure*, Internat. J. Numer. Methods Engrg., 40 (1997), pp. 1573–1596.

[25] X. Cai and K. Samuelsson, *Parallel multilevel methods with adaptivity on unstructured grids*, 1999. Preprint.

[26] T. F. Chan, P. Ciarlet, and W. K. Szeto, *On the optimality of the median cut spectral bisection method*, SIAM J. Sci. Comput., 18 (1997), pp. 943–948.

[27] H. L. deCougny, K. D. Devine, J. E. Flaherty, R. M. Loy, C. Ozturan, and M. S. Shephard, *Load balancing for the parallel adaptive solution of partial differential equations*, Appl. Num. Math., 16 (1994), pp. 157–182.

[28] J. E. Flaherty, R. M. Loy, C. Ozturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz, *Parallel structures and dynamic load balancing for adaptive finite element computation*, Appl. Num. Math., 26 (1998), pp. 241–263.

[29] S. Kohn, J. Weare, M. E. Ong, and S. B. Baden, *Software abstractions and computational issues in parallel structured adaptive mesh methods for electronic structure calculations*, in Workshop on Structured Adaptive Mesh Refinement Grid Methods, Institute for Mathematics and Its Applications, University of Minnesota, Minneapolis, MN., 1997.

[30] W. F. Mitchell, *A comparison of adaptive refinement techniques for elliptic problems*, ACM Trans. Math. Software, 15 (1989), pp. 326–347.

[31] ——, *The full domain partition approach to distributing adaptive grids*, Applied Numerical Mathematics, 26 (1998), pp. 265–275.

[32] ——, *A parallel multigrid method using the full domain partition*, Electronic Transactions on Numerical Analysis, 6 (1998), pp. 224–233.

[33] H. D. Mittelmann, *Multi-grid continuation and spurious solutions for nonlinear boundary value problems*, Rocky Mountain J. Math., 18 (1988), pp. 387–401.

[34] H. D. Mittelmann and H. Weber, *Multigrid solution of bifurcation problems*, SIAM J. Sci. Stat. Comp., 6 (1985), pp. 49–60.

[35] B. N. Parlett, *The Symmetric Eigenvalue Problem*, Prentice Hall, Englewood Cliffs, New Jersey, 1980.

[36] A. Pothen, H. D. Simon, and K.-P. Liou, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 430–452.

[37] M. C. Rivara, *Mesh refinement processes based on the generalized bisection of simplices*, SIAM J. Numer. Anal., 21 (1984), pp. 604–613.

[38] P. M. Selwood, M. Berzins, and P. M. Dew, *3D parallel mesh adaptivity: Data structures and algorithms*, in Parallel Processing for Scientific Computing, Philadelphia, 1997, SIAM.

[39] H. D. Simon and S.-H. Teng, *How good is recursive bisection?*, SIAM J. Sci. Comput., 18 (1997), pp. 1436–1445.

[40] R. Verfürth, *A Posteriori Error Estimation and Adaptive Mesh Refinement Techniques*, Teubner Skripten zur Numerik, B. G. Teubner, Stuttgart, 1995.

[41] C. Walshaw and M. Berzins, *Dynamic load balancing for pde solvers on adaptive unstructured meshes*, Concurrency: Practice and Experience, 7 (1995), pp. 17–28.

[42] A. Weiser, *Local-Mesh, Local-Order Adaptive Finite Element Methods with A-Posteriori Error Estimators for Elliptic Partial Differential Equations*, PhD thesis, Yale University, 1981.

Index