033762-2-T

UPGRADE OF FEMA-PRISM FOR LOG-PERIODIC ANTENNAS

M.D. Casciato, T. Ozdemir, M. Carr and J.L. Volakis

Naval Air Warfare Center Weapons Div.
China Lake, CA 93555-6001

December 1996

33762-2-T = RL-2465

PROJECT INFORMATION

PROJECT TITLE: Further Development of FEMATS, Including Prismatic Meshes, Graphical Interface and New Mesh Truncation Schemes

REPORT TITLE: Upgrade of FEMA-PRISM for Log-Periodic Antenna Analysis

REPORT No.: 033762-2-T

DATE: December 1996

SPONSOR:

Dr. Helen Wang
Code 472210D, Bldg 01400
Naval Air Warfare Center Weapons Div.
China Lake, CA 93555-6001
Phone: (619) 939-3931

Dr. A. R. (Ron) Skatvold
Code 472320D
Naval Air Warfare Center-Weapons Div.
China Lake, CA 93555-6001
Phone: (619) 939-8915
Fax: (619) 939-6594

SPONSOR P.O. or
CONTRACT No.: N68936-95-M-E248

U-M PRINCIPAL
INVESTIGATOR:

John L. Volakis
EECS Dept.
University of Michigan
1301 Beal Ave
Ann Arbor, MI 48109-2122
Phone: (313) 764-0500 FAX: (313) 647-2106

CONTRIBUTORS
TO THIS REPORT: M. D. Casciato, T. Ozdemir, M. Carr and J.L. Volakis

Upgrade of FEMA-PRISM for Log-Periodic Antennas

M.D. Casciato and J. L. Volakis

Radiation Laboratory
Department of Electrical Engineering & Computer Science
University of Michigan
Ann Arbor, Michigan 48109-2122

This Package contains the following:

Upgrade of FEMA-PRISM for Log-Periodic Antenna Anlysis

M.D. Casciato and J. L. Volakis

Radiation Laboratory
Department of Electrical Engineering & Computer Science
University of Michigan
Ann Arbor, Michigan 48109-2122
December 18, 1996

This Package contains the following:

1. Project Report.
2. Paper on FEM techniques used in FEMA-PRISM.
3. User's Manual for FEMA-PRISM with updated input file format.
4. AutoCAD FEMA-PRISM Interface Manual.
5. Triangle 2D Mesh Generator User's Manual.
6. Paper on meshing techniques used in Triangle.
7. Paper on automated design of log-periodic geometries.
8. Listing and description of all directories in code delivery.

Upgrade of FEMA-PRISM for Log-Periodic Antenna Analysis

M. D. Casciato and J. L. Volakis

Radiation Laboratory
Department of Electrical Engineering & Computer Science
University of Michigan
Ann Arbor, Michigan 48109-2122
December 18, 1996

Abstract

This report describes the interfacing upgrades and modifications made to the Finite Element (FEM) antenna simulation code FEMA-PRISM, for the analysis of log-periodic antenna structures. A description and background of the code are given including meshing and termination schemes used and why. Limitations of the current methods are discussed along with additional code modifications in progress to improve performance. Details of upgrades and modifications are given. A 2D triangular mesh generator, TRIANGLE, was used in mesh generation. Pre-processing interfaces were written to convert an AutoCAD .dxf line file to a TRIANGLE input file, and to convert TRIANGLE output files to the FEMA-PRISM input file "surfmesh". Examples runs are shown of both a single element folded slot and a 7-element log-periodic folded slot array (LPFSA).

Introduction

Over the past 10 years we have witnessed an increasing reliance on computational methods for the characterization of electromagnetic problems. Although traditional integral equation methods continue to be used for many applications, one can safely state that in recent years the greatest progress in computational electromagnetics has been in the development and application of partial differential equation(PDE) methods such as the finite difference-time domain and finite element (FEM) methods, including hybridizations of these with integral equation and high frequency techniques. The major reasons for the increasing reliance on PDE methods stem from their inherent geometrical adaptability, low O(N) memory demand and their capability to model heterogeneous (isotropic or anisotropic) geometries. These attributes are essential in developing general-purpose codes for electromagnetic analysis/design, including antenna design and characterization. Other attributes of the finite element method are:

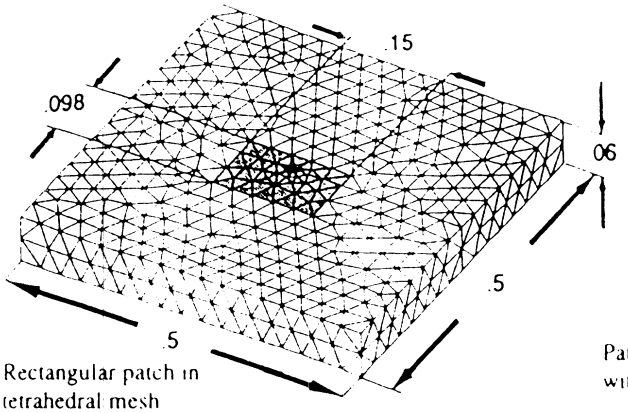
Input data to FEM software can be extracted directly from commercially available (i.e. well-supported) solid modeling packages which run on all popular workstation platforms and are well documented. This is particularly important to problems in antenna analysis and design, where a high degree of geometrical fidelity must be maintained(see Figure 1 for examples of antenna and circuit meshes).

FEM is totally insensitive to the material composition of the radiating or scattering structure. Also, resistive/material and impedance boundary conditions are readily implemented in a modular fashion.

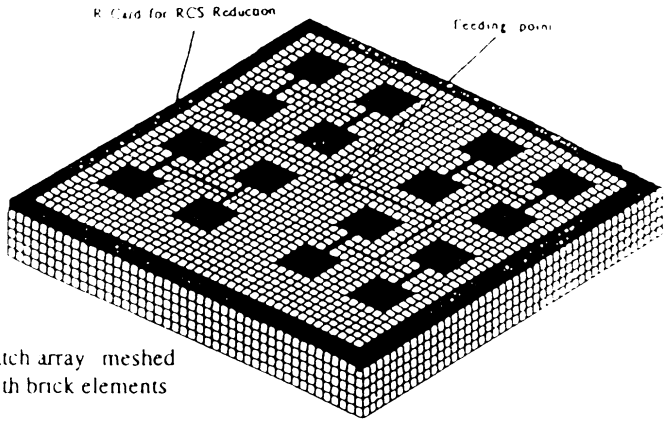
Being a near-neighbor method, new "physics" can be added to the FEM codes without a need to alter the original code structure. Neither the moment method nor the finite difference method share this feature.

Being a frequency domain method, the FEM is ideal for antenna analysis and design purposes. Established hybridizations of the FEM with moment method and ray methods provide an added advantage by delivering the most adaptable and efficient code when compared to other approaches.

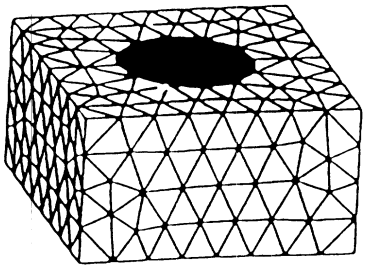
Advances in mesh termination schemes have relaxed accuracy compromises with that aspect of the method. Also, the FEM can benefit from recent fast integral equation algorithms. One may therefore think of the FEM as the core method for treating the heterogeneous volumetric structures including feeds and loads



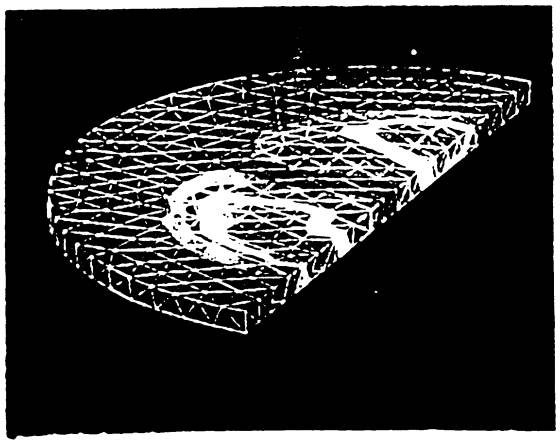
Rectangular patch in tetrahedral mesh



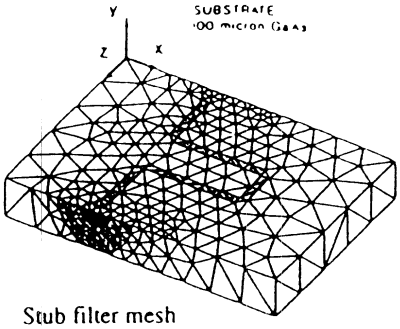
Patch array meshed with brick elements



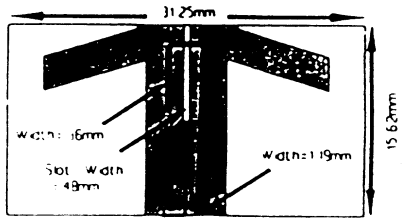
Circular patch in tetrahedral mesh



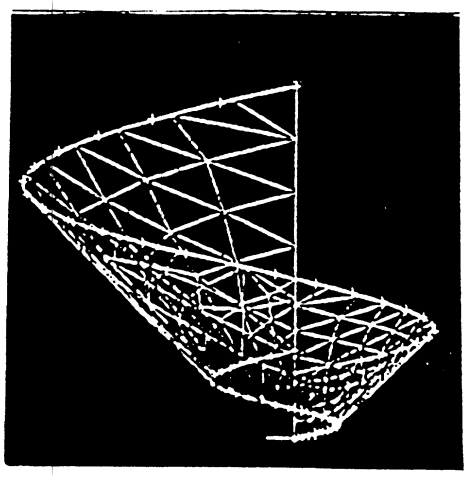
Mesh and fields of dual patch antenna (aperture coupled feed) at resonance



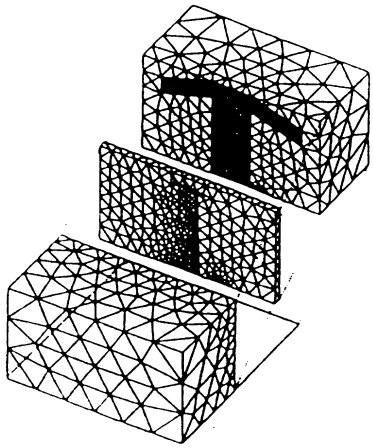
Stub filter mesh



Printed dipole radiator design



Spiral antenna surface mesh (one arm)



Printed dipole geometry and mesh

Figure 1 Illustration of finite element meshes for antenna and circuit problems.

Fine geometrical details such as those found in spiral antennas and the feeding structure can be modeled without sacrificing geometrical fidelity.

The immediate output of FEM codes is the near zone fields which can be readily visualized (superimposed with the geometry) and further processed using commercially available tools. Moment method codes do not share this important feature. This inherent aspect of the FEM codes allows for extraction and visualization of many different parameters as needed by the user.

Several finite element codes have been developed at the Univ. of Michigan for the analysis and design of printed antenna configurations. Typically, the printed antenna configuration is assumed to be recessed in some metallic or coated platform and the various codes differ in the element used for the tessellation of the antenna, the type of platform assumed in the analysis (planar, cylindrical or doubly curved) and the closure condition employed for terminating the finite element mesh. FEMA-PRISM, the code being upgraded, uses prismatic elements and an artificial absorber (AA) mesh termination scheme. A brief description follows.

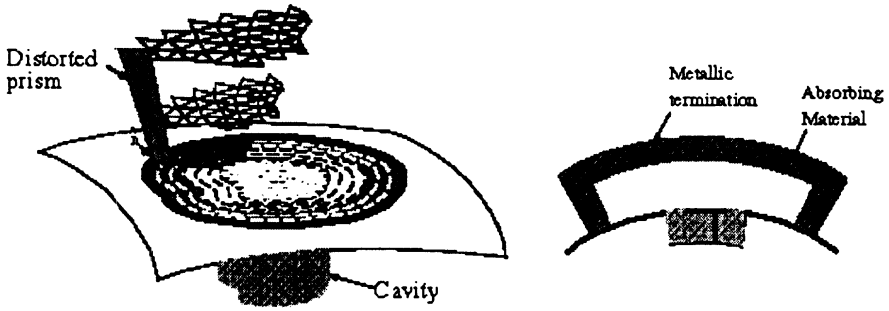
FEMA-PRISM--Background

FEMA-PRISM is a conformal antenna analysis code employing Finite Element (FEM) numerical techniques to solve Maxwell's equations. It employs a distorted prismatic volume mesh and an Artificial Absorber (AA) termination scheme to solve for near fields on and in the antenna. Post processing routines [1] generate far-field antenna patterns. It can do both planar and non-planar geometries and has been validated for several antenna shapes [2]. This version of FEMA-PRISM also has the capability to define conducting pins and layers as well as holes in the antenna substrate. These features allow for the modeling of a finite thickness conductor, as well as substrate antenna patches.

FEMA-PRISM uses an internally generated 3-D distorted-prism volume mesh, grown from a user supplied 2-D surface mesh, to define the computational domain. Figure 2 shows how the volume mesh is grown from the user supplied surface mesh.

FEMA-PRISM ATTRIBUTES

Mesh Generation



(Perspective view) (Cross-sectional view)

Features

Analyzes antennas on doubly curved platforms

Anisotropic material coatings/substrates are accomodated

Grows volume mesh using distorted prisms. Given the surface mesh, volume mesh grown along surface normals

Built-in surface mesh generator for circular and rectangular patch antennas

Mesh turncated using isotropic/anisotropic artificial absorbers

Figure 2: FEMA-PRISM Mesh Generation, Absorbers, and Features.

The prismatic shape is a good compromise between the more complicated tetrahedron shape requiring a 3-D volume mesh input and rectangular bricks which do not model circular geometries well. Generating a 2-D surface mesh, while much simpler than generating a 3-D mesh, is not a trivial task, especially for broadband antennas or antennas containing small geometrical features. As an example consider Figure 3, showing the outline of a 7-element log-periodic slot antenna (LPSA). The slot widths are a function of frequency with the higher frequency (smaller) elements having narrower slot widths, usually about 1/100 wavelength. This is much smaller then the minimum 1/20 wavelength element size recommended for accurate FEM computations.

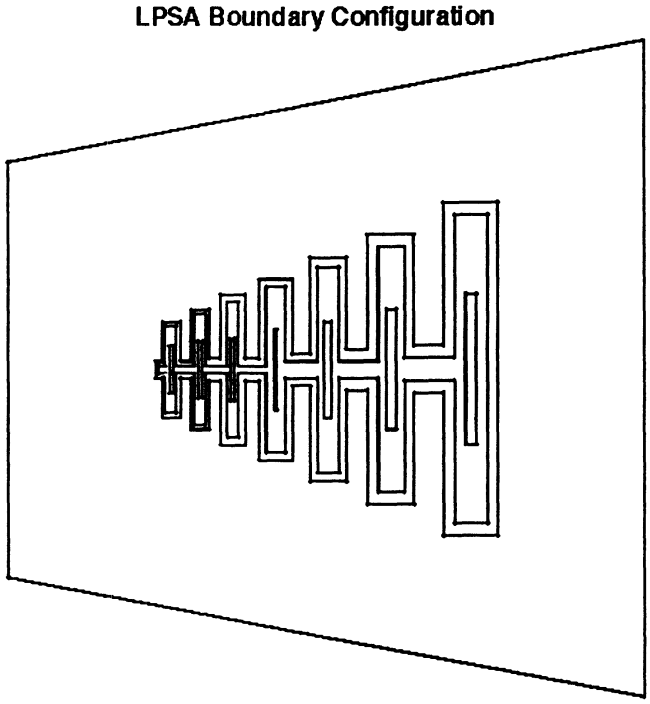


Figure 3: Log-Periodic Slot Antenna AutoCAD Boundary Description.

Element size in a uniform mesh will be dominated by this slot width and the resulting mesh for any practical problem will have too many unknowns to be analyzed on existing workstations in a reasonable time period. In FEM analysis a uniform mesh is not necessary and non-uniform meshing as shown in Figure 4 for a triangular cavity slot and also a 7-element log-periodic slot, will greatly reduce the number of unknowns. Algorithms to generate non-uniform meshes are complex and not easy to implement. Because of this a meshing package developed by J.R. Shewchuck at Carnegie-Mellon

University [3,4] and referred to as TRIANGLE was used to generate the FEMA-PRISM surface mesh. Non-Uniform meshing however, while decreasing the number of unknowns in the FEM system does not completely solve the problem. The FEM system requires that the mesh gradient, i.e., rate of change of element size, be gradual for convergence. This requirement will still keep the number of unknowns high and limit the antenna problems that can be solved by the code.

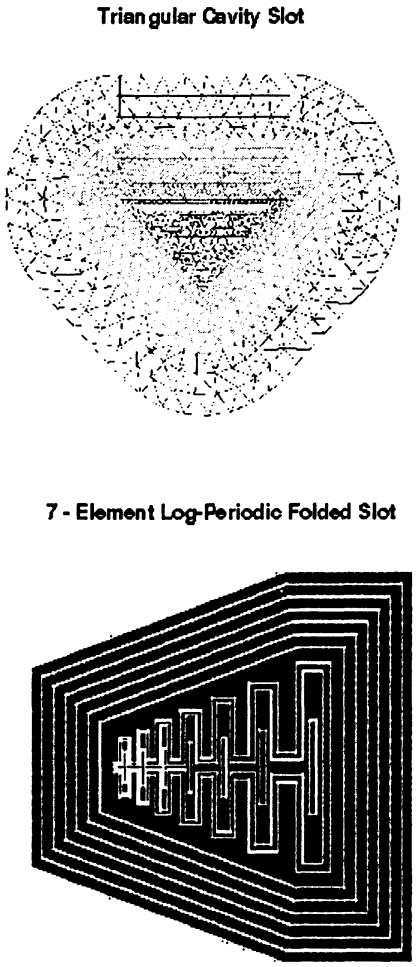


Figure 4: Non-Uniform Mesh - Triangular Cavity Antenna and Log-Periodic Folded Slot Antenna Array.

Artificial Absorbers (AA) are the mesh termination scheme used in FEMA-PRISM [2,5]. Figure 2 shows how the AA's are implemented. The FEM domain is surrounded by a Perfect Electric (PEC) conducting shell. The region between the antenna cavity and the shell consists of an inner air region and an outer absorbing region. These regions are defined according to criteria outlined in [5]. The total air/absorber thickness is usually 0.3 free-space wavelengths, 0.15 wavelengths air and 0.15 wavelengths absorber. The air/absorber regions are sampled (divided) into 1/20 wavelength thick regions, thus creating 3 layers of air and 3 of absorber. The permittivity and permeability of the absorber material are identical, usually 1-j2.7. The absorbing layers are designed to attenuate the incident wave at normal incidence, thus minimizing reflections and creating the appearance of infinite free space to the antenna. This material is physical unrealizable thus the name artificial absorber. The outer 6 layers of the meshes shown in Figure 4 are the air/absorber layers. FEMA-PRISM grows these surface layers in 3-D to create the side air/absorber layers shown in Figure 2. The code then caps the volume mesh by user-specified air/absorber layers, again shown in Figure 2. This air/absorber cap is usually is of the same parameters as the side air/absorber layers (6 layers, 1-j2.7, etc.). AA's, while creating a convenient way to terminate the FEM mesh, significantly increase the number of unknowns.

A serious limitation of FEMA-PRISM's capabilities to do practical size problems is the number of unknowns involved. A promising short-term solution will be to extend FEMA-PRISM's capabilities to a Boundary Integral (BI) termination scheme. This technique terminates the mesh at the cavity surface, thus eliminating the air/absorber elements and significantly reducing the number of unknowns allowing for larger problems to be solved. The BI option is currently being added to FEMA-PRISM and will be provided when available. Long term it is believed necessary to conduct research to determine the most efficient numerical scheme to solve these broad-band antenna problems.

Far-field patterns can be plotted using FEMA-PRISM post-processing routines [1]. Figure 5 shows an example of these outputs for the triangular slot antenna shown in Figure 4, along with surface near field plots. All plots were done in Matlab.

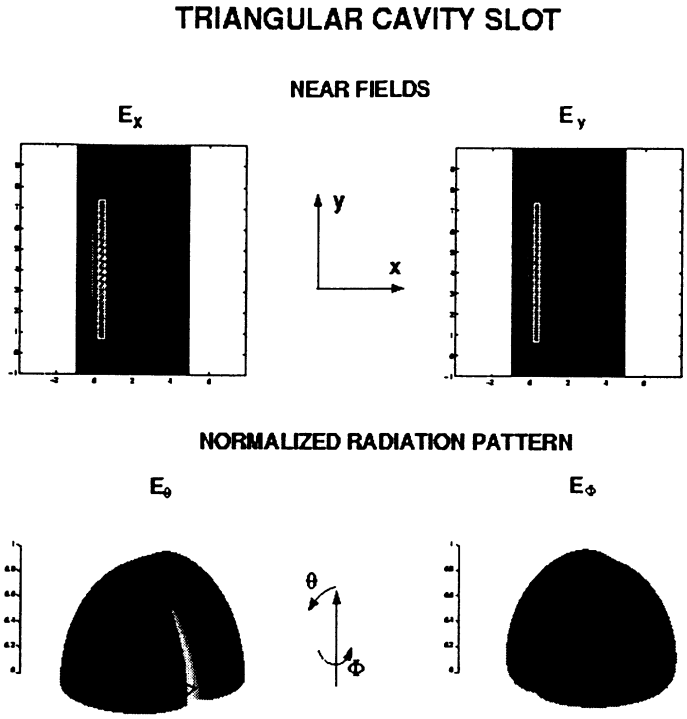


Figure 5: FEMA-PRISM Example Far-Field Patterns.

Tasks Performed

FEMA-PRISM Interface with AUTO-CAD (Task 1c)

For irregular LPDA and LPSA configurations, it is necessary that the structural details be specified using a CAD file provided, for example, by AutoCAD. Under this task U of M upgraded the I/O options of FEMA-PRISM to permit reading and decoding of AutoCAD files (in IGES or DXF formats) for generating the required surface meshes used for field calculations.

Note that [6] gives detailed information on the usage of FORTRAN routines AcadProc.f and TriProc.f outlined below. Reference [3] shows how to use the meshing package TRIANGLE, also discussed below, and also the input and output TRIANGLE file formats (.poly, .ele, .node).

The AutoCAD to FEMA-PRISM interface is a 4 step process. AutoCAD is used to generate a .dxf file containing line and boundary information. The .dxf file is then pre-processed using the FORTRAN routine AcadProc.f [6], to output a file with extension .poly for input to the TRIANGLE meshing package [3]. Each boundary in the AutoCAD file is identified by the layer it is on. This layer information is used by AcadProc to identify boundary regions for TRIANGLE. Note that unix-based AutoCAD v.13 was used in these examples. This should create no problems if AutoCad LT is used

since AcadProc simply looks for line information in the .dxf file. Figure 6 shows a sample .dxf file format with line and layer information and Figure 7 shows a 1 element folded slot .poly file with line, layer, and region identifying information. TRIANGLE uses the .poly file to generate a mesh based on user supplied parameters [3]. It outputs several file types including those with extensions .node and .ele, containing node and element information of the mesh. TriProc.f, another FORTRAN preprocessing routine [6], converts the TRIANGLE .ele and .node files to a FEMA-PRISM compatible mesh file called "surfmesh". TriProc allows the user to specify patch or slot (reverse conducting and dielectric material), cavity or microstrip [1], and Boundary Integral (BI, not yet implemented) or Artificial Absorber (AA) termination.

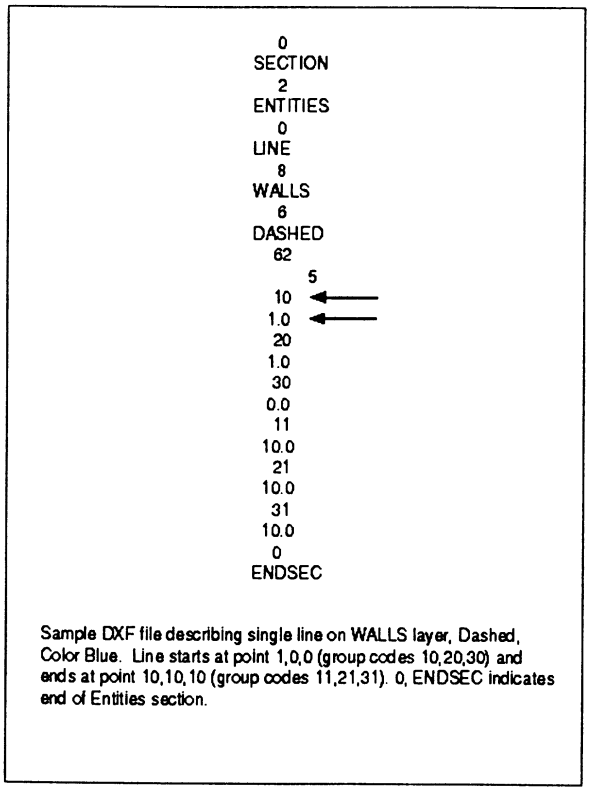


Figure 6: Sample AutoCAD .dxf file.

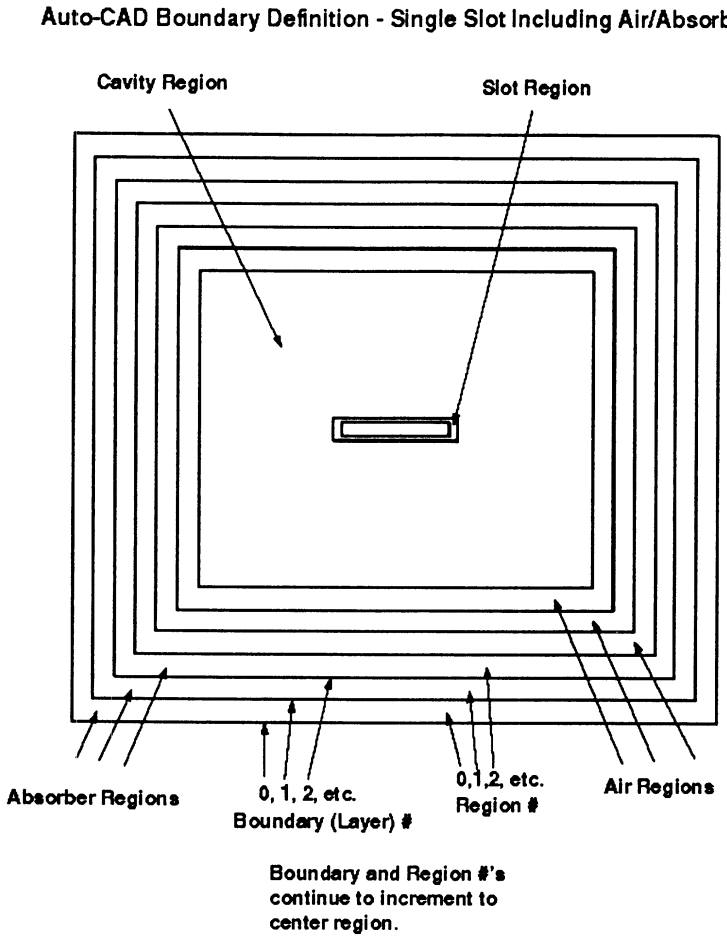


Figure 7: Single Slot AutoCAD Boundary/Region definition and Identifiers.

Output Format (Task 1d)

Currently, FEMA-PRISM outputs all data (radiation/scattering pattern, input impedance and so on) and antenna surface fields in a format for importing into MATLAB. Since MATLAB has well established and versatile plotting utilities, the data can then be cast in various presentation formats. Common spreadsheets such as Excel also allow for data plotting and an option will be included for outputting the data in a format suitable for porting into these PC data manipulation packages.

The post processing FORTRAN routine FarField.f [6], outputs pattern data in the

following column format:

Theta ----- Phi ----- Power (dB) Theta ----- Power (dB) Phi ----- Total Power (dB)

Mesh Generator of LPSA and LPDA (Task 2c)

Under this task U of M developed interfaces to mesh Log-Periodic dipole and slot arrays for use in FEMA-PRISM.

After examing many meshing packages for this application we chose a package referred to as TRIANGLE. TRIANGLE, written in C, is a 2-D mesh generation package using Delaunay Triangulation algorithms [3,4]. It was written by J.R. Shewchuk of the School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA (jrs@cs.cmu.edu). It is available for free on Netlib and at web location:

http://www.cs.cmu.edu/~quake/TRIANGLE.html

Note that a viewing package, Showme, written for unix workstations is also available at this website. Showme allows the user to view TRIANGLE input and output .node, .poly, and .ele files.

Details of file formats and usage are contained in [3]. Under this task TRIANGLE was interfaced with FEMA-PRISM. As previously stated, two FORTRAN preprocessing routines, AcadProc.f and TriProc.f were written to interface AutoCAD to TRIANGLE and then TRIANGLE to FEMA-PRISM. While many options are available in TRIANGLE for simplicity only a few were used to generate meshes for our applications. The options used control minimum element area, and minimum TRIANGLE interior angle. Options were also used which specify that the input file from AcadProc is a .poly (boundary line) file and to use the region coordinate locations and identifiers at the end of the .poly file to identify bounded regions. This is done to simplify sorting of various element types (conducting, cavity, absorbers, etc.) for the FEMA-PRISM input file "surfmesh". TriProc also contains an option to reverse surface conducting/non-conducting elements. Figure 8 shows a single element folded slot along with dimensions and a sample TRIANGLE mesh.

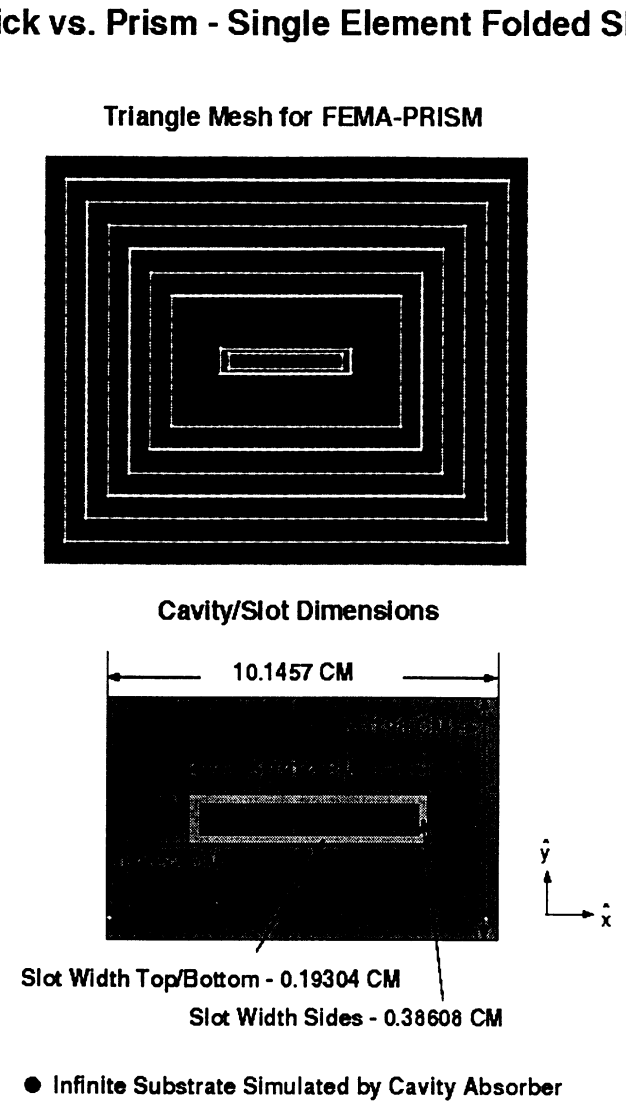


Figure 8: Dimensions and TRIANGLE Mesh - Folded Slot.

Feed locations, curvature of the platform, cavity depth and material layers are specified in the FEMA-PRISM input file [1].

Feed Modeling (Task 2d)

This version of FEMA-PRISM employs probe feed modeling. The probes can be placed along any horizontal or vertical edge in the antenna mesh.

Validation (Task 4b)

Two examples were chosen for validation, the single element folded slot shown in Figures 7 and 8 and the 7-element Log-Periodic Folded Slot Antenna (LPFSA) shown at the bottom of Figure 4. All runs were on an SGI Indigo2 with 200MHz processor and

64M ram. Far-field patterns for both problems are shown in Figure 10. A Matlab "quiver" vector field plot of the single slot is shown in Figure 9.

The single folded slot mesh containing 54393 unknowns was run at 2.075 GHz, the approximate resonant frequency of the slot. It converged in 10356 iterations and took approximately 5.54 hours to run. The average time per iteration was 1.91 seconds. E-Cut is across the narrow cross section of the slot with H-Cut perpendicular to this. The is an anomaly in the E-Cut cross-pol just above 36.0 degrees which cannot be explained at this time.

The LPSFA mesh contained 47984 unknowns and was run at 3.05 GHz, or approximately the resonant frequency of the center of 7 elements. It converged in 8543 iterations and took approximately 3.5 hours with an average time per iteration of 1.46

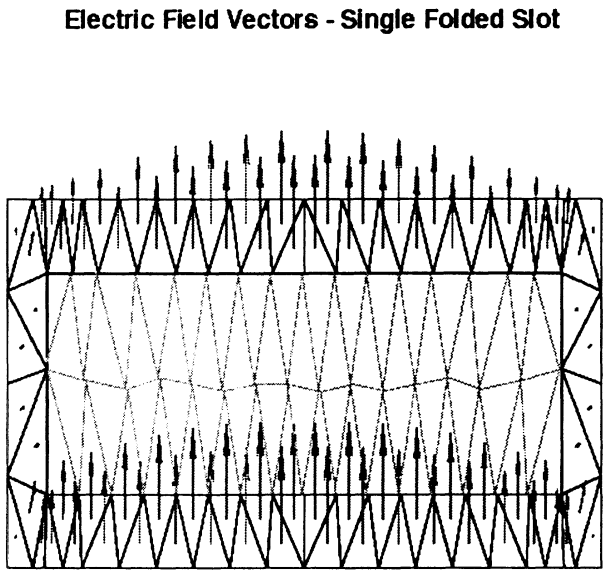


Figure 9: Electric Field Vector Matlab "Quiver Plot" - Single Slot.

seconds. Note that the mesh size gradient was not optimal in order to keep the number of unknowns at a reasonable level. A better gradient would have produced an un-manageable number of unknowns. E-Cut for the LPFSA was perpendicular to the folded slots, with H-Cut parallel to them.

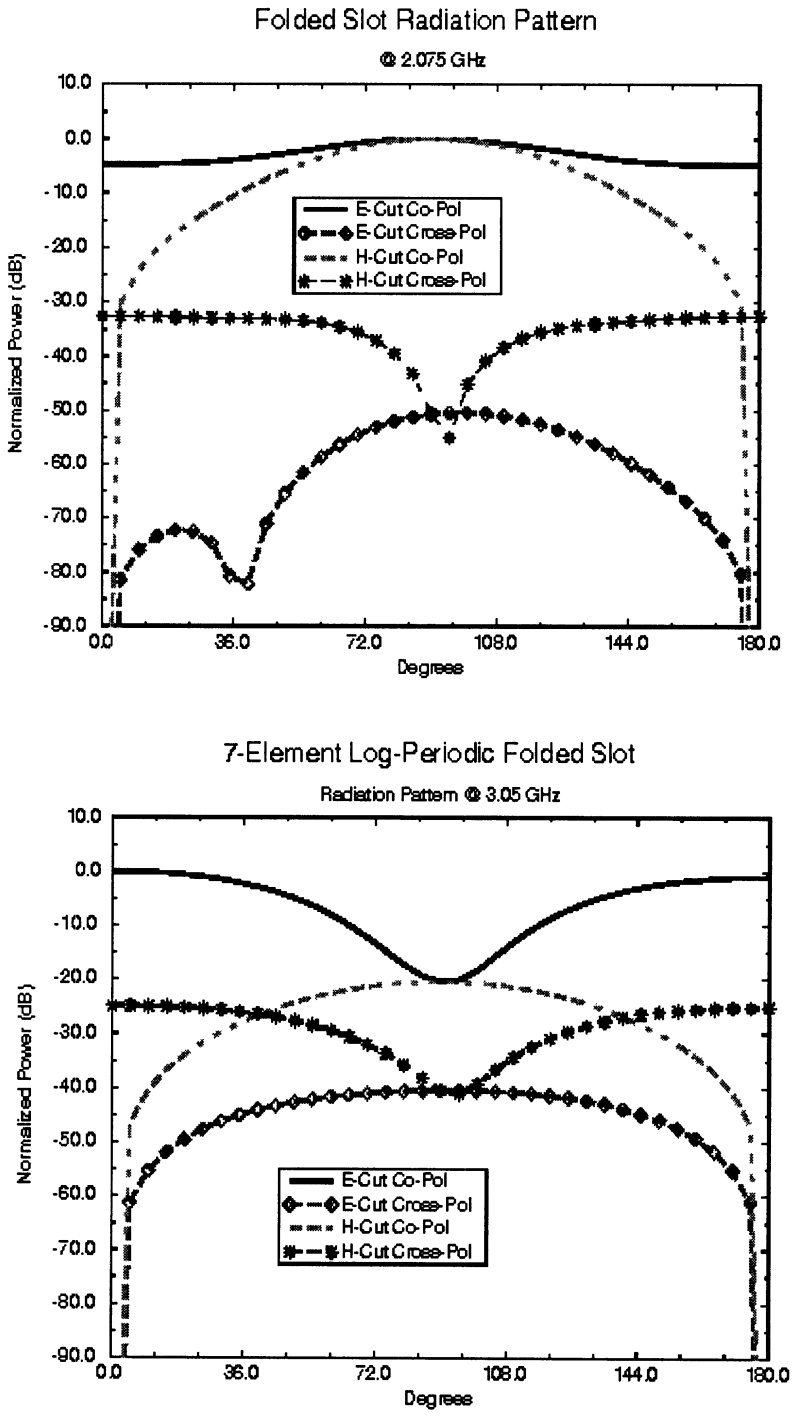


Figure 10: Far field patterns - Folded Slot and LPFSA.

Bibliography

[1] T. Ozdemir and J. L. Volakis, "Users manual for FEMA-PRISM, "Univ of Michigan Radiation Lab. Techn. Report 031307-6-T, March 1996. 15pp.

[2] T. Ozdemir and J. L. Volakis, "Triangular prisms for edge-based vector finite element analysis of conformal antennas," Univ of Michigan Radiation Lab., October 1996.

[3] J. R. Shewchuk, "TRIANGLE, A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator. v1.3," Carnegie Mellon University.

[4] J. R. Shewchuk, "TRIANGLE: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator," School of Computer Science, Carnegie Mellon University.

[5] T. Ozdemir and J. L. Volakis, "A comparative study of an absorbing boundary condition and an artificial absorber for truncating finite element meshes," *Radio Sci.*, Vol. 29, No. 5, pp. 1255-1263, Sept.-Oct. 1994.

[6] M. D. Casciato and J. L. Volakis, "AutoCAD FEMA-PRISM Interface Manual," University of Michigan Radiation Lab., December 1996.

Triangular prisms for edge-based vector finite element analysis of conformal antennas[1]

T. Özdemir and J. L. Volakis

Radiation Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, Michigan 48109-2122

October 31, 1996

Abstract

This paper deals with the derivation of the edge-based shape functions for the distorted triangular prism and their applications for the analysis of doubly curved conformal antennas in the context of the finite element method (FEM). Although the tetrahedron is often the element of choice for volume tessellation, mesh generation using tetrahedra is cumbersome and CPU intensive. On the other hand, the distorted triangular prism allows for meshes which are unstructured in two dimensions and structured in the third dimension. This leads to substantial simplifications in the meshing algorithm and many conformal printed antenna and microwave circuit geometries can be easily tessellated using such a mesh. The new edge-based shape functions are first validated by computing the eigenvalues of three different cavities (rectangular, cylindrical and pie-shell). We then proceed with their application to computing the input impedance of conformal patch antennas on planar, spherical, conical and other doubly curved (ogival) platforms, where the FEM mesh is terminated using an artifical absorber applied conformal to the platform. Use of artificial absorbers for mesh termination avoids introduction of Green's functions and, in contrast to absorbing boundary conditions, a knowledge of the principal radii of curvature of the closure's boundary is not required.

[1]This work was supported in part by the U.S. Air Force Rome Laboratory and the NASA Langley Research Center.

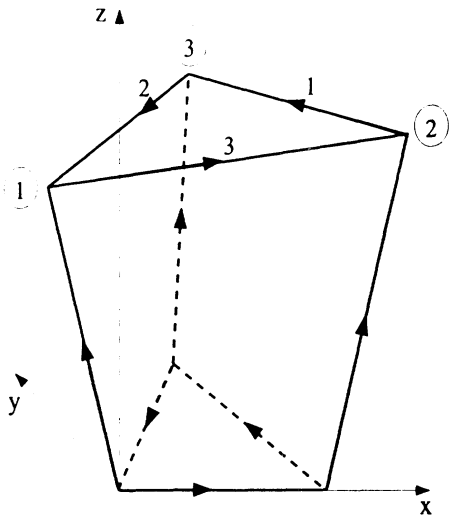


Figure 1: The distorted triangular prism shown with the direstions of the edge vectors

cation, the finite element mesh is terminated conformal to the platform's surface using an artificial absorber rather than an absorbing boundary condition (ABC) [5]. The employed conformal mesh termination is easily implemented by using prisms, and in contrast to the ABCs, the artificial absorber does not require *a priori* knowledge of the closure's radii of curvature or the wave's propagation characteristics. The utility and versatility of the proposed finite element method (FEM) formulation is demonstrated by considering the analysis of several printed antennas on different platforms. Specifically, we include input impedance computations for rectangular and circular patches on planar, spherical and conical surfaces. The radiation from a patch on an ogive is also considered.

2 Vector edge-based basis functions

Consider the distorted prism shown in Figure 3. The prism's top and bottom triangular faces are not necessarily parallel to each other and the three vertical arms are not perpendicular to the triangular faces. The first step toward specifying a set of shape functions for the prism is the identification of a unique cross-section containing the observation point (x, y, z) (see Figure 3). This is done by introducing a parametric representation for the nodes (x_i, y_i, z_i) of this cross-section as illustrated in Figure 4. These parametric

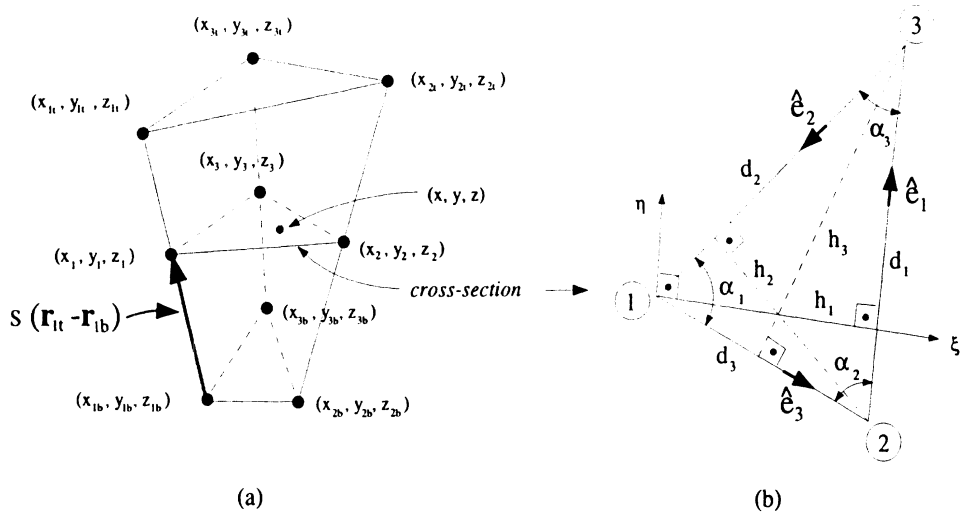


Figure 4: (a) Nodal coordinates, (b) triangular cross-section with the local coordinates ξ and η.

representations are given in the Appendix. They involve the parameter s such that (x_i, y_i, z_i) reduce to the nodes of the bottom triangle when $s = 0$ and those of the top triangle when $s = 1$. Given a point (x, y, z) interior to the prism, a unique value for s can be computed as discussed in Appendix A.

Having specified the cross-section through the point (x, y, z), we next proceed with the derivation of the basis functions. We chose to represent the field variation across the triangular cross-section using the Whitney-1 form [6]. A simple linear variation will be assumed along the length of the prism. Specifically, the vector basis functions for the top triangle edges can be expressed as

$$\begin{aligned} \mathbf{N}_1 &= d_1 \left(L_2 \nabla L_3 - L_3 \nabla L_2 \right) s \\ \mathbf{N}_2 &= d_2 \left(L_3 \nabla L_1 - L_1 \nabla L_3 \right) s \\ \mathbf{N}_3 &= d_3 \left(L_1 \nabla L_2 - L_2 \nabla L_1 \right) s \end{aligned}$$

and correspondingly those for the bottom triangle edges will be

$$\begin{aligned} \mathbf{M}_1 &= d_1 \left(L_2 \nabla L_3 - L_3 \nabla L_2 \right) (1 - s) \\ \mathbf{M}_2 &= d_2 \left(L_3 \nabla L_1 - L_1 \nabla L_3 \right) (1 - s) \\ \mathbf{M}_3 &= d_3 \left(L_1 \nabla L_2 - L_2 \nabla L_1 \right) (1 - s). \end{aligned}$$

The subscripts in these expressions identify the edge numbers as shown in Figure 1 and the distance parameters d_i are equal to the side lengths of the

chose to express these by the representations (linear over the cross-section)

$$\begin{aligned} \mathbf{K}_1(\xi,\eta) &= \hat{v}(\xi,\eta)\, L_1(\xi,\eta) \\ \mathbf{K}_2(\xi,\eta) &= \hat{v}(\xi,\eta)\, L_2(\xi,\eta) \\ \mathbf{K}_3(\xi,\eta) &= \hat{v}(\xi,\eta)\, L_3(\xi,\eta). \end{aligned} \tag{2}$$

As before, L_i are the node-based shape functions defined in (1) and a pictorial description of $\mathbf{K}_1$ is found in Figure 5(b). Of particular importance in (2) is the unit vector $\hat{v}(\xi,\eta)$. It is a linear weighting of the unit vectors $\hat{v}_1$, $\hat{v}_2$ and $\hat{v}_3$ associated with the vertical arms (see Figure 5(c)), and is given by

$$\hat{v}(\xi,\eta) = \frac{\sum_{i=1}^{3} L_i(\xi,\eta)\, \hat{v}_i}{\| \sum_{i=1}^{3} L_i(\xi,\eta)\, \hat{v}_i \|}. \tag{3}$$

This particular choice of $\hat{v}$ is oriented parallel to the side faces of the prism when evaluated on those surfaces and minimizes tangential field discontinuity across the side faces. Another choice is

$$\hat{v}(\xi,\eta) = \nabla s$$

which is always oriented normal to the triangular cross-sections of the prism and ensures tangential field continuity across the top and bottom triangular faces. Both choices are equally useful. However, (3) is more computationally efficient and has been used in generating the results presented in sections 3 and 4.

The shape functions derived above for the distorted prism reduce to the right prism shape functions presented by Nédélec [8]. However, it should be pointed out that the above shape functions do not guarantee tangential field continuity across the faces of neighboring prisms. The possible discontinuity is primarily an issue for highly distorted elements and is caused by the fact that the horizontal vector basis functions (**N** and **M**) may have small non-vanishing tangential conponents across the vertical faces of the prism (and vice versa for the vertical basis functions). The expressions given in Appendix B neglect contributions due to the tangential discontinuities across the inter-element boundaries. These contributions were ignored because the discontinuity depends on the distortion of the prism and in practice the elements are marginally distorted. For general applications, our basis functions can be safely employed (within the accuracy of the finite element method) as

$$\sum_{j=1}^{3} E_{jK}[NKC_{ij} - k_o^2 NKD_{ij}] = 0 \qquad (8)$$

$$\sum_{j=1}^{3} E_{jN}[MNC_{ij} - k_o^2 MND_{ij}] + \sum_{j=1}^{3} E_{jM}[MMC_{ij} - k_o^2 MMD_{ij}] + \sum_{j=1}^{3} E_{jK}[MKC_{ij} - k_o^2 MKD_{ij}] = 0 \qquad (9)$$

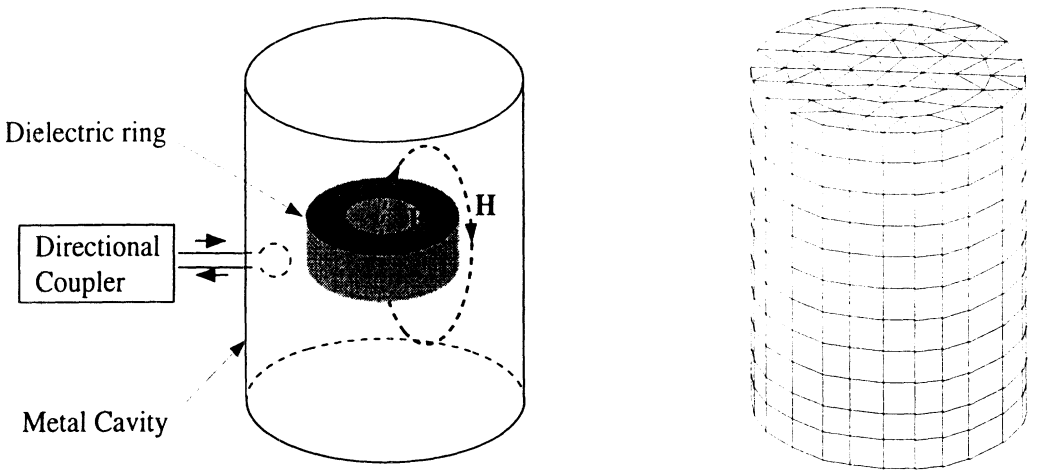
$$\sum_{j=1}^{3} E_{jN}[KNC_{ij} - k_o^2 KND_{ij}] + \sum_{j=1}^{3} E_{jM}[KMC_{ij} - k_o^2 KMD_{ij}] + \sum_{j=1}^{3} E_{jK}[KKC_{ij} - k_o^2 KKD_{ij}] = 0, \ i = 1,2,3. \qquad (10)$$

where the quantities NNC_{ij}, NND_{ij}, etc. are integrals (over the volume of the prism) involving vector basis functions. Explicit expressions for the integrals are given in Appendix B.

Upon assembly of (8)-(10) and boundary condition enforcement, we obtain the generalized eigenvalue system

$$[A]\{x\} = k_o^2[B]\{x\}$$

in which $\lambda = k_o^2$ represent the eigenvalues of the problem. The matrices $[A]$ and $[B]$ are real, symmetric and sparse ($[B]$ is also positive definite).



Cavity dimensions:
Height = 2.2cm
Inner diameter = 7.5cm
Outer diameter = 11cm
$\varepsilon_r = 13.5$

Dielectric ring info:
Height = 13.84cm
Diameter = 15.24 cm

Results:
Measured = 1282 MHz
Computed = 1257 MHz
(Error = 2 %)

Figure 7: Dielectric ring resonator, (a) geometry, (b) mesh.

Example 3: *Pie-shell Cavity*

The third example is a pie-shell sector as shown in Figure 8(a) modeled using distorted prisms. It is obtained by bending the rectangular cavity considered earlier. The computed and exact eigenvalues for the first five dominant modes are given in Figure 8(b), and these testify to the accuracy of the distorted prisms in modeling curved geometries. The number of degrees of freedom used for this computation was 382.

4 Application to antennas

4.1 Finite element-artificial absorber method

Prisms facilitate the modeling of conformal antennas since the presence of curvature does not complicate the mesh generation. As is the case with all PDE methods, the mesh must be terminated using a boundary integral, some approximate boundary condition or an absorber. When a Green's function is available, the mesh termination can be achieved using the boundary integral method right on the antenna surface yielding exceptionally accurate results. However, there are two problems with the boundary integral mesh trunca-

tion techniques. First, the Green's function is only available for canonical geometries, i.e., planar [11], cylindrical [12] and spherical [13]. Second, the presence of material overlays can prohibitively complicate the derivation and evaluation of the Green's function making it very difficult to implement the boundary integral termination method. Thus, for modeling non-canonical geometries with possible material overlays, it is preferable to avoid use of the Green's function. Instead, an artificial absorber will be used for truncating the mesh as illustrated in Figure 9. The resulting implementation will be referred to as the finite element-artificial absorber (FE-AA) method and has several attractive computational features. Among them are fast convergence [5] and a capability to provide accurate input impedance computations. Accurate radiation pattern data can also be obtained when the rest of the platform has little or no effect. In its simplest form, this is done by ignoring the extent of the platform and integrating the aperture fields using the free-space Green's function.

We next discuss the artificial absorber and then proceed with the application of the FE-AA method for the analysis of a variety of patch antennas on different platforms, some of which are presented here for the first time.

4.2 Design of the artificial absorber

The absorber consists of a lossy dielectric layer backed by a metal. Metal backing enables us to terminate the mesh while the lossy dielectric lining traps the incoming wave and absorbs it, thereby forming a trasparent boundary. The absorber material parameters are chosen to completely eliminate wave reflections at normal incidence. Away from normal, the absorber reflectivity increases monotonically but can be minimized by proper selection of the absorber thickness and material parameters. Clearly, a thicker absorber has a better performance but carries additional computational burden. In addition, higher attenuation is achieved by making the layer more lossy. However, in this case more samples are required along the thickness of the layer to accurately model the field's amplitude decay. For a given thickness and sampling, an optimization can however be carried out. Such a study was performed in [5] and it showed that a minimization of the reflectivity for a 3-layer, $0.15\lambda_o$ (free-space) thick metal-backed absorber yields the constitutive parameters of $\epsilon_r = \mu_r = 1 - j2.7$ (see Figure 9). Below, we make use of this absorbing layer for mesh truncation. A layer based on the recently

For this example, the computational domain was discretized using 7,440 prisms resulting in 12,523 degrees of freedom. One frequency run took 3.5 minutes on an HP 715/64 workstation.

Microstrip circular patch on sphere

Figure 11a shows a microstrip circular patch placed on a sphere. This is an example which clearly shows the advantage of the finite element-artificial absorber technique. Once the antenna aperture is triangulated, the volume mesh is grown along surface normals (using prisms) and terminated with the artificial absorber. Figure 11b shows the comparison with the moment method [13] and the effect of sphere radius on the resonance behavior. Resonance frequencies predicted by this method and the moment method are within 2% of each other. Although negligible, the difference must be evaluated in view of the fact that the reference data is based on the approximation that only the dominant TM_{11} mode exists under the patch. Such assumptions are absent in the finite element analysis. Clearly, the value of the input resistance is a strong function of the employed feed model and therefore, it is not surprising to see differences in the levels of the resistance as computed by the finite element method and the single mode moment method solution. Figure 11b also shows the resonance behavior for different sphere radii, and it is observed that the resonance frequency decreases with increasing radius. This trend might be explained by noting that the shortest distance between the radiating edges of the patch is greater for a larger sphere radius. The patch radius (measured on the spherical surface) has been kept constant for this calculation.

Figures 12c shows the input resistance variation with the excitation frequency for different substrate/superstrate material constants and thicknesses. Similarly to the planar patch, we observe the downward shift in the resonance frequency caused by the increasing relative permittivity of the substrate. Notice also that the shift is only half as much when the superstrate is present even though the increase in the refractive index of the superstrate layer is 1.6 times higher than that of the substrate. This is because the field is much stronger under the patch than above it. As expected, Figure 11c shows that the loss in the substrate material has no effect on the resonance frequency but it lowers the overall level of the input impedance. The typical bandwidth increase with increasing substrate thickness is also observed.

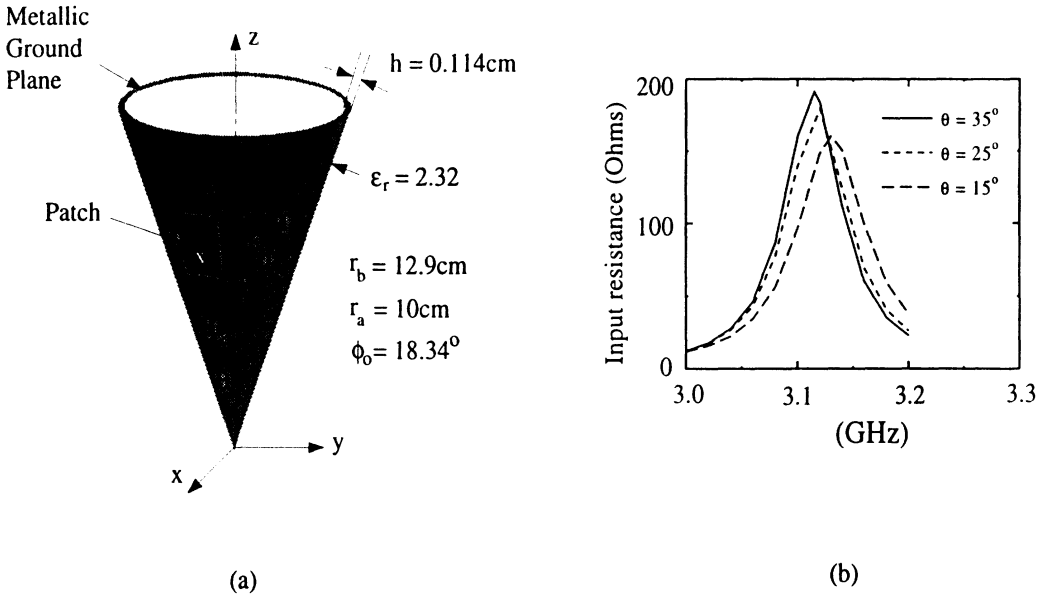


Figure 12: Microstrip sectoral patch on cone: (a) Configuration, (b) change in the resonance behavior as a function of the cone angle θ.

For this example, the computation region has been discretized using 6,048 prisms resulting in 9,997 degrees of freedom. One frequency run took 4.5 minutes on an HP 715/64 workstation.

Microstrip sectoral patch on cone

This is an example which illustrates the effectivenenss of the new technique to model antennas on doubly curved platforms with varying radii of curvature. To our knowledge, this is the first analysis of patches on such a platform. Figure 12a displays a sectoral microstrip patch printed on a conical ground plane and Figure 12b shows the antenna resonance behavior as a function of the cone angle θ. We clearly observe that the resonance frequency drops with the cone angle for the same patch dimensions. It should be also remarked that the computed resonance frequency is within 3.2 percent of that predicted by the approximate cavity model, and this is reasonable.

In generating the results given in Figure 12, the computational domain was discretized using 2,358 prisms resulting in 3,790 degrees of freedom. One frequency run took 5.5 minutes on an HP 715/64 workstation.

puted θ-polarized radiation as compared to the measurement [15]. Clearly, the agreement is very good for this polarization. However, predicting the ϕ-polarized radiation (not shown) requires modeling of the entire ogive as this particular polarization has a vertical surface field component which is known to cause diffraction from the ogive's tips. A way to account for such secondary diffractions is to interface the FE-AA method with a high frequency technique and an encouraging study in this direction has been carried out in [16].

5 Conclusions

A new finite element technique was introduced for the analysis of printed antennas recessed in platforms of non-canonical shape. The distorted prism was introduced as the volume discretization element, and corresponding edge-based shape functions were derived and tested for eigenvalue computations.

A major part of the paper though was devoted to characterizations of printed antennas using the new technique. Use of prismatic elements was found very attractive for this application and was essential in simplifying the modeling of antennas on doubly curved platforms. An artificial absorber was used to terminate the mesh conformal to the platform thereby minimizing the size of the problem while preserving the sparsity of the finite element matrix. Use of the absorber also avoids difficulties associated with the conformal application of the classical ABCs. Antennas on spherical, conical and ogival platforms were considered without using a Green's function and therefore, the superstrate/substrate metarials can be readily accounted for.

A limitation of the technique is that it models the immediate neighborhood of the antenna, and therefore ignores the details associated with the rest of the platform and possible substructures around the radiator. However, the proposed method was shown to be a good approach for predicting the resonance behavior of antennas, and could evolve as an important tool for designing conformal antennas on doubly curved platforms.

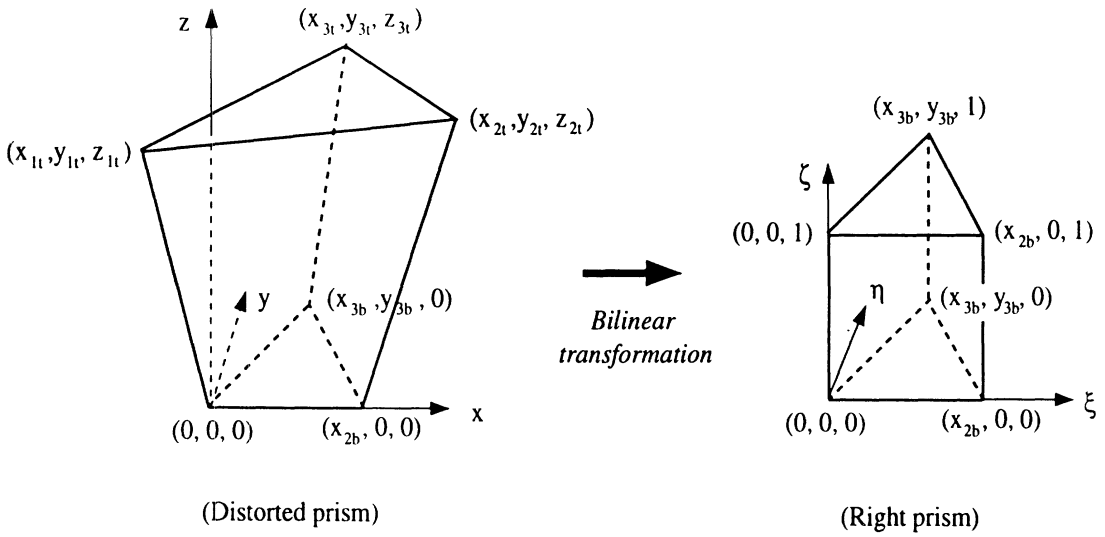
Appendix B

In this Appendix, we show how to compute the following quantities:

$$
\begin{aligned}
NNC_{i\ell} &= \int\int\int_V (\nabla \times \mathbf{N}_i) \cdot (\nabla \times \mathbf{N}_\ell) dV \\
NMC_{i\ell} &= \int\int\int_V (\nabla \times \mathbf{N}_i) \cdot (\nabla \times \mathbf{M}_\ell) dV \\
NKC_{i\ell} &= \int\int\int_V (\nabla \times \mathbf{N}_i) \cdot (\nabla \times \mathbf{K}_\ell) dV \\
MMC_{i\ell} &= \int\int\int_V (\nabla \times \mathbf{M}_i) \cdot (\nabla \times \mathbf{M}_\ell) dV \\
MKC_{i\ell} &= \int\int\int_V (\nabla \times \mathbf{M}_i) \cdot (\nabla \times \mathbf{K}_\ell) dV \\
KKC_{i\ell} &= \int\int\int_V (\nabla \times \mathbf{K}_i) \cdot (\nabla \times \mathbf{K}_\ell) dV \\
NND_{i\ell} &= \int\int\int_V \mathbf{N}_i \cdot \mathbf{N}_\ell dV \\
NMD_{i\ell} &= \int\int\int_V \mathbf{N}_i \cdot \mathbf{M}_\ell dV \\
NKD_{i\ell} &= \int\int\int_V \mathbf{N}_i \cdot \mathbf{K}_\ell dV \\
MMD_{i\ell} &= \int\int\int_V \mathbf{M}_i \cdot \mathbf{M}_\ell dV \\
MKD_{i\ell} &= \int\int\int_V \mathbf{M}_i \cdot \mathbf{K}_\ell dV \\
KKD_{i\ell} &= \int\int\int_V \mathbf{K}_i \cdot \mathbf{K}_\ell dV.
\end{aligned}
$$

where the integrals are over the volume of the prism and they must be evaluated numerically. However, numerical integration over the distorted prism volume is cumbersome and therefore, the distorted prism is first mapped onto a right prism as shown in Figure B1. The integration over the new volume is then carried out by using the five point Gaussian formula. The relationship between the two integrations is given by

$$
\int\int\int_{V_{xyz}} f(x,y,z)\, dxdydz = \int\int\int_{V_{\xi\eta\zeta}} f(x,y,z)\, det[J]\, d\xi d\eta d\zeta
$$



(Distorted prism) (Right prism)

Figure 14: Mapping distorted prism onto a right prism

$$\frac{\cos\beta_{jn}}{h_j h_n}\chi_{km} + \frac{2}{3}\frac{c^2 h_1 d_1}{h_j h_k h_m h_n} sin\beta_{jk}\, sin\beta_{mn}\,)$$

$$ENMC_{i\ell} = \frac{d_i d_\ell}{c}\,(-\frac{\cos\beta_{kn}}{h_k h_n}\chi_{jm} - \frac{\cos\beta_{jm}}{h_j h_m}\chi_{kn} + \frac{\cos\beta_{km}}{h_k h_m}\chi_{jn} +$$
$$\frac{\cos\beta_{jn}}{h_j h_n}\chi_{km} + \frac{1}{3}\frac{c^2 h_1 d_1}{h_j h_k h_m h_n} sin\beta_{jk}\, sin\beta_{mn}\,)$$

$$ENKC_{i\ell} = \frac{h_1 d_1}{6} d_i\,(\frac{\cos\beta_{j\ell}}{h_j h_\ell} - \frac{\cos\beta_{k\ell}}{h_k h_\ell})$$

$$EMMC_{i\ell} = ENNC_{i\ell}$$

$$EMKC_{i\ell} = -ENKC_{i\ell}$$

$$EKKC_{i\ell} = c\frac{h_1 d_1}{2}\frac{\cos\beta_{i\ell}}{h_i h_l}$$

References

[1] Z. Sacks, S. Mohan, N. Buris, and J. F. Lee, "A prism finite element time domain method with automatic mesh generation for solving microwave cavities," *IEEE APS Int. Symp. Digest*, Vol. 3, pp. 2084-87, Seattle, Washington, June 19-24, 1994.

[2] J. C. Nédélec, "Mixed finite elements in R^3," *Numer. Math.*,Vol. 35, pp. 315-341, 1980.

[3] A. Bossavit, "A rationale for 'edge-elements' in 3-D fields computations," *IEEE Trans. Magn.*, Vol. 24, No. 1, pp. 74-79, January 1988.

[4] J. P. Webb, "Edge elements and what they can do for you," *IEEE Trans. Magn.*, vol. 29, no. 2, pp. 1460-1465, March 1993.

[5] T. Özdemir and J. L. Volakis, "A comparative study of an absorbing boundary condition and an artificial absorber for truncating finite element meshes," *Radio Sci.*, Vol. 29, No. 5, pp. 1255-1263, Sept.-Oct. 1994.

[6] A. Bossavit, "Whitney forms: a class of finite elements for three-dimensional computations in electromagnetism," *IEE Proc. Part A*, Vol. 135, No. 8, November 1988.

[7] O. C. Zienkiewics and R. L. Taylor, *The Finite Element Method*, 4th ed., Vol. 1, McGraw-Hill, New York, 1989.

[8] J. C. Nédélec, "A New Family of Mixed Finite Elements in R^3," *Numer. Math.*, No. 50, pp. 57-81, 1986.

[9] A. Chatterjee, J. M. Jin and J. L. Volakis, "Computation of Cavity Resonances Using Edge-Based Finite Elements," *IEEE Trans. Microwave Theory Tech.*, Vol. 40, No. 11, pp. 2106-2108, Nov. 1992.

[10] J. B. Muldavin, A. D. Krisch and M. Skalsey, Personal communication, University of Michigan, 1995.

[11] J. Gong, J. L. Volakis, A. C, Woo, and H. T. G. Wang, "A hybrid finite element-boundary integral method for the analysis of cavity-backed

User's Manual for FEMA-PRISM (Version 1)

Tayfun Özdemir
John L. Volakis

Radiation Laboratory
Department of Electrical Engineering and Computer Science
1301 Beal Ave.
University of Michigan
Ann Arbor, MI 48109-2122
TEL: (313) 764-0502, (313) 747-1797
FAX: (313) 747-2106
E-MAIL: tayfun@umich.edu, volakis@umich.edu
HOME PAGE: http://www-personal.engin.umich.edu/~volakis/

CONTENTS

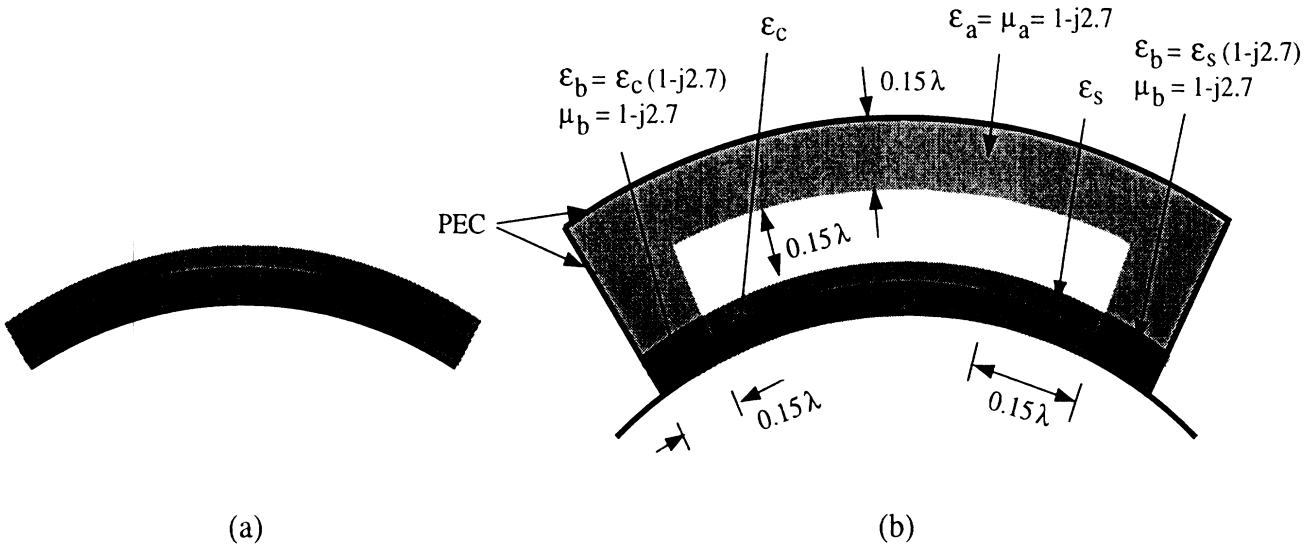


Figure 1: Antenna Modeling

1 General code description

FEMA-PRISM is written in Fortran 77 computer language and has been verified to run on Unix platforms for HP, Sun, and SGI workstations. It is currently a serial code with potential for parallellization. It can be run with limited memory allocation at the expense of speed.

FEMA-PRISM is used to analyze printed antennas on doubly curved surfaces. It employs the finite element method (FEM) in conjunction with artificial absorbers to truncate the mesh (see Figure 1). The resulting FEM system of linear equations are solved interatively using BiCG technique. For details of the analysis and the theory of the formulation, the user is referred to [1] and [2].

2 Types of antennas that can be modeled

Microstrip as well as cavity-backed antennas with or without coatings can be modeled (see Figure 1). Currently, only probe feed can be specified at any arbitrary location.

3 Specifying antenna geometry

Because the antennas conform to the platform, only a surface mesh is needed. A built-in surface mesh generator exists for rectangular and circular patch antennas (see Section 11). Once the surface mesh is created, it can be viewed very easily using MatLab tools (see Section 12). The volume mesh is simply grown along the surface normal and the distorted prism is the building block of the resulting mesh [1].

4 Input Files

MainInput : Contains information about the geometry and other input data
SurfMesh : Contains the surface mesh data (specifying antenna surface detail)

5 Output Files

Imp : Stores the input impedance
EqvCur : Equivalent magnetic current over the surface of the antenna
EdgeUnk : Complex amplitudes of edge unknowns
MeshDsply: Contains surface mesh data for plotting using MatLab
ElmMat : Contains element matrices for each prism in the mesh. It is only computed when a new mesh is generated and can be reused as long as the mesh does not change. Whether it is generated or read in is controlled by an entry in the input file *MainInput.*

6 Running the code

Compile the code by typing

f77 FEMA-PRISM.f -o *executable-filename*

To run the code, type in *executable-filename.* The code will first read the integer entry of the first row of the input file *MainInput*, which will tell the code whether the user supplied surface is alreay in the file *SurfMesh* or the surfmesh is to be generated by the code according to the second row of *MainInput.* If the user supplies the surface mesh, the code proceeds to read in the contents of the file *SurfMesh* and carries out the analysis according to the subsequent rows of information in the file *MainInput.* If the mesh is to be generated by the code's built-in mesh generator, the code also reads in the second row of the file *MainInput*, stores the surface mesh data in the file *SurfMesh* and terminates. The code has to be rerun with user-supplied mesh option. For more detailed explanation, see the following three sections.

7 Input files *MainInput* and *SurfMesh*

MainInput contains information describing the antenna geometry, substrate/superstrate materials, frequency of operation, etc. Figure 2 shows the data format. As shown there, the first row tells the code whether the user supplies the surface mesh or whether the mesh will be generated by the built-in mesh generator. The second row has the antenna geometry info if built-in mesh generator is used. Otherwise this information is skipped. Starting with the third row (second row if user-supplied surface mesh

option is chosen), the rest of the information is concerned with the geometry and the run. **All length quantities are in units of Centimeter, frequencies are in GHz, currents are in Amperes, electric filed is in Volts/cm, impedance is in Ohms and material parameters are always relative quantities with respect to those of the free-space.**

SurfMesh contains surface mesh data and must be ready prior to running the code if the user-supplied mesh option is chosen. It can be created for rectangular and circular patches by the code itself. The first row of the file *SurfMesh* contains a series of numbers specifying how many triangles and nodes are contained in the surface mesh, the number of triangles within the absorbing layer, etc.

8 Running with user-supplied surface mesh option

In this operation mode, the surface mesh data has to be ready in the file called *SurfMesh*. Figure 2 shows the general set up of the *MainInput* along with a description for each entry. Each filled circle indicates a row. All rows are read by the code with free format. Letters R,I, or C refer to a real, integer or a complex number entry. All entries on the same row must be seperated at least by a single space.

9 Running with built-in surface mesh generator

In this operation mode, the code has to be run twice. In the first run, the first row of the file *MainInput* has the entry value "2" or "3". The second row provides the information the built-in mesh generator needs to generate the mesh. The code stores the mesh data in *SurfMesh* and terminates. The code must then be rerun with the user-supplied mesh option.

Note that the built-in surface mesh generator generates planar surface meshes (located in the plane $z = 0$).

Caution: **In creating a surface mesh for a rectangular patch, if the patch is cavity-backed, care must be taken to leave at least one cell between the cavity wall and the air-absorber interface. If the patch is a microstrip, at least two cells must be left between the patch boundary and the air-absorber interface.**

- Input File Name: For Internal Mesh generator - Output Mesh File Name
  For FEM run Input Mesh File Name
- I I
  - 1 = User-supplied surface mesh, 2 = built-in mesh generator for circular patch,
    3 = built-in mesh generator for rectangular patch
  - For internal rectangular patch only: 1 = antenna patch, 2 = rectangular slot

Present only if the first row is "2"

- R I I I I
  - Radius of the surface mesh (all the way to the termination boundary)
  - Number of rings from the center to the edge of the antenna
  - Number of rings from the center to the edge of the cavity
  - Total number of rings in the surface mesh
  - Number of absorber rings

Present only if the first row is "3"

- R R I I I I I I I I
  - Incremental length in x-direction (uniform sampling)
  - " y-direction (uniform sampling)
  - Number of cells along x-dimension of the patch
  - " y-dimension of the patch
  - Number of cells between patch and cavity wall in x-direction
  - " y-direction
  - Number of cells between the cavity wall and the termination boundary in x-dir.
  - " y-dir.
  - Number of absorber cells in x-direction
  - " y-direction
- I I I I
  - # of substrate layers
  - 1 = all substrate layers are identical, 0 = otherwise
  - # of superstrate layers (enter zero for no superstrate)
  - 1 = all superstrate layers have the same thickness and material parameters, 0 = otherwise
- R C C
- R C C } Ordered from the bottom of the cavity up, each row corresponds to a substrate layer. Only one row is needed if all layers are identical (row has the info for a single layer).
- : : : :
- R C C
  - Thickness of the layer
  - Relative permittivity of the layer
  - Relative permeability of the layer
- R C C
- R C C } Same as above but for the superstrate. Ordered from the antenna surface up (first row corresponds to the layer just above the antenna surface).
- : : : :
- R C C
- I I
  - # of Conducting Layers
  - # of Conducting Pin Layers
- I I I I
  - Conducting Layer # (Top of Layer)
  - Expand the Layer: 1 = yes, 0 = no
  - Number of Segments to expand the layer
  - # of Holes in Layer
- I I
  - Surface Node Center of Hole
  - # Segments to Expand Hole

  } Repeat for each hole in conducting layer.

} Repeat for each conducting Layer

- I
- : : : :
- I
  - Conducting Pin Layers, All edges corresponding to identified conducting pin node in surface mesh and normal to surface in specified layer will be make conducting

- I → # of probe feeds
- I I I C
- I I I C } Each row corresponds to a probe feed
- : : : :
- I I I C

→ Surface node number #1
→ Surface node number #2 } Probe current flows from node #1 to node #2.
→ Layer # (layer within which the normally oriented probe is located, or the layer at top of which the laterally oriented probe is located). Entry can be positive or negative and increase away from the surface of the antenna with zero corresponding to the layer immediately below the antenna.
→ Complex amplitude of the probe current

- R R R I I R I I I

→ Starting frequency in GHz
→ Final frequency in GHz
→ Increment frequency in GHz
→ Frequency run to save (1 = save the first freq. run, 2 = next frequency, etc.)
→ 0 = BiCG, 1 = QMR
→ Tolerance (~ 0.01)
→ 1 = monitor convergence (dump residual error at each iter.), 0 = otherwise
→ 1 = compute element matrices, 0 = read in element matrices
→ 1 = write element matrix to file, 0 do not write matrix to file

- I → 1 = Read in user specified termination parameters (given in the following row), 0 = code will figure out the optimum parameters (this is the safe course if one is not familiar with the artificial absorber termination).

- R I I C

→ Thickness of one layer (all layers have the same thickness)
→ Total number of layers from the top of the outer-most superstrate layer to the termination boundary
→ Number of absorber layers
→ Relative permittivity of the absorbing layers.

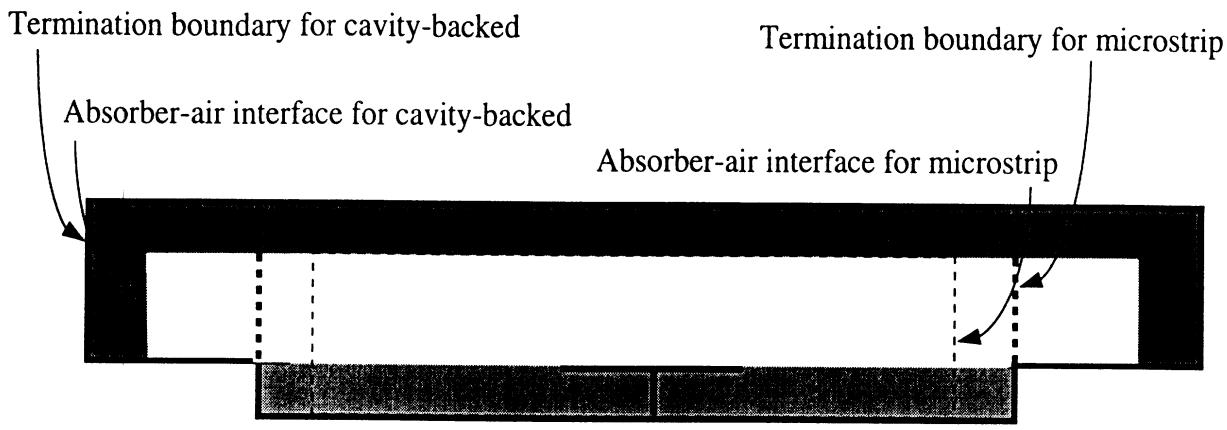


Figure 3: Microstrip vs cavity-backed configuration

10 Microstrip *vs* cavity-backed

As shown in Figure 3, the microstrip configuration can be realized as a cavity-backed configuration with the mesh terminated at the cavity walls. Basically, at first, the code treats every configuration as cavity-backed and if the extent of the mesh is the same as the cavity, the code recognizes it as a microstrip. Consequently, in the first row of *SurfMesh*, the second and third entries are identical and also the fourth and the fifth entries are identical for microstrip geometry.

In specifying a rectangular microstrip patch for the built-in mesh generator, zero should be entered for the distance between the cavity wall and the termination boundary (entries #7 and 8). For a circular patch, the same quantity should be entered for the number of rings from the center to the cavity wall and for the total number of rings in the mesh (entries #3 and 4).

11 How to create a user-defined surface mesh

The file *SurfMesh* contains surface mesh data. The format is given in Figure 4. As in Figure 2, here also the filled circles represent rows, and the letters I, R, or C imply real, integer, or complex number entries, respectively. The first row has information about the number of surface triangles and surface nodes. Starting from the second row is the information about the relation between the local and the global indexing of surface nodes. **It is important to note that the local numbering of surface nodes increase counter-clock wise. Also, the triangular patches are numbered starting from the antenna region, continuing with the region between the antenna boundary and the cavity boundary (if cavity-backed) and finishing**

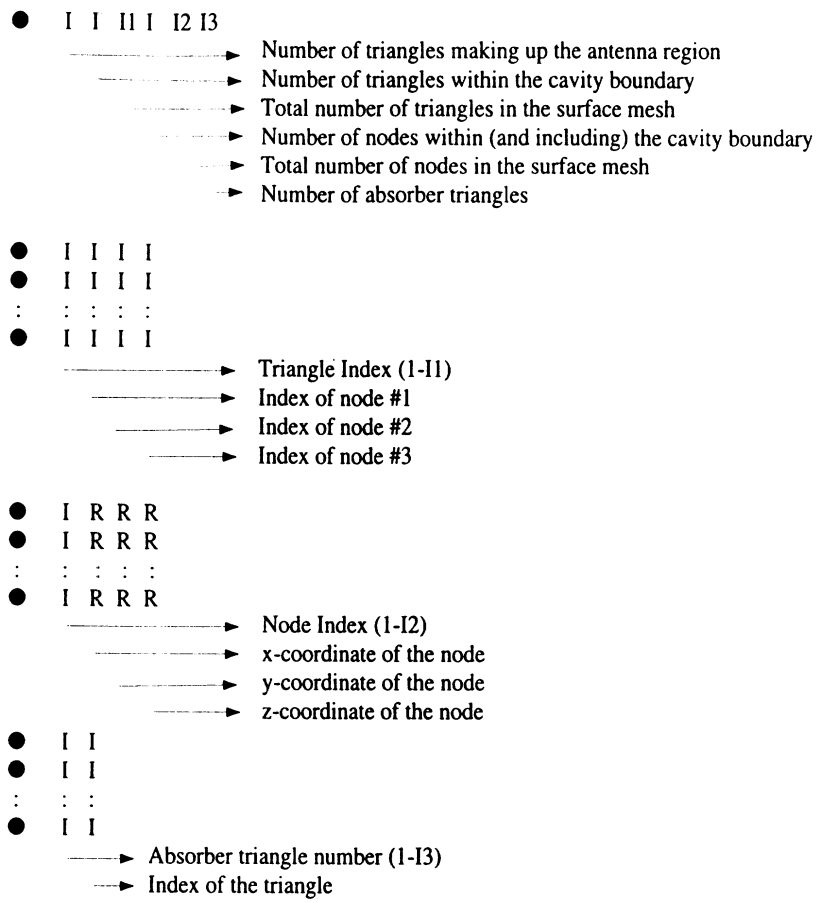


Figure 4: Input file *SurfMesh*

with the region between the cavity and the termination boundaries. This provides a simple way of identifying the antenna and cavity patches. For example, if N is the number of triangles making up the antenna region, triangles numbered 1 through N are the antenna triangles, etc.

The information about the coordinates of the nodes follows with each row corresponding to a node. Coordinates are in units of Centimeter. **It is also important to note that the nodes over the cavity aperture are numbered first.** This simplifies the volume indexing for cavity-backed antennas.

The last section of the file identifies the triangular patches making up the absorber section which is the outer skirt of the surface mesh.

12 Viewing the surface mesh using MatLab

For determining the node location of feeds and to inspect the mesh quality, it is useful to view the surface mesh. In fact, this is a must if the built-in mesh generator is

used to create the mesh unless one is familiar with the workings of the mesh generator.

The file *MeshPlot* contains a MatLab program for viewing the surface mesh. To run the code, execute the line commands in the given order. There are five seperate sections. The first section consists of six lines of commands, and it reads in the file *MeshDsply* (which has the mesh data) and sets up the screen. Before executing these commands, variables "nt" and "nn" (second and third lines) must be set to the number of triangles and number of nodes in the mesh, respectively. The following four sections fall under the titles *Display Mesh*, *Triangle Numbering*, *Global Node Numbering*, and *Local Node Numbering*. The functions of these sections are self-explanatory and sections can be executed in any order.

Index of each triangle is indicated at the center of the triangle and the index of each node is shown at the location of the node (the lines stop short of converging at the nodes in order not to cross over the text). The local indexing of three nodes of each triangle is indicated (as 1, 2, or 3) counter-clock wise just inside that node within the repective triangle. With all this information on the screen, the picture can look too crowded. In these cases, zooming on a particular section of the mesh is necessary for clarity. To do this just type *zoom* in the command window and click the part of the screen that interests you (with the left mouse botton).

From the first line of the program, one sees that it reads in the file *MeshDsply* which contains the node indexing and coordinates. *MeshDsply* is generated each time the code is run with user-supplied mesh option. If the built-in mesh generator is used, it is created at the same time as *SurfMesh*.

It should be noted that this plotting scheme is only useful for planar meshes. When the surface mesh defines a three dimensional curved surface , the MatLab program will display its projection onto the $x - y$ plane since it considers only the x and y coordinates of nodes.

13 Output file *EqvCur*

EqvCur is the file containing the equivalent magnetic currents (radiating in free-space) distributed across the platform surface. They need to be integrated to obtain the radiated field. They are the true currents in the case of planar antennas, and the Fortran code *FarField.f* can be used to integrate them for far field evaluation. In the case of non-planar platforms, the equivalent currents are local approximations to the true quantities.

The equivalent magnetic currents have been computed from the apeture electric field using $\mathbf{M} = 2\mathbf{E} \times \mathbf{n}$ where $\mathbf{n}$ is the surface normal pointing away from the surface. The

factor of 2 comes from removing the ground plane (assuming locally flat), implying that without the factor of 2, the currents radiate in the presence of the ground plane. Hence, given the ability to radiate the currents in the presence of the ground plane, the quantities in the file *EqvCur* must be divided by 2. This is possible for planar, cylindrical, spherical and conical ground planes.

The actual computation of the currents is carried out by averaging the electric field vector over each non-zero triangular patch (using the basis functions). The resulting vector is then crossed with the surface normal and the outcome is the average value of the equivalent magnetic current vector over that triangular patch. Except for the first row which indicates the number of patches (on which **M** is given) and the free-space wave number, each successive row contains the patch number, the area of the patch in cm^2, the (x, y, z) coordinates of the center of the patch (computed by averaging the coordinates of the three nodes of the patch), (x, y, z) coordiantes of the unit normal of the patch (pointing away from the platform), and the complex amplitudes of the (x, y, z) components of the magnetic current multiplied by a factor of 2 as noted earlier. The Fortran code *FarField.f* reads the magnetic current data in this format. Thus to compute the radiated field *FarField.f* must be executed with *EqvCur* as the input file.

14 Output file *EdgeUnk*

The file *EdgeUnk* stores the values to all edge unknowns. Needless to say, it has as many rows of information as the total number of edge unknowns. The first column is the index of the edge. The next six columns are the (x, y, z) coordinates of the end nodes of the edge. The second set of coordinates belong to the node toward which the edge points. The next column is the magnitude of the electric field vector (the unknown) which is parallel to and constant along the edge. The last two columns are the real and imaginary parts of the complex amplitude of the electric field unknown, respectively. All field quantities are in units of $Volts/cm$.

The frequency for which this information is saved is determined by the first integer entry of the frequency information row of the input file *MainInput* (see Figure 2). If the entry is zero, the file is not stored. This is often the user choice as the file takes up a substantial amount of memory and should be saved only if needed. The non-zero value of the entry specifies which frequeny run to save. The number of frequency runs are determined by the first three real entries of the same row.

15 Output file *Imp*

The file *Imp* stores the input impedance measured at the locations of the probe feeds. Input impedance is calculated as $Z_{in} = -E\,l\,/\,I$ where E is the complex amplitude of electric field unknown along the edge coinciding with the probe (in $Volts/cm$), l is the probe length (in $Centimeters$) and I is the complex amplitude of the probe current (in *Amperes*). The resulting impedance value is in units of *Ohms*. Here it has been assumed that the current in the probe flows in the direction of that edge.

The first column of the file is the frequency (in GHz), the second column is the probe number (in the order specified in the input file *MainInput*), the next two columns are the real and imaginary parts of the impedance (in *Ohms*), respectively, and the last column is the number of iterations the BiCG solver had to carry out for convergence. The same information is also dumped on the screen while the code is running. This is a very useful product of the code as it can be used to predict the resonance frequency of the antenna (the frequency at which the input impedance is purely real).

16 Output file *ElmMat*

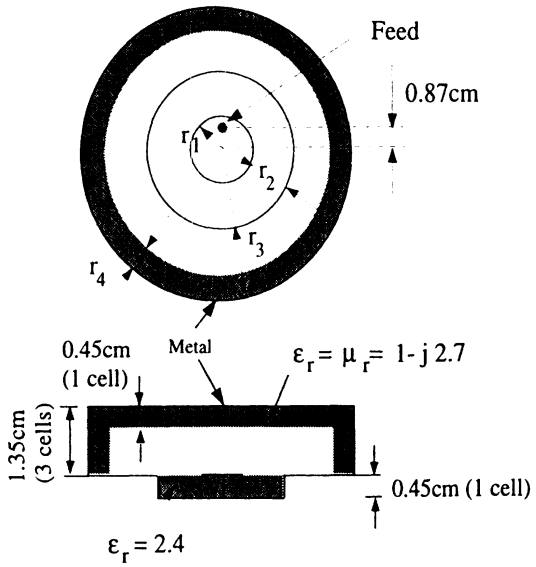
The file *ElmMat* stores the element matrices associated with the prisms making up the volume mesh. They are stored by the code every time a new volume mesh is introduced. Obviously, as long as the mesh stays the same so do the prisms and the element matrices for that matter. Therefore, if they are saved the first time the mesh is created, they are simply read. One disadvantage is that the file requires substantial memory space. The number of stored real entries (each with twelve decimals) is $162 \times Numberofprisms$ in the mesh. As shown in Figure 2, whether the data in *ElmMat* are computed or read in depends on the last entry of the frequency information row of the input file *MainInput*.

17 Demonstration runs

This section contains two demonstration runs which the user must carry out to insure that the code is working properly. For each run, the corresponding input/output files are provided in respective directories.

Demo #1: Cavity-backed circular patch

Before proceeding with the rest of the section, the reader must be advised that these demonstration runs are intended to show the operation of the code and should not be used as a measure of the code's accuracy. For example, both the thickness and the distance of the absorbing layer have been chosen half or one third of what they should



r_1 = 1.3cm (3 rings)
r_2 = 1.3cm (3 rings)
r_3 = 1.3cm (3 rings)
r_4 = 0.43cm (1 ring)

(a)

(b)

MainInput (for mesh generation)

```
1
3.9  3  6  9  1
```

MainInput (for run)

```
1
1  1  0  0
.45  (2.14, 0.)  (1., 0.)
1
8  8  0  (1., 0.)
3.  4.  .1  6  .01  0  1
1
.45  3  1  (1., -2.7)
```

(c)

Output on the screen:

```
SURFACE EDGE INDEXING ...
NUMBER OF ANTENNA EDGES= 90
NUMBER OF CAVITY EDGES= 342
TOTAL NUMBER OF EDGES= 756
VOLUME NODAL AND EDGE INDEXING ...
NUMBER OF PRISMS= 1674
NUMBER OF GLOBAL EDGES= 4306
NUMBER OF GLOBAL NODES= 1211
NUMBER OF BOUNDARY EDGES= 54
NUMBER OF BOUNDARY NODES= 54
NUMBER OF CAVITY BOUNDARY EDGES= 36
NUMBER OF CAVITY BOUNDARY NODES= 36
NUMBER OF GLOBAL METAL EDGES= 1944
NUMBER OF ABSORBER PRISMS= 690
NUMBER OF NON-ZERO EDGES= 2362
COMPUTING ELEMENT MATRICES ...
0 % Done
10 % Done
20 % Done
:
80 % Done
90 % Done
100 % Done
Begin Frequency Sweep
```

Freq(GHz)	Feed #	Re(Zin)	Im(Zin)	# of Iterat.
3.0000	1	7.8512	59.1441	223
:				
3.5000	1	46.6215	114.9450	222
3.6000	1	81.9289	128.6968	221
3.7000	1	142.2111	115.9399	228
3.8000	1	181.5739	39.8753	227
3.9000	1	141.1083	-28.9999	231
4.0000	1	89.7684	-41.3362	225

Resonance

Figure 5: Cavity-backed circular patch: (a) Geometry of the patch and finite element-artificial absorber modeling, (b) contents of the input files, (c) screen dump produced by the code and the input impedance as a function of excitation frequency.

be to minimize the geometry and hence the CPU time required by each frequency run. For proper modeling of the antennas, both the thickness and the distance of the absorber must be at least $0.15\lambda_o$ at the operation frequency.
The first demonstration example is a cavity-backed circular patch. Figure 5(a) shows the patch and termination geometry. Note that in order to save CPU time, only one layer of absorber ($0.05\lambda_o$ thick) has been employed and placed about $0.1\lambda_o$ away from the cavity surface and walls. Figure 5(b) shows the contents of the input file for both the mesh generation and the actual run. Figure 5(c) shows the screen dump of the code after the run. An inspection of the frequency sweep shows that the antenna is resonant at $3.85GHz$. The input/output files are provided in electronic form in the directory "Demo1".

Demo #2: Microstrip rectangular patch with two layers of overlay

The second case is a microstrip rectangular patch with two layers of superstrate. Similar information for this patch is given in Figure 6. Note in Figure 6(a), the absorber sections that are in direct contact with the substrate and superstrate layers are colored differently to indicate that they have different material constants and the code automatically determines the permittivity and permeability of these sections in such a way that the waves normally incident on these sections of the absorber are totally absorbed, i.e., the wave impedances ($Z = \sqrt{\frac{\mu}{\epsilon}}$) on both sides of the interface are matched. This is clearly shown in Figure 1. Notice that the absorber section in contact with air have its relative permittivity and permeability equal to each other ($\epsilon_r = \mu_r$) resulting in the wave impedance inside the absorber section's being equal to that of the free-space (since $\sqrt{\frac{\mu}{\epsilon}} = \sqrt{\frac{\mu_r \mu_o}{\epsilon_r \epsilon_o}} = \sqrt{\frac{\mu_o}{\epsilon_o}} = Z_o$). As before, Figure 6(b) shows the contents of the input files for both mesh generation and actual run, and Figure 6(c) shows the screen dump created by the code after the run. The data show that the patch resonates at $5.05GHz$. The input/output files are provided in electronic form in the directory "Demo2".

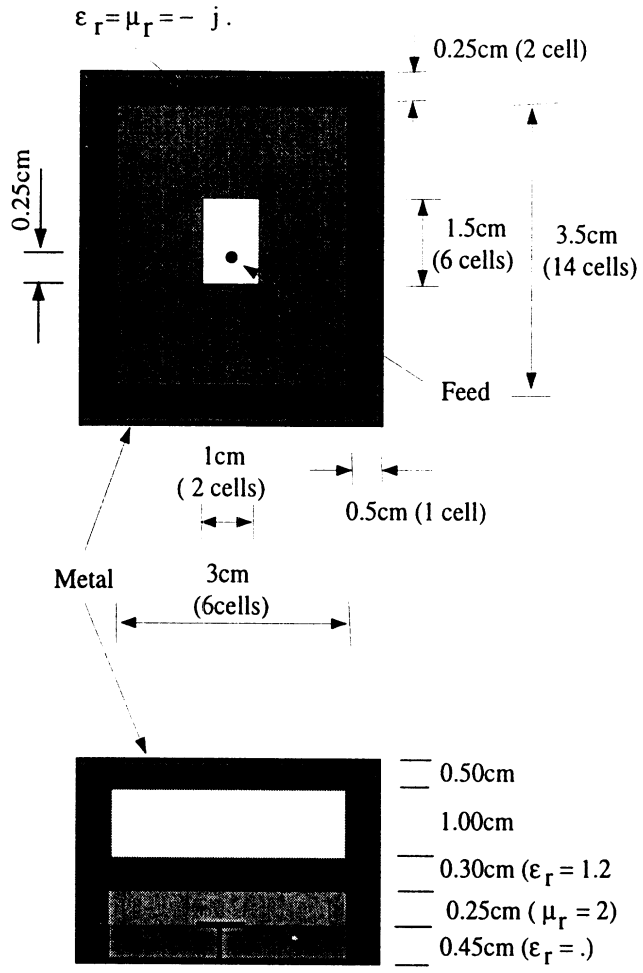
Note that, the input files for the first runs (for mesh generation) are not provided in electronic form since they can easily be copied from the figures. Also the files *ElmMat* and *EdgeUnk* are not provided due to memory restrictions.

18 Distribution disk and installation of the code

Below is the directory list of the distribution list:

README	: Text file containing brief information about the distribution disk
FEMA=PRISM.f	: Source code for FEMA-PRISM
FarField.f	: Source code for Far Field evaluation

(a)



(b)

MainInput (for mesh generation)

```
3
0.5 0.25 2 6 3 6 0 0 1 2
```

MainInput (for the actual run)

```
1
1 1 2 0
.45 (2.14,0.) (1.,0.)
0.25 (1.,0.) (2.,0.)
0.3 (1.2,0.) (1.,0.)
1
23 23 0 (1.,0.)
4.5 5.5 .1 6 .01 0 1
1
.45 3 1 (1.,-2.7)
```

(c)

Screen dump

```
SURFACE EDGE INDEXING ...
NUMBER OF ANTENNA EDGES= 44
NUMBER OF CAVITY EDGES= 458
TOTAL NUMBER OF EDGES= 458
VOLUME NODAL AND EDGE INDEXING ...
NUMBER OF PRISMS= 1728
NUMBER OF GLOBAL EDGES= 4232
NUMBER OF GLOBAL NODES= 1197
NUMBER OF BOUNDARY EDGES= 52
NUMBER OF BOUNDARY NODES= 52
NUMBER OF CAVITY BOUNDARY EDGES= 52
NUMBER OF CAVITY BOUNDARY NODES= 52
NUMBER OF GLOBAL METAL EDGES= 1532
NUMBER OF ABSORBER PRISMS= 888
NUMBER OF NON-ZERO EDGES= 2700
COMPUTING ELEMENT MATRICES ...
0 % Done
10 % Done
20 % Done
:
90 % Done
100 % Done
Begin Frequency Sweep
```

Resonance

Freq(GHz)	Feed #	Re(Zin)	Im(Zin)	# of Iterat.
4.5000	1	32.6467	103.0583	411
4.6000	1	46.5897	116.9454	451
4.7000	1	70.2919	131.6365	426
4.8000	1	110.8929	140.4605	436
4.9000	1	170.3384	121.6465	505
5.0000	1	211.6461	49.7454	504
5.1000	1	184.2637	-28.5242	459
5.2000	1	129.2563	-57.3716	428
5.3000	1	88.1359	-55.3745	342
5.4000	1	62.9073	-44.5909	328
5.5000	1	47.5992	-32.9925	321

Figure 6: Microstrip rectangular patch with multiple superstrates: (a) Geometry of the patch and finite element-artificial absorber modeling, (b) contents of the input files, (c) screen dump produced by the code and the input impedance as a function of excitation frequency.

MeshPlot	: File containing the MatLab program
Demo1	: Directory containing the files for Demo #1
Demo2	: Directory containing the files for Demo #2

/Demo1:

MainInput	: File *MainInput* for Demo #1
SurfMesh	: File *SurfMesh* for Demo #1
MeshDsply	: File *MeshDsply* for Demo #1
EqvCur	: File *EqvCur* for Demo #1
Imp	: File *Imp* for Demo #1

/Demo2:

MainInput	: File *MainInput* for Demo #2
SurfMesh	: File *SurfMesh* for Demo #2
MeshDsply	: File *MeshDsply* for Demo #2
EqvCur	: File *EqvCur* for Demo #2
Imp	: File *Imp* for Demo #2

The disk is formatted on a Power Machintosh. The contents of the disk should be loaded into the working directory. No extra effort needed to install the code. Section 6 explains how to run the code.

References

[1] Özdemir, T. and J. L. Volakis, "Triangular prisms for edge-based vector finite element antenna analysis," *Radiation Laboratory Tech. Rep. No. 031307-4-T*, Radiation Laboratory, Dept. of Elect. Engr. Comp. Sci., Univ. of Michigan, Ann Arbor, Michigan 48109-2122, March 1995.

[2] Özdemir, T., J. Gong, S. Legault, J. Volakis, T. Senior, J. Berrie, R. Kipp and H. Wang, "Modeling of conformal antennas on doubly curved platforms and their interactions with aircraft platforms," *Annual Progress Report, Tech. Rep. No. 031307-5-T*, Radiation Laboratory, Dept. of Elect. Engr. Comp. Sci., Univ. of Michigan, Ann Arbor, Michigan 48109-2122, October 1995.

AutoCAD FEMA-PRISM Interface Manual

M. D. Casciato and J. L. Volakis

Radiation Laboratory
Department of Electrical Engineering & Computer Science
University of Michigan
Ann Arbor, Michigan 48109-2122
December 18, 1996

Introduction

This manual contains instructions on converting an AutoCAD .dxf boundary file to a FEMA-PRISM compatible surfmesh triangular surface mesh file. In addition it describes the ability of FEMA-PRISM to implement conducting pins, conducting layers, and substrate holes. A description of FEMA-PRISM post-processing routines are also included. This is a 3 step procedure in which the .dxf file is converted by the FORTRAN routine AcadProc.f to a .poly file compatible with the meshing package TRIANGLE [3]. TRIANGLE is then used to create the mesh, contained in output files .node and .ele, which is then converted to a FEMA-PRISM compatible "surfmesh" file by the FORTRAN routine TriProc.f Users are referred to [3] for details of TRIANGLE file formats.

The procedure to convert a .dxf file to a FEMA-PRISM "surfmesh file" is as follows:

1. Convert the AutoCAD .dxf file to the TRIANGLE .poly input file using AcadProc.f
2. Mesh the .poly file using TRIANGLE.
3. Convert the TRIANGLE output files, .1.ele and .1.node, to a FEMA-PRISM surfmesh file using TriProc.f

Note that detailed instructions on how to use each part of the package follows.

AcadProc

AcadProc is a FORTRAN routine which converts an AutoCAD .dxf file to a .poly file used by TRIANGLE for mesh generation. Figure 1 show a sample AutoCAD .dxf file containing both line and layer information. Before using AcadProc certain procedures must be used in creating the AutoCAD drawing. Referring to Figure 2 each line boundary in AutoCAD must be saved on a different layer numbered consecutively starting with 0 for the outer layer. AcadProc uses this layer information to identify bounded regions in the antenna. An algorithm in AcadProc finds the left (assumes +x right) vertical boundary (y-directed) in each region. It then identifies a point (x,y coordinate) in each region. and places an identifying integer label on the region (1 is the outermost region with 0 it's identifier). These region coordinates and identifiers are placed at the end of

the .poly file as shown below (labels added for clarification).

9 --> Number of regions.

Region #	x-coordinate	y-coordinate	region identifier	area weighting
1	0.47470	12.35628	0	0.35
2	1.42410	12.35628	1	0.35
3	2.37350	12.35628	2	0.35
4	3.32290	12.35628	3	0.35
5	4.27230	12.35628	4	0.2
6	5.22170	12.35628	5	0.2
7	8.78505	12.35628	6	0.2
8	12.06674	12.35628	7	0.075
9	14.76925	12.35628	8	0.075

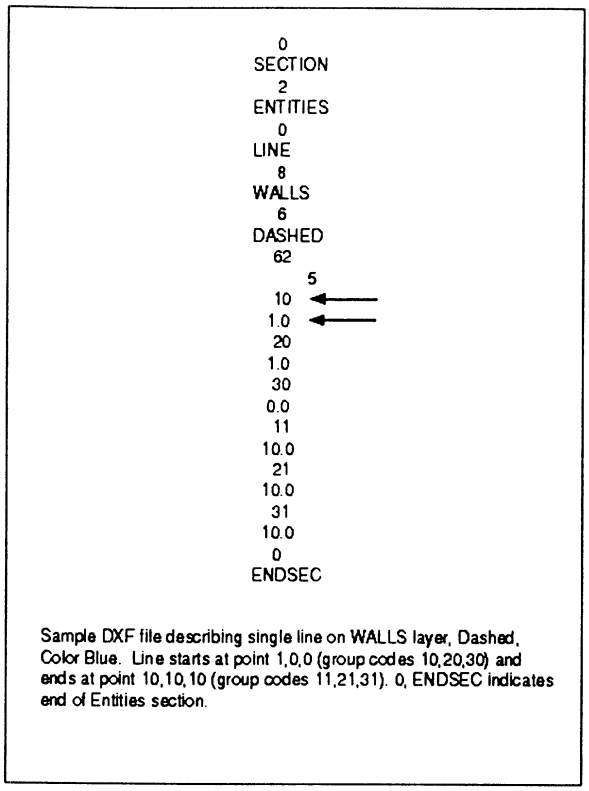


Figure 1: Sample AutoCAD .dxf File.

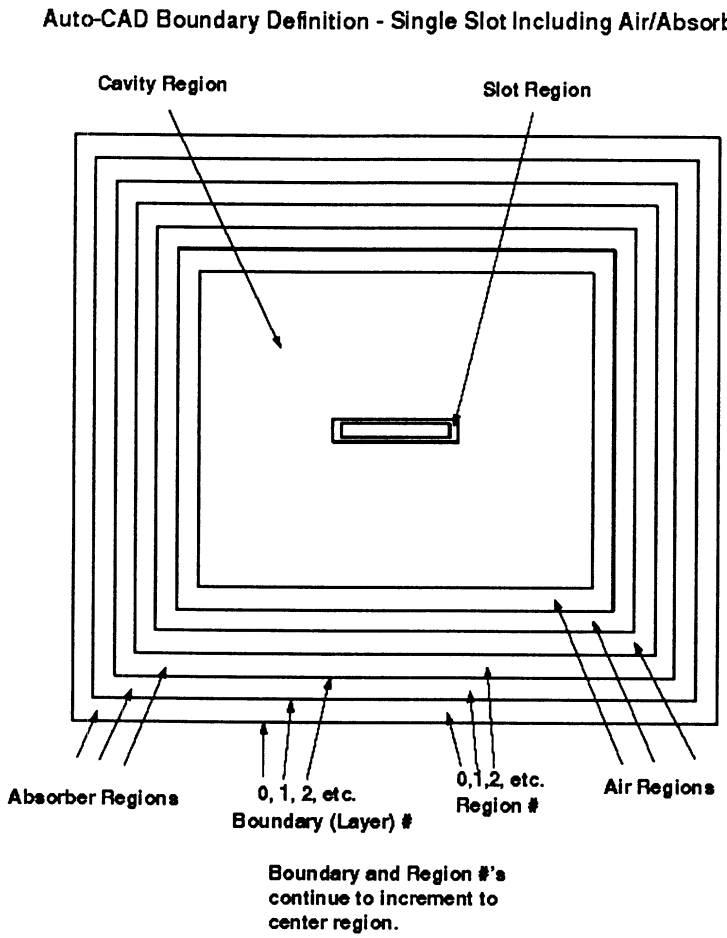


Figure 2: Auto-CAD Boundary (Layer #'s) and Region definitions.

TRIANGLE will then give all elements created in these bounded regions the same identifier as the point within the region when the "-A" option is used (this will be explained in the next section). This simplifies the separation of different element types (conducting, absorber, etc.) by TriProc. Note that the algorithm for finding the region points is not perfected. The user should inspect the .poly file and verify that the number of regions, the x,y coordinates, and the region identifiers are correct. An explanation of how the region identifiers are used will follow in the TriProc section. The area weightings are added by the user. When the "-a" option is used (again explained in the next section) TRIANGLE attempts to restrict the maximum element area in each region using the area weightings as a guide. This is an iterative process where the user will need to try different weightings to get the desired mesh.

To use AcadProc first modify the input file Acad.in shown below (with comments added):

```
smplsltabs6.dxf          !Input Autocad DXF file.
smplsltabs6.poly         !Output TRIANGLE .poly line file.
2                !Units of Drawing. Meters(1), cm(2), mm(3), Feet(4), inches(5)
0                !(0) no phasing slots, (1) phasing slots.
```

The first 2 lines are self explanatory. The third line indicates the units of the AutoCAD drawing (FEMA-PRISM input file "surfmesh" must be in cm). The third line indicates whether phasing slots (such as in the LPSA) exist in the antenna. If so AcadProc attempts to automatically find the interior boundary points for identification purposes as described earlier. To run AcadProc simply type the executable "acadproc" to generate the .poly file.

TRIANGLE

TRIANGLE, written in C, is a 2-D mesh generation package using Delaunay Triangulation algorithms [4]. It was written by J.R. Shewchuk of the School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA (jrs@cs.cmu.edu). It is available for free on Netlib and at web location:

http://www.cs.cmu.edu/~quake/TRIANGLE.html

Showme, a tool for viewing TRIANGLE input and output files and written for unix workstations is also available at this website. Showme will also output .ps and .eps files for printing.

For details on how to use TRIANGLE the user is referred to [3]. To illustrate the usage of TRIANGLE for our application a simple example will be given, meshing the single element slot shown in figure 2. The region identifiers and weightings described earlier will be used for this example.

To run TRIANGLE simply type:

triangle -pAq28as smplsltabs6.poly

where triangle is the executable and smplsltabs6.poly is the input .poly file. The "-" sign precedes any options used. The options are as follows:

The "p" tells TRIANGLE to look for a .poly file as the input.

The "A" tells TRIANGLE to look for the region identifiers at the end of the .poly file and uses them to identify triangles created in each region.

The "q28" indicates that the minimum acceptable interior angle of any element is 28 degrees. If no angle is specified the default is 30 degrees.

The "a" tells TRIANGLE to use the area constraints at the end of the .poly file. If "a" is followed immediately by a number this option will attempt to constrain the maximum area of all triangles in the mesh to the number specified.

The "s" forces segments into mesh by splitting.

The user can see all options available in TRIANGLE by simply typing the executable "triangle".

"triangle -h" will output the TRIANGLE user's manual [3] to standard output (screen).

Figure 3 shows the folded slot drawing dimensions as well as the mesh created by TRIANGLE using the indicated options. This figure was generated from a postscript file created by Showme.

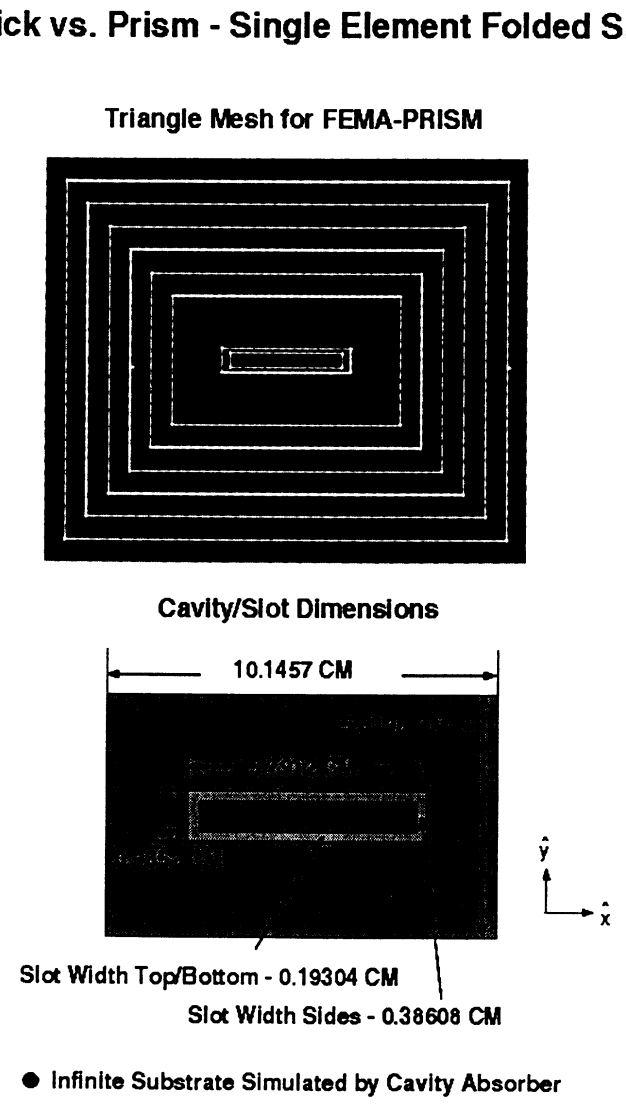


Figure 3: Single Slot - Dimensions and TRIANGLE Mesh.

TRIANGLE will output 3 files:

smplsltabs6.1.poly
smplsltabs6.1.node
smplsltabs6.1.ele

The ".1." indicates revision 1. If TRIANGLE is executed using the ".1." files it will create revision ".2.", etc. for higher numbers. The .1.poly is a duplicate of the original .poly file. The .1.node file contains node coordinates and the .1.ele contains connectivity. These 2 files are used as the input for TriProc. Note that all output files created by TRIANGLE end with the TRIANGLE options used to generate it, for future reference.

TriProc

The final step in creating a surfmesh file for FEMA-PRISM is to convert the .1.node and .1.ele TRIANGLE outputs to the "surfmesh" format using the FORTRAN routine TriProc. FEMA-PRISM needs the conducting, dielectric (no conducting patch over cavity), air, and absorber regions separated [1]. TriProc first reads the node and element information generated by TRIANGLE in the .1.node and .1.poly mesh files, including the element identifiers in each boundary region. As mentioned previously, the boundary regions are identified by an integer which starts at "0" for the outermost region and increases inward toward the antenna center. TRIANGLE, using the "-A" option, then attaches this identifier to all TRIANGLEs created in this region. TriProc first sorts triangles in the air/absorber regions. It then needs to sort triangles within the antenna cavity regions depending on whether they are conducting or dielectric. If the user has specified that this is a slot antenna the first region in from the air/absorbers is assumed conducting (i.e., conducting triangles). If a patch antenna is specified the first region in from the air/absorbers is dielectric. Triangle types within regions then alternate between dielectric and conducting (or vice-versa) inward toward antenna center. An example input file to TriProc (called Tri.in) follows:

```
smplsltabs6.1.node     !Input TRIANGLE Node File
smplsltabs6.1.ele      !Input TRIANGLE Element File
1                !(0) patch, (1) slot
1                !(0) cavity, (1) microstrip
1                !(0) BI, (1) absorber
3                ! # absorber layers (dummy for BI)
3                ! # air layers (dummy for BI)
```

The first 2 lines are the input files from TRIANGLE. The next line specifies slot or patch antenna. Line 4 indicates cavity or microstrip. If the cavity option is chosen the antenna will be placed in a PEC cavity. If the microstrip option is chosen, the cavity walls will be extended and an absorber will be placed along the former cavity boundary creating a microstrip effect [1]. Line 5 will be 1 for absorber, with lines 6 and 7 showing the number of absorber/air layers respectively. An algorithm in TriProc uses the total number of air/absorber layers to identify the first region that is in the actual antenna cavity. From this TriProc determines the conducting/dielectric regions in the antenna cavity.

The output of TriProc is a FEMA-PRISM compatible mesh file called "surfmesh". To run Triproc simply type the executable "triproc" after modifying Tri.in appropriately.

Post-Processing Data

Far field patterns can be generated using the FORTRAN routine FarField.f. Farfield reads in the magnetic current file EqvCur generated by FEMA-PRISM. Simply type the executable "farfield" and the user is prompted for the appropriate inputs as follows:

Magnetic Current file
Name of Output pattern file
Name of Output field file
Theta Begin, End, Step Size
Phi Begin, End, Step Size
DB amount to be subtracted from computation (outputis unnormalized)
Does user want quiver plot (matlab) of magnetic current.

If quiver plot is desired the user will be prompted for:

File name for X coordinates
File name for Y coordinates
File name for X component
File name for Y component

The pattern file output 5 columns of data consisting of:

Theta Phi Power (dB) Theta Power (dB) Phi Total Power (dB)

Quiver is a matlab routine for plotting localized field vectors. Figure 4 shows a sample quiver output for a single element folded slot. When running matlab load the X,Y coordinates/components data files generated by FarField. The matlab command quiver(x,y,my,mx) will plot the electric field vectors of each element on the antenna surface, where x,y are the X,Y coordinate data points, and my, mx are the Y,X component data points.

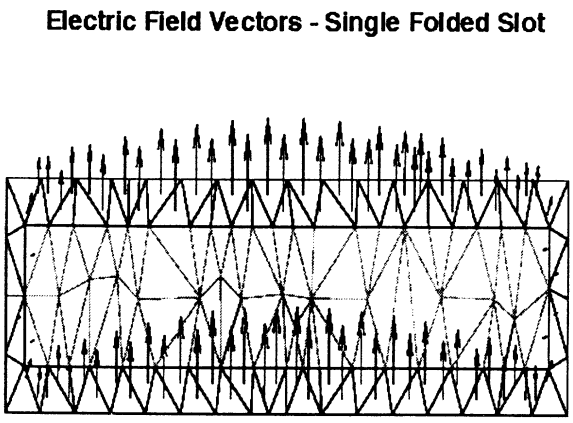


Figure 4: Quiver Vector Electric Field Plot - Folded Slot.

Notes

The user is referred to [1] for details on operating FEMA-PRISM. Remember that the Meshdsply routine described in [1] must be used to find probe node numbers before running FEMA-PRISM. Also a new version of FEMA-PRISM included in this package allows for the addition of vertical conducting pins (normal to surface mesh), layers, and substrate holes. Reference [1] contains an updated description of the FEMA-PRISM input file to allow for these additions. Their usage is described as follows:

Conducting Pins: To use this option at this time the FEMA-PRISM surfmesh input file must be modified to identify the nodes under which vertical conducting pins exist. Simply add the con-

ducting pin number and corresponding surface node number to the end of the surface mesh file, for example:

1 32
2 24
.
.

This shows 2 conducting pin locations under surface node numbers 32, and 24. In addition change the 7th (of 8) field in line 1 of the "surfmesh" file from 0 to the number of conducting pins (2 in this case). In the FEMA-PRISM input file the user states the number of conducting pin layers and the layer number the pins are in ([1], layer 1 at bottom of cavity). NOTE: In the current FEMA-PRISM user's manual [1] the "surfmesh" file description does not show fields 7 and 8 of the first line of the file. To run this version of FEMA-PRISM these fields must exist (both set at 0 if no conducting pins or layers). TriProc automatically sets these fields to 0.

Conducting Layers: These are essentially substrate conducting patches duplicating the conducting patch on the antenna surface. Again in the input file the user specifies the top of each layer on which the patches reside (layer 1 at bottom again [1]).

Expanded Layers/Holes: The conducting layers (substrate patches) described in the previous section can be expanded in size. In additon holes can now be inserted into these substrate patches. In the FEMA-PRISM input file the user specifies whether to expand the substrate patch, and if so how many segments to expand it. For each segment specified the non-conducting edges connected to the conducting patch are changed to conducting as well as the segments that connect these new conducting edges, thus expanding the patch by one layer of triangles. The user can also specifie if the patch contains a hole, the number of segments to expand the hole [1], and the hold center. The center is specified by a corresponding surface node location. For the holes the conducting mesh edges connected to hole center node are made non-conducting. For each additional segment the hole is expanded, conducting edges connecting hole segments are made non-conducting, as well as the next layer of edges outward. For details of the new FEMA-PRISM input file users are referred to [1].

Bibliography

[1] T. Ozdemir and J. L. Volakis, "Users manual for FEMA-PRISM, "Univ of Michigan Radiation Lab. Techn. Report 031307-6-T, March 1996. 15pp.

[2] T. Ozdemir and J. L. Volakis, "Triangular prisms for edge-based vector finite element analysis of conformal antennas," Univ of Michigan Radiation Lab., October 1996.

[3] J. R. Shewchuk, "TRIANGLE, A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator. v1.3," Carnegie Mellon University.

[4] J. R. Shewchuk, "TRIANGLE: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator," School of Computer Science, Carnegie Mellon University.

Triangle
A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator.
Version 1.3

Triangle generates exact Delaunay triangulations, constrained Delaunay triangulations, and quality conforming Delaunay triangulations. The latter can be generated with no small angles, and are thus suitable for finite element analysis. If no command line switches are specified, your .node input file will be read, and the Delaunay triangulation will be returned in .node and .ele output files. The command syntax is:

triangle [-prq__a__AcevngBPNEIOXzo_YS__iFlsCQVh] input_file

Underscores indicate that numbers may optionally follow certain switches; do not leave any space between a switch and its numeric parameter. input_file must be a file with extension .node, or extension .poly if the -p switch is used. If -r is used, you must supply .node and .ele files, and possibly a .poly file and .area file as well. The formats of these files are described below.

Command Line Switches:

- -p Reads a Planar Straight Line Graph (.poly file), which can specify points, segments, holes, and regional attributes and area constraints. Will generate a constrained Delaunay triangulation fitting the input; or, if -s, -q, or -a is used, a conforming Delaunay triangulation. If -p is not used, Triangle reads a .node file by default.
- -r Refines a previously generated mesh. The mesh is read from a .node file and an .ele file. If -p is also used, a .poly file is read and used to constrain edges in the mesh. Further details on refinement are given below.
- -q Quality mesh generation by Jim Ruppert's Delaunay refinement algorithm. Adds points to the mesh to ensure that no angles smaller than 20 degrees occur. An alternative minimum angle may be specified after the 'q'. If the minimum angle is 20.7 degrees or smaller, the triangulation algorithm is theoretically guaranteed to terminate (assuming infinite precision arithmetic - Triangle may fail to terminate if you run out of precision). In practice, the algorithm often succeeds for minimum angles up to 33.8 degrees. For highly refined meshes, however, it may be necessary to reduce the minimum angle to well below 20 to avoid problems associated with insufficient floating-point precision. The specified angle may include a decimal point.
- -a Imposes a maximum triangle area. If a number follows the 'a', no triangle will be generated whose area is larger than that number. If no number is specified, an .area file (if -r is used) or .poly file (if -r is not used) specifies a number of maximum area constraints. An .area file contains a separate area constraint for each triangle, and is useful for refining a finite element mesh based on a posteriori error estimates. A .poly file can optionally contain an area constraint for each segment-bounded region, thereby enforcing triangle densities in a first triangulation. You can impose both a fixed area constraint and a varying area constraint by invoking the -a switch twice, once with and once without a number following. Each area specified may include a decimal point.

-A Assigns an additional attribute to each triangle that identifies what segment-bounded region each triangle belongs to. Attributes are assigned to regions by the .poly file. If a region is not explicitly marked by the .poly file, triangles in that region are assigned an attribute of zero. The -A switch has an effect only when the -p switch is used and the -r switch is not.

-c Creates segments on the convex hull of the triangulation. If you are triangulating a point set, this switch causes a .poly file to be written, containing all edges in the convex hull. (By default, a .poly file is written only if a .poly file is read.) If you are triangulating a PSLG, this switch specifies that the interior of the convex hull of the PSLG should be triangulated. If you do not use this switch when triangulating a PSLG, it is assumed that you have identified the region to be triangulated by surrounding it with segments of the input PSLG. Beware: if you are not careful, this switch can cause the introduction of an extremely thin angle between a PSLG segment and a convex hull segment, which can cause overrefinement or failure if Triangle runs out of precision. If you are refining a mesh, the -c switch works differently; it generates the set of boundary edges of the mesh, rather than the convex hull.

-e Outputs (to an .edge file) a list of edges of the triangulation.

-v Outputs the Voronoi diagram associated with the triangulation. Does not attempt to detect degeneracies.

-n Outputs (to a .neigh file) a list of triangles neighboring each triangle.

-g Outputs the mesh to an Object File Format (.off) file, suitable for viewing with the Geometry Center's Geomview package.

-B No boundary markers in the output .node, .poly, and .edge output files. See the detailed discussion of boundary markers below.

-P No output .poly file. Saves disk space, but you lose the ability to impose segment constraints on later refinements of the mesh.

-N No output .node file.

-E No output .ele file.

-I No iteration numbers. Suppresses the output of .node and .poly files, so your input files won't be overwritten. (If your input is a .poly file only, a .node file will be written.) Cannot be used with the -r switch, because that would overwrite your input .ele file. Shouldn't be used with the -s, -q, or -a switch if you are using a .node file for input, because no .node file will be written, so there will be no record of any added points.

-O No holes. Ignores the holes in the .poly file.

-X No exact arithmetic. Normally, Triangle uses exact floating-point arithmetic for certain tests if it thinks the inexact tests are not accurate enough. Exact arithmetic ensures the robustness of the triangulation algorithms, despite floating-point roundoff error. Disabling exact arithmetic with the -X switch will cause a small improvement in speed and create the possibility (albeit small) that Triangle will fail to produce a valid mesh. Not recommended.

-z Numbers all items starting from zero (rather than one). Note that this switch is normally overrided by the value used to number the first point of the input .node or .poly file. However, this switch is useful when calling Triangle from another program.

-o2 Generates second-order subparametric elements with six nodes each.

-Y No new points on the boundary. This switch is useful when the mesh boundary must be preserved so that it conforms to some adjacent mesh. Be forewarned that you will probably sacrifice some of the quality of the mesh; Triangle will try, but the resulting mesh may contain triangles of poor aspect ratio. Works well if all the boundary points are closely spaced. Specify this switch twice ('-YY') to prevent all segment splitting, including internal boundaries.

-S Specifies the maximum number of Steiner points (points that are not in the input, but are added to meet the constraints of minimum angle and maximum area). The default is to allow an unlimited

number. If you specify this switch with no number after it, the limit is set to zero. Triangle always adds points at segment intersections, even if it needs to use more points than the limit you set. When Triangle inserts segments by splitting (-s), it always adds enough points to ensure that all the segments appear in the triangulation, again ignoring the limit. Be forewarned that the -S switch may result in a conforming triangulation that is not truly Delaunay, because Triangle may be forced to stop adding points when the mesh is in a state where a segment is non-Delaunay and needs to be split. If so, Triangle will print a warning.

-i Uses an incremental rather than divide-and-conquer algorithm to form a Delaunay triangulation. Try it if the divide-and-conquer algorithm fails.

-F Uses Steven Fortune's sweepline algorithm to form a Delaunay triangulation. Warning: does not use exact arithmetic for all calculations. An exact result is not guaranteed.

-l Uses only vertical cuts in the divide-and-conquer algorithm. By default, Triangle uses alternating vertical and horizontal cuts, which usually improve the speed except with point sets that are small or short and wide. This switch is primarily of theoretical interest.

-s Specifies that segments should be forced into the triangulation by recursively splitting them at their midpoints, rather than by generating a constrained Delaunay triangulation. Segment splitting is true to Ruppert's original algorithm, but can create needlessly small triangles near external small features.

-C Check the consistency of the final mesh. Uses exact arithmetic for checking, even if the -X switch is used. Useful if you suspect Triangle is buggy.

-Q Quiet: Suppresses all explanation of what Triangle is doing, unless an error occurs.

-V Verbose: Gives detailed information about what Triangle is doing. Add more 'V's for increasing amount of detail. '-V' gives information on algorithmic progress and more detailed statistics. '-VV' gives point-by-point details, and will print so much that Triangle will run much more slowly. '-VVV' gives information only a debugger could love.

-h Help: Displays these instructions.

Definitions:

A Delaunay triangulation of a point set is a triangulation whose vertices are the point set, having the property that no point in the point set falls in the interior of the circumcircle (circle that passes through all three vertices) of any triangle in the triangulation.

A Voronoi diagram of a point set is a subdivision of the plane into polygonal regions (some of which may be infinite), where each region is the set of points in the plane that are closer to some input point than to any other input point. (The Voronoi diagram is the geometric dual of the Delaunay triangulation.)

A Planar Straight Line Graph (PSLG) is a collection of points and segments. Segments are simply edges, whose endpoints are points in the PSLG. The file format for PSLGs (.poly files) is described below.

A constrained Delaunay triangulation of a PSLG is similar to a Delaunay triangulation, but each PSLG segment is present as a single edge in the triangulation. (A constrained Delaunay triangulation is not truly a Delaunay triangulation.)

A conforming Delaunay triangulation of a PSLG is a true Delaunay triangulation in which each PSLG segment may have been subdivided into several edges by the insertion of additional points. These inserted points are necessary to allow the segments to exist in the mesh while

maintaining the Delaunay property.

File Formats:

All files may contain comments prefixed by the character '#'. Points, triangles, edges, holes, and maximum area constraints must be numbered consecutively, starting from either 1 or 0. Whichever you choose, all input files must be consistent; if the nodes are numbered from 1, so must be all other objects. Triangle automatically detects your choice while reading the .node (or .poly) file. (When calling Triangle from another program, use the -z switch if you wish to number objects from zero.) Examples of these file formats are given below.

.node files:
First line: <# of points> <dimension (must be 2)> <# of attributes>
<# of boundary markers (0 or 1)>
Remaining lines: <point #> <x> <y> [attributes] [boundary marker]

The attributes, which are typically floating-point values of physical quantities (such as mass or conductivity) associated with the nodes of a finite element mesh, are copied unchanged to the output mesh. If -s, -q, or -a is selected, each new Steiner point added to the mesh will have attributes assigned to it by linear interpolation.

If the fourth entry of the first line is '1', the last column of the remainder of the file is assumed to contain boundary markers. Boundary markers are used to identify boundary points and points resting on PSLG segments; a complete description appears in a section below. The .node file produced by Triangle will contain boundary markers in the last column unless they are suppressed by the -B switch.

.ele files:
First line: <# of triangles> <points per triangle> <# of attributes>
Remaining lines: <triangle #> <point> <point> <point> ... [attributes]

Points are indices into the corresponding .node file. The first three points are the corners, and are listed in counterclockwise order around each triangle. (The remaining points, if any, depend on the type of finite element used.) The attributes are just like those of .node files. Because there is no simple mapping from input to output triangles, an attempt is made to interpolate attributes, which may result in a good deal of diffusion of attributes among nearby triangles as the triangulation is refined. Diffusion does not occur across segments, so attributes used to identify segment-bounded regions remain intact. In output .ele files, all triangles have three points each unless the -o2 switch is used, in which case they have six, and the fourth, fifth, and sixth points lie on the midpoints of the edges opposite the first, second, and third corners.

.poly files:
First line: <# of points> <dimension (must be 2)> <# of attributes>
<# of boundary markers (0 or 1)>
Following lines: <point #> <x> <y> [attributes] [boundary marker]
One line: <# of segments> <# of boundary markers (0 or 1)>
Following lines: <segment #> <endpoint> <endpoint> [boundary marker]
One line: <# of holes>
Following lines: <hole #> <x> <y>
Optional line: <# of regional attributes and/or area constraints>
Optional following lines: <constraint #> <x> <y> <attrib> <max area>

A .poly file represents a PSLG, as well as some additional information. The first section lists all the points, and is identical to the format of .node files. <# of points> may be set to zero to indicate that the points are listed in a separate .node file; .poly files produced by Triangle always have this format. This has the advantage that a point

set may easily be triangulated with or without segments. (The same effect can be achieved, albeit using more disk space, by making a copy of the .poly file with the extension .node; all sections of the file but the first are ignored.)

The second section lists the segments. Segments are edges whose presence in the triangulation is enforced. Each segment is specified by listing the indices of its two endpoints. This means that you must include its endpoints in the point list. If -s, -q, and -a are not selected, Triangle will produce a constrained Delaunay triangulation, in which each segment appears as a single edge in the triangulation. If -q or -a is selected, Triangle will produce a conforming Delaunay triangulation, in which segments may be subdivided into smaller edges. Each segment, like each point, may have a boundary marker.

The third section lists holes (and concavities, if -c is selected) in the triangulation. Holes are specified by identifying a point inside each hole. After the triangulation is formed, Triangle creates holes by eating triangles, spreading out from each hole point until its progress is blocked by PSLG segments; you must be careful to enclose each hole in segments, or your whole triangulation may be eaten away. If the two triangles abutting a segment are eaten, the segment itself is also eaten. Do not place a hole directly on a segment; if you do, Triangle will choose one side of the segment arbitrarily.

The optional fourth section lists regional attributes (to be assigned to all triangles in a region) and regional constraints on the maximum triangle area. Triangle will read this section only if the -A switch is used or the -a switch is used without a number following it, and the -r switch is not used. Regional attributes and area constraints are propagated in the same manner as holes; you specify a point for each attribute and/or constraint, and the attribute and/or constraint will affect the whole region (bounded by segments) containing the point. If two values are written on a line after the x and y coordinate, the former is assumed to be a regional attribute (but will only be applied if the -A switch is selected), and the latter is assumed to be a regional area constraint (but will only be applied if the -a switch is selected). You may also specify just one value after the coordinates, which can serve as both an attribute and an area constraint, depending on the choice of switches. If you are using the -A and -a switches simultaneously and wish to assign an attribute to some region without imposing an area constraint, use a negative maximum area.

When a triangulation is created from a .poly file, you must either enclose the entire region to be triangulated in PSLG segments, or use the -c switch, which encloses the convex hull of the input point set. If you do not use the -c switch, Triangle will eat all triangles on the outer boundary that are not protected by segments; if you are not careful, your whole triangulation may be eaten away. If you do use the -c switch, you can still produce concavities by appropriate placement of holes just inside the convex hull.

An ideal PSLG has no intersecting segments, nor any points that lie upon segments (except, of course, the endpoints of each segment.) You aren't required to make your .poly files ideal, but you should be aware of what can go wrong. Segment intersections are relatively safe - Triangle will calculate the intersection points for you and add them to the triangulation - as long as your machine's floating-point precision doesn't become a problem. You are tempting the fates if you have three segments that cross at the same location, and expect Triangle to figure out where the intersection point is. Thanks to floating-point roundoff error, Triangle will probably decide that the three segments intersect at three different points, and you will find a minuscule triangle in your output - unless Triangle tries to refine the tiny triangle, uses up the last bit of machine precision, and fails to terminate at all.

You're better off putting the intersection point in the input files, and manually breaking up each segment into two. Similarly, if you place a point at the middle of a segment, and hope that Triangle will break up the segment at that point, you might get lucky. On the other hand, Triangle might decide that the point doesn't lie precisely on the line, and you'll have a needle-sharp triangle in your output - or a lot of tiny triangles if you're generating a quality mesh.

When Triangle reads a .poly file, it also writes a .poly file, which includes all edges that are part of input segments. If the -c switch is used, the output .poly file will also include all of the edges on the convex hull. Hence, the output .poly file is useful for finding edges associated with input segments and setting boundary conditions in finite element simulations. More importantly, you will need it if you plan to refine the output mesh, and don't want segments to be missing in later triangulations.

.area files:
First line: <# of triangles>
Following lines: <triangle #> <maximum area>

An .area file associates with each triangle a maximum area that is used for mesh refinement. As with other file formats, every triangle must be represented, and they must be numbered consecutively. A triangle may be left unconstrained by assigning it a negative maximum area.

.edge files:
First line: <# of edges> <# of boundary markers (0 or 1)>
Following lines: <edge #> <endpoint> <endpoint> [boundary marker]

Endpoints are indices into the corresponding .node file. Triangle can produce .edge files (use the -e switch), but cannot read them. The optional column of boundary markers is suppressed by the -B switch.

In Voronoi diagrams, one also finds a special kind of edge that is an infinite ray with only one endpoint. For these edges, a different format is used:

<edge #> <endpoint> -1 <direction x> <direction y>

The 'direction' is a floating-point vector that indicates the direction of the infinite ray.

.neigh files:
First line: <# of triangles> <# of neighbors per triangle (always 3)>
Following lines: <triangle #> <neighbor> <neighbor> <neighbor>

Neighbors are indices into the corresponding .ele file. An index of -1 indicates a mesh boundary, and therefore no neighbor. Triangle can produce .neigh files (use the -n switch), but cannot read them.

The first neighbor of triangle i is opposite the first corner of triangle i, and so on.

Boundary Markers:

Boundary markers are tags used mainly to identify which output points and edges are associated with which PSLG segment, and to identify which points and edges occur on a boundary of the triangulation. A common use is to determine where boundary conditions should be applied to a finite element mesh. You can prevent boundary markers from being written into files produced by Triangle by using the -B switch.

The boundary marker associated with each segment in an output .poly file or edge in an output .edge file is chosen as follows:

- If an output edge is part or all of a PSLG segment with a nonzero boundary marker, then the edge is assigned the same marker.
- Otherwise, if the edge occurs on a boundary of the triangulation (including boundaries of holes), then the edge is assigned the marker one (1).
- Otherwise, the edge is assigned the marker zero (0).

The boundary marker associated with each point in an output .node file is chosen as follows:

- If a point is assigned a nonzero boundary marker in the input file, then it is assigned the same marker in the output .node file.
- Otherwise, if the point lies on a PSLG segment (including the segment's endpoints) with a nonzero boundary marker, then the point is assigned the same marker. If the point lies on several such segments, one of the markers is chosen arbitrarily.
- Otherwise, if the point occurs on a boundary of the triangulation, then the point is assigned the marker one (1).
- Otherwise, the point is assigned the marker zero (0).

If you want Triangle to determine for you which points and edges are on the boundary, assign them the boundary marker zero (or use no markers at all) in your input files. Alternatively, you can mark some of them and leave others marked zero, allowing Triangle to label them.

Triangulation Iteration Numbers:

Because Triangle can read and refine its own triangulations, input and output files have iteration numbers. For instance, Triangle might read the files mesh.3.node, mesh.3.ele, and mesh.3.poly, refine the triangulation, and output the files mesh.4.node, mesh.4.ele, and mesh.4.poly. Files with no iteration number are treated as if their iteration number is zero; hence, Triangle might read the file points.node, triangulate it, and produce the files points.1.node and points.1.ele.

Iteration numbers allow you to create a sequence of successively finer meshes suitable for multigrid methods. They also allow you to produce a sequence of meshes using error estimate-driven mesh refinement.

If you're not using refinement or quality meshing, and you don't like iteration numbers, use the -I switch to disable them. This switch will also disable output of .node and .poly files to prevent your input files from being overwritten. (If the input is a .poly file that contains its own points, a .node file will be written.)

Examples of How to Use Triangle:

'triangle dots' will read points from dots.node, and write their Delaunay triangulation to dots.1.node and dots.1.ele. (dots.1.node will be identical to dots.node.) 'triangle -I dots' writes the triangulation to dots.ele instead. (No additional .node file is needed, so none is written.)

'triangle -pe object.1' will read a PSLG from object.1.poly (and possibly object.1.node, if the points are omitted from object.1.poly) and write their constrained Delaunay triangulation to object.2.node and object.2.ele. The segments will be copied to object.2.poly, and all edges will be written to object.2.edge.

'triangle -pq31.5a.1 object' will read a PSLG from object.poly (and possibly object.node), generate a mesh whose angles are all greater than 31.5 degrees and whose triangles all have area smaller than 0.1, and write the mesh to object.1.node and object.1.ele. Each segment may have been broken up into multiple edges; the resulting constrained edges are written to object.1.poly.

Here is a sample file 'box.poly' describing a square with a square hole:

```
# A box with eight points in 2D, no attributes, one boundary marker.
8 2 0 1
# Outer box has these vertices:
 1   0 0   0
 2   0 3   0
 3   3 0   0
 4   3 3   33     # A special marker for this point.
# Inner square has these vertices:
 5   1 1   0
 6   1 2   0
 7   2 1   0
 8   2 2   0
# Five segments with boundary markers.
5 1
 1   1 2   5      # Left side of outer box.
 2   5 7   0      # Segments 2 through 5 enclose the hole.
 3   7 8   0
 4   8 6   10
 5   6 5   0
# One hole in the middle of the inner square.
1
 1   1.5 1.5
```

Note that some segments are missing from the outer square, so one must use the '-c' switch. After 'triangle -pqc box.poly', here is the output file 'box.1.node', with twelve points. The last four points were added to meet the angle constraint. Points 1, 2, and 9 have markers from segment 1. Points 6 and 8 have markers from segment 4. All the other points but 4 have been marked to indicate that they lie on a boundary.

```
12  2  0  1
   1    0   0      5
   2    0   3      5
   3    3   0      1
   4    3   3     33
   5    1   1      1
   6    1   2     10
   7    2   1      1
   8    2   2     10
   9    0   1.5    5
  10    1.5   0    1
  11    3   1.5    1
  12    1.5   3    1
# Generated by triangle -pqc box.poly
```

Here is the output file 'box.1.ele', with twelve triangles.

```
12  3  0
   1     5   6   9
   2    10   3   7
   3     6   8  12
   4     9   1   5
   5     6   2   9
   6     7   3  11
   7    11   4   8
   8     7   5  10
   9    12   2   6
  10     8   7  11
  11     5   1  10
  12     8   4  12
# Generated by triangle -pqc box.poly
```

Here is the output file 'box.1.poly'. Note that segments have been added

to represent the convex hull, and some segments have been split by newly added points. Note also that <# of points> is set to zero to indicate that the points should be read from the .node file.

```
  0  2  0  1
  12  1
     1      1   9      5
     2      5   7      1
     3      8   7      1
     4      6   8     10
     5      5   6      1
     6      3  10      1
     7      4  11      1
     8      2  12      1
     9      9   2      5
    10     10   1      1
    11     11   3      1
    12     12   4      1
  1
     1   1.5 1.5
  # Generated by triangle -pqc box.poly
```

Refinement and Area Constraints:

The -r switch causes a mesh (.node and .ele files) to be read and refined. If the -p switch is also used, a .poly file is read and used to specify edges that are constrained and cannot be eliminated (although they can be divided into smaller edges) by the refinement process.

When you refine a mesh, you generally want to impose tighter quality constraints. One way to accomplish this is to use -q with a larger angle, or -a followed by a smaller area than you used to generate the mesh you are refining. Another way to do this is to create an .area file, which specifies a maximum area for each triangle, and use the -a switch (without a number following). Each triangle's area constraint is applied to that triangle. Area constraints tend to diffuse as the mesh is refined, so if there are large variations in area constraint between adjacent triangles, you may not get the results you want.

If you are refining a mesh composed of linear (three-node) elements, the output mesh will contain all the nodes present in the input mesh, in the same order, with new nodes added at the end of the .node file. However, there is no guarantee that each output element is contained in a single input element. Often, output elements will overlap two input elements, and input edges are not present in the output mesh. Hence, a sequence of refined meshes will form a hierarchy of nodes, but not a hierarchy of elements. If you a refining a mesh of higher-order elements, the hierarchical property applies only to the nodes at the corners of an element; other nodes may not be present in the refined mesh.

It is important to understand that maximum area constraints in .poly files are handled differently from those in .area files. A maximum area in a .poly file applies to the whole (segment-bounded) region in which a point falls, whereas a maximum area in an .area file applies to only one triangle. Area constraints in .poly files are used only when a mesh is first generated, whereas area constraints in .area files are used only to refine an existing mesh, and are typically based on a posteriori error estimates resulting from a finite element simulation on that mesh.

'triangle -rq25 object.1' will read object.1.node and object.1.ele, then refine the triangulation to enforce a 25 degree minimum angle, and then write the refined triangulation to object.2.node and object.2.ele.

'triangle -rpaa6.2 z.3' will read z.3.node, z.3.ele, z.3.poly, and z.3.area. After reconstructing the mesh and its segments, Triangle will

refine the mesh so that no triangle has area greater than 6.2, and furthermore the triangles satisfy the maximum area constraints in z.3.area. The output is written to z.4.node, z.4.ele, and z.4.poly.

The sequence 'triangle -qa1 x', 'triangle -rqa.3 x.1', 'triangle -rqa.1 x.2' creates a sequence of successively finer meshes x.1, x.2, and x.3, suitable for multigrid.

Convex Hulls and Mesh Boundaries:

If the input is a point set (rather than a PSLG), Triangle produces its convex hull as a by-product in the output .poly file if you use the -c switch. There are faster algorithms for finding a two-dimensional convex hull than triangulation, of course, but this one comes for free. If the input is an unconstrained mesh (you are using the -r switch but not the -p switch), Triangle produces a list of its boundary edges (including hole boundaries) as a by-product if you use the -c switch.

Voronoi Diagrams:

The -v switch produces a Voronoi diagram, in files suffixed .v.node and .v.edge. For example, 'triangle -v points' will read points.node, produce its Delaunay triangulation in points.1.node and points.1.ele, and produce its Voronoi diagram in points.1.v.node and points.1.v.edge. The .v.node file contains a list of all Voronoi vertices, and the .v.edge file contains a list of all Voronoi edges, some of which may be infinite rays. (The choice of filenames makes it easy to run the set of Voronoi vertices through Triangle, if so desired.)

This implementation does not use exact arithmetic to compute the Voronoi vertices, and does not check whether neighboring vertices are identical. Be forewarned that if the Delaunay triangulation is degenerate or near-degenerate, the Voronoi diagram may have duplicate points, crossing edges, or infinite rays whose direction vector is zero. Also, if you generate a constrained (as opposed to conforming) Delaunay triangulation, or if the triangulation has holes, the corresponding Voronoi diagram is likely to have crossing edges and unlikely to make sense.

Mesh Topology:

You may wish to know which triangles are adjacent to a certain Delaunay edge in an .edge file, which Voronoi regions are adjacent to a certain Voronoi edge in a .v.edge file, or which Voronoi regions are adjacent to each other. All of this information can be found by cross-referencing output files with the recollection that the Delaunay triangulation and the Voronoi diagrams are planar duals.

Specifically, edge i of an .edge file is the dual of Voronoi edge i of the corresponding .v.edge file, and is rotated 90 degrees counterclockwise from the Voronoi edge. Triangle j of an .ele file is the dual of vertex j of the corresponding .v.node file; and Voronoi region k is the dual of point k of the corresponding .node file.

Hence, to find the triangles adjacent to a Delaunay edge, look at the vertices of the corresponding Voronoi edge; their dual triangles are on the left and right of the Delaunay edge, respectively. To find the Voronoi regions adjacent to a Voronoi edge, look at the endpoints of the corresponding Delaunay edge; their dual regions are on the right and left of the Voronoi edge, respectively. To find which Voronoi regions are adjacent to each other, just read the list of Delaunay edges.

Statistics:

After generating a mesh, Triangle prints a count of the number of points, triangles, edges, boundary edges, and segments in the output mesh. If

you've forgotten the statistics for an existing mesh, the -rNEP switches (or -rpNEP if you've got a .poly file for the existing mesh) will regenerate these statistics without writing any output.

The -V switch produces extended statistics, including a rough estimate of memory use and a histogram of triangle aspect ratios and angles in the mesh.

Exact Arithmetic:

Triangle uses adaptive exact arithmetic to perform what computational geometers call the 'orientation' and 'incircle' tests. If the floating-point arithmetic of your machine conforms to the IEEE 754 standard (as most workstations do), and does not use extended precision internal registers, then your output is guaranteed to be an absolutely true Delaunay or conforming Delaunay triangulation, roundoff error notwithstanding. The word 'adaptive' implies that these arithmetic routines compute the result only to the precision necessary to guarantee correctness, so they are usually nearly as fast as their approximate counterparts. The exact tests can be disabled with the -X switch. On most inputs, this switch will reduce the computation time by about eight percent - it's not worth the risk. There are rare difficult inputs (having many collinear and cocircular points), however, for which the difference could be a factor of two. These are precisely the inputs most likely to cause errors if you use the -X switch.

Unfortunately, these routines don't solve every numerical problem. Exact arithmetic is not used to compute the positions of points, because the bit complexity of point coordinates would grow without bound. Hence, segment intersections aren't computed exactly; in very unusual cases, roundoff error in computing an intersection point might actually lead to an inverted triangle and an invalid triangulation. (This is one reason to compute your own intersection points in your .poly files.) Similarly, exact arithmetic is not used to compute the vertices of the Voronoi diagram.

Underflow and overflow can also cause difficulties; the exact arithmetic routines do not ameliorate out-of-bounds exponents, which can arise during the orientation and incircle tests. As a rule of thumb, you should ensure that your input values are within a range such that their third powers can be taken without underflow or overflow. Underflow can silently prevent the tests from being performed exactly, while overflow will typically cause a floating exception.

Calling Triangle from Another Program:

Read the file triangle.h for details.

Troubleshooting:

Please read this section before mailing me bugs.

'My output mesh has no triangles!'

If you're using a PSLG, you've probably failed to specify a proper set of bounding segments, or forgotten to use the -c switch. Or you may have placed a hole badly. To test these possibilities, try again with the -c and -O switches. Alternatively, all your input points may be collinear, in which case you can hardly expect to triangulate them.

'Triangle doesn't terminate, or just crashes.'

Bad things can happen when triangles get so small that the distance between their vertices isn't much larger than the precision of your machine's arithmetic. If you've compiled Triangle for single-precision

arithmetic, you might do better by recompiling it for double-precision. Then again, you might just have to settle for more lenient constraints on the minimum angle and the maximum area than you had planned.

You can minimize precision problems by ensuring that the origin lies inside your point set, or even inside the densest part of your mesh. On the other hand, if you're triangulating an object whose x coordinates all fall between 6247133 and 6247134, you're not leaving much floating-point precision for Triangle to work with.

Precision problems can occur covertly if the input PSLG contains two segments that meet (or intersect) at a very small angle, or if such an angle is introduced by the -c switch, which may occur if a point lies ever-so-slightly inside the convex hull, and is connected by a PSLG segment to a point on the convex hull. If you don't realize that a small angle is being formed, you might never discover why Triangle is crashing. To check for this possibility, use the -S switch (with an appropriate limit on the number of Steiner points, found by trial-and-error) to stop Triangle early, and view the output .poly file with Show Me (described below). Look carefully for small angles between segments; zoom in closely, as such segments might look like a single segment from a distance.

If some of the input values are too large, Triangle may suffer a floating exception due to overflow when attempting to perform an orientation or incircle test. (Read the section on exact arithmetic above.) Again, I recommend compiling Triangle for double (rather than single) precision arithmetic.

'The numbering of the output points doesn't match the input points.'

You may have eaten some of your input points with a hole, or by placing them outside the area enclosed by segments.

'Triangle executes without incident, but when I look at the resulting mesh, it has overlapping triangles or other geometric inconsistencies.'

If you select the -X switch, Triangle's divide-and-conquer Delaunay triangulation algorithm occasionally makes mistakes due to floating-point roundoff error. Although these errors are rare, don't use the -X switch. If you still have problems, please report the bug.

Strange things can happen if you've taken liberties with your PSLG. Do you have a point lying in the middle of a segment? Triangle sometimes copes poorly with that sort of thing. Do you want to lay out a collinear row of evenly spaced, segment-connected points? Have you simply defined one long segment connecting the leftmost point to the rightmost point, and a bunch of points lying along it? This method occasionally works, especially with horizontal and vertical lines, but often it doesn't, and you'll have to connect each adjacent pair of points with a separate segment. If you don't like it, tough.

Furthermore, if you have segments that intersect other than at their endpoints, try not to let the intersections fall extremely close to PSLG points or each other.

If you have problems refining a triangulation not produced by Triangle: Are you sure the triangulation is geometrically valid? Is it formatted correctly for Triangle? Are the triangles all listed so the first three points are their corners in counterclockwise order?

Show Me:

Triangle comes with a separate program named 'Show Me', whose primary purpose is to draw meshes on your screen or in PostScript. Its secondary

purpose is to check the validity of your input files, and do so more thoroughly than Triangle does. Show Me requires that you have the X Windows system. If you didn't receive Show Me with Triangle, complain to whomever you obtained Triangle from, then send me mail.

Triangle on the Web:

To see an illustrated, updated version of these instructions, check out

http://www.cs.cmu.edu/~quake/triangle.html

A Brief Plea:

If you use Triangle, and especially if you use it to accomplish real work, I would like very much to hear from you. A short letter or email (to jrs@cs.cmu.edu) describing how you use Triangle will mean a lot to me. The more people I know are using this program, the more easily I can justify spending time on improvements and on the three-dimensional successor to Triangle, which in turn will benefit you. Also, I can put you on a list to receive email whenever a new version of Triangle is available.

If you use a mesh generated by Triangle in a publication, please include an acknowledgment as well.

Research credit:

Of course, I can take credit for only a fraction of the ideas that made this mesh generator possible. Triangle owes its existence to the efforts of many fine computational geometers and other researchers, including Marshall Bern, L. Paul Chew, Boris Delaunay, Rex A. Dwyer, David Eppstein, Steven Fortune, Leonidas J. Guibas, Donald E. Knuth, C. L. Lawson, Der-Tsai Lee, Ernst P. Mucke, Douglas M. Priest, Jim Ruppert, Isaac Saias, Bruce J. Schachter, Micha Sharir, Jorge Stolfi, Christopher J. Van Wyk, David F. Watson, and Binhai Zhu. See the comments at the beginning of the source code for references.

Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator

Jonathan Richard Shewchuk
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
jrs@cs.cmu.edu

1 Introduction

Triangle is a C program for two-dimensional mesh generation and construction of Delaunay triangulations, constrained Delaunay triangulations, and Voronoï diagrams. Triangle is fast, memory-efficient, and robust; it computes Delaunay triangulations and constrained Delaunay triangulations exactly. Guaranteed-quality meshes (having no small angles) are generated using Ruppert's Delaunay refinement algorithm. Features include user-specified constraints on angles and triangle areas, user-specified holes and concavities, and the economical use of exact arithmetic to improve robustness. Triangle is freely available on the Web at "http://www.cs.cmu.edu/~quake/triangle.html" and from Netlib. This paper discusses many of the key implementation decisions, including the choice of triangulation algorithms and data structures, the steps taken to create and refine a mesh, a number of issues that arise in Ruppert's algorithm, and the use of exact arithmetic.

2 Triangulation Algorithms and Data Structures

A triangular mesh generator rests on the efficiency of its triangulation algorithms and data structures, so I discuss these first. I assume the reader is familiar with Delaunay triangulations, constrained Delaunay triangulations, and the incremental insertion algorithms for constructing them. Consult the survey by Bern and Eppstein [2] for an introduction.

There are many Delaunay triangulation algorithms, some of which are surveyed and evaluated by Fortune [7] and Su and Drysdale [18]. Their results indicate a rough parity in speed among the incremental insertion algorithm of Lawson [11], the divide-and-conquer algorithm of Lee and Schachter [12], and the plane-sweep algorithm of Fortune [6]; however, the implementations they study were written by different people. I believe that Triangle is the first instance in which all three algorithms have been implemented with the same data structures and floating-point tests, by one person who gave roughly equal attention to optimizing each. (Some details of how these implementations were optimized appear in Appendix A.)

Table 1 compares the algorithms, including versions that use exact arithmetic (see Section 4) to achieve robustness, and versions that use approximate arithmetic and are hence faster but may fail or produce incorrect output. (The robust and non-robust versions are otherwise identical.) As Su and Drysdale [18] also found, the divide-and-conquer algorithm is fastest, with the sweepline algorithm second. The incremental algorithm performs poorly, spending most of its time in point location. (Su and Drysdale produced a better incremental insertion implementation by using bucketing to perform point location, but it still ranks third. Triangle does not use bucketing because it is easily defeated, as discussed in the appendix.) The agreement between my results and those of Su and Drysdale lends support to their ranking of algorithms.

An important optimization to the divide-and-conquer algorithm, adapted from Dwyer [5], is to partition the vertices with alternating horizontal and vertical cuts (Lee and Schachter's algorithm uses only vertical cuts). Alternating cuts speed the algorithm and, when exact arithmetic is disabled, reduce its likelihood of failure. One million points can be triangulated correctly in a minute on a fast workstation.

All three triangulation algorithms are implemented so as to eliminate duplicate input points; if not eliminated, duplicates can cause catastrophic failures. The sweepline algorithm can easily detect duplicate points as they are removed from the event queue (by comparing each with the previous point removed from the queue), and the incremental insertion algorithm can detect a duplicate point after point location. The divide-and-conquer algorithm begins by sorting the points by their x-coordinates, after which duplicates can be detected and removed. This sorting step is a necessary part of the divide-and-conquer algorithm with vertical cuts, but not of the variant with alternating cuts (which must perform a sequence of median-finding operations, alternately by x and

Supported in part by the Natural Sciences and Engineering Research Council of Canada under a 1967 Science and Engineering Scholarship and by the National Science Foundation under Grant ASC-9318163.

Delaunay triangulation timings (seconds)									
Number of points	10,000			100,000			1,000,000		
Point distribution / Algorithm	Uniform Random	Boundary of Circle	Tilted Grid	Uniform Random	Boundary of Circle	Tilted Grid	Uniform Random	Boundary of Circle	Tilted Grid
Div&Conq, alternating cuts									
robust	0.33	0.57	0.72	4.5	5.3	5.5	58	61	58
non-robust	0.30	0.27	0.27	4.0	4.0	3.5	53	56	44
Div&Conq, vertical cuts									
robust	0.47	1.06	0.96	6.2	9.0	7.6	79	98	85
non-robust	0.36	0.17	failed	5.0	2.1	4.2	64	26	failed
Sweepline									
non-robust	0.78	0.62	0.71	10.8	8.6	10.5	147	119	139
Incremental									
robust	1.15	3.88	2.79	24.0	112.7	101.3	545	1523	2138
non-robust	0.99	2.74	failed	21.3	94.3	failed	486	1327	failed

Table 1: Timings for triangulation on a DEC 3000/700 with a 225 MHz Alpha processor, not including I/O. Robust and non-robust versions of the Delaunay algorithms triangulated points chosen from one of three different distributions: uniformly distributed random points in a square, random approximately cocircular points, and a tilted 1000 × 1000 square grid.

y-coordinates). Hence, the timings in Table 1 for divide-and-conquer with alternating cuts could be improved slightly if one could guarantee that no duplicate input points would occur; the initial sorting step would be unnecessary.

Should one choose a data structure that uses a record to represent each edge, or one that uses a record to represent each triangle? Triangle was originally written using Guibas and Stolfi's *quad-edge* data structure [10] (without the *Flip* operator), then rewritten using a triangle-based data structure. The quad-edge data structure is popular because it is elegant, because it simultaneously represents a graph and its geometric dual (such as a Delaunay triangulation and the corresponding Voronoï diagram), and because Guibas and Stolfi give detailed pseudocode for implementing the divide-and-conquer and incremental Delaunay algorithms using quad-edges.

Despite the fundamental differences between the data structures, the quad-edge-based and triangle-based implementations of Triangle are both faithful to the Delaunay triangulation algorithms presented by Guibas and Stolfi [10] (I did not implement a quad-edge sweepline algorithm), and hence offer a fair comparison of the data structures. Perhaps the most useful observation of this paper for practitioners is that the divide-and-conquer algorithm, the incremental algorithm, and the Delaunay refinement algorithm for mesh generation were all sped by a factor of two by the triangular data structure. (However, it is worth noting that the code devoted specifically to triangulation is roughly twice as long for the triangular data structure.) A difference so pronounced demands explanation.

First, consider the different storage demands of each data structure, illustrated in Figure 1. Each quad-edge record contains four pointers to neighboring quad-edges, and two pointers to vertices (the endpoints of the edge). Each triangle record contains three pointers to neighboring triangles, and three pointers to vertices. Hence, both structures contain six pointers.[1] A triangulation contains roughly three edges for every two triangles. Hence, the triangular data structure is more space-efficient.

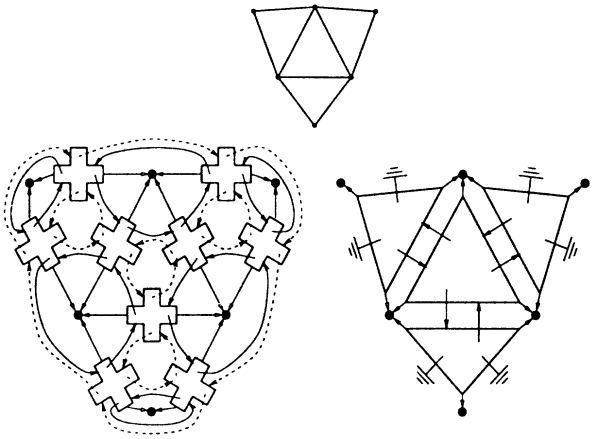


Figure 1: A triangulation (top) and its corresponding representations with quad-edge and triangular data structures. Each quad-edge and each triangle contains six pointers.

It is difficult to ascertain with certainty why the triangular data structure is superior in time as well as space, but one can make educated inferences. When a program makes structural changes to a triangulation, the amount of time used depends in part on the number of pointers that have to be read and written.

[1] Both the quad-edge and triangle data structures must store not only pointers to their neighbors, but also the *orientations* of their neighbors, to make clear how they are connected. For instance, each pointer from a triangle to a neighboring triangle has an associated orientation (a number between zero and two) that indicates which edge of the neighboring triangle is contacted. An important space optimization is to store the orientation of each quad-edge or triangle in the bottom two bits of the corresponding pointer. Thus, each record must be aligned on a four-byte boundary.

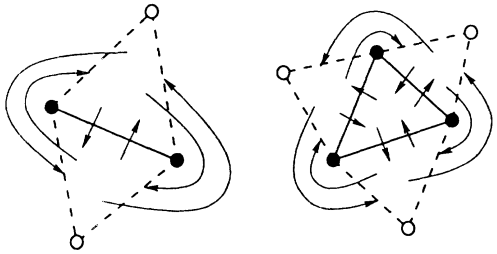


Figure 2: How the triangle-based divide-and-conquer algorithm represents an isolated edge (left) and an isolated triangle (right). Dashed lines represent ghost triangles. White vertices all represent the same "vertex at infinity"; only black vertices have coordinates.

This amount is smaller for the triangular data structure; more of the connectivity information is implicit in each triangle. Caching is improved by the fact that fewer structures are accessed. (For large triangulations, any two adjoining quad-edges or triangles are unlikely to lie in the same cache line.)

Because the triangle-based divide-and-conquer algorithm proved to be fastest, it is worth exploring in some depth. At first glance, the algorithm and data structure seem incompatible. The divide-and-conquer algorithm recursively halves the input vertices until they are partitioned into subsets of two or three vertices each. Each subset is easily triangulated (yielding an edge, two collinear edges, or a triangle), and the triangulations are merged together to form larger ones. If one uses a degenerate triangle to represent an isolated edge, the resulting code is clumsy because of the need to handle special cases. One might partition the input into subsets of three to five vertices, but this does not help if the points in a subset are collinear.

To preserve the elegance of Guibas and Stolfi's presentation of the divide-and-conquer algorithm, each triangulation is surrounded with a layer of "ghost" triangles, one triangle per convex hull edge. The ghost triangles are connected to each other in a ring about a "vertex at infinity" (really just a null pointer). A single edge is represented by two ghost triangles, as illustrated in Figure 2.

Ghost triangles are useful for efficiently traversing the convex hull edges during the merge step. Some are transformed into real triangles during this step; two triangulations are sewn together by fitting their ghost triangles together like the teeth of two gears. (Some edge flips are also needed. See Figure 3.) Each merge step creates only two new triangles; one at the bottom and one at the top of the seam. After all the merge steps are done, the ghost triangles are removed and the triangulation is passed on to the next stage of meshing.

Precisely the same data structure, ghost triangles and all, is used in the sweepline implementation to represent the growing triangulation (which often includes dangling edges). Details are omitted.

Augmentations to the data structure are necessary to support the constrained triangulations needed for mesh generation. Constrained edges are edges that may not be removed in the process of improving the quality of a mesh, and hence may not be flipped during incremental insertion of a vertex. One or more constrained edges collectively represent an input segment. Constrained edges may carry additional information, such as boundary conditions for finite element simulations. (A future version of Triangle may support curved segments this way.) The quad-edge structure supports such constraints easily; each quad-edge is simply annotated to mark the fact that it is constrained, and perhaps annotated with extra information. It is more expensive to represent constraints with the triangular structure; I augment each triangle with three extra pointers (one for each edge), which are usually null but may point to *shell edges*, which represent constrained edges and carry additional information. This eliminates the space advantage of the triangular data structure, but not its time advantage. Triangle uses the longer record only if constraints are needed.

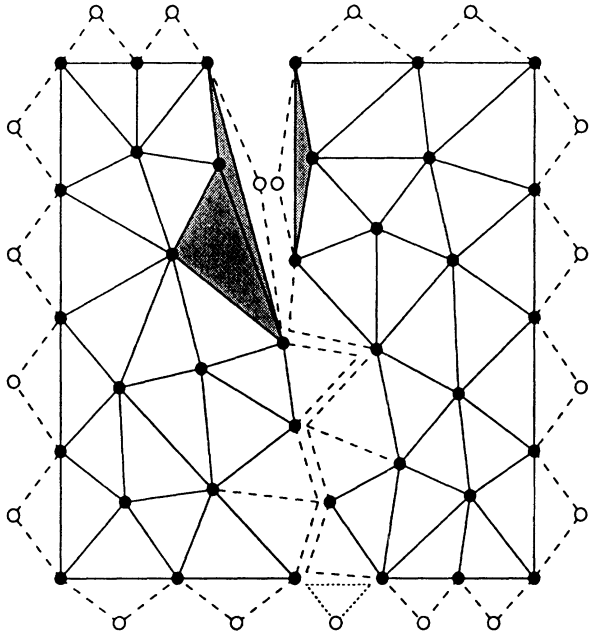


Figure 3: Halfway through a merge step of the divide-and-conquer algorithm. Dashed lines represent ghost triangles and triangles displaced by edge flips. The dotted triangle at bottom center is a newly created ghost triangle. Shaded triangles are non-Delaunay and will be displaced by edge flips.

3 Ruppert's Delaunay Refinement Algorithm

Ruppert's algorithm for two-dimensional quality mesh generation [15] is perhaps the first theoretically guaranteed meshing algorithm to be truly satisfactory in practice. It produces meshes with no small angles, using relatively few triangles (though the density of triangles can be increased under user control) and allowing the density of triangles to vary quickly over short distances, as illustrated in Figure 4. (Chew [3] independently developed a similar algo-

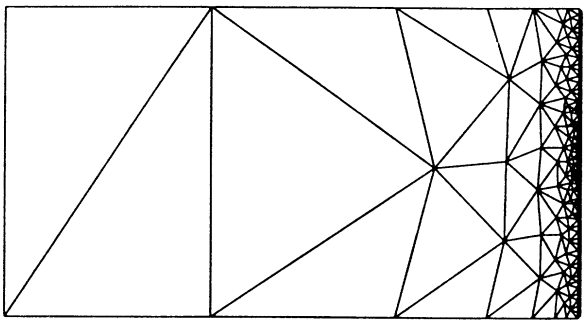


Figure 4: A demonstration of the ability of the Delaunay refinement algorithm to achieve large gradations in triangle size while constraining angles. No angles are smaller than 24°.

rithm.) This section describes Ruppert's Delaunay refinement algorithm as it is implemented in Triangle.

Triangle's input is a *planar straight line graph* (PSLG), defined to be a collection of vertices and segments (where the endpoints of every segment are included in the list of vertices). Figure 5 illustrates a PSLG defining an electric guitar. Although the definition of "PSLG" normally disallows segment intersections (except at segment endpoints), Triangle can detect segment intersections and insert vertices.

The first stage of the algorithm is to find the Delaunay triangulation of the input vertices, as in Figure 6. In general, some of the input segments are missing from the triangulation; the second stage is to insert them. Triangle can force the mesh to conform to the segments in one of two ways, selectable by the user. The first is to insert a new vertex corresponding to the midpoint of any segment that does not appear in the mesh, and use Lawson's incremental insertion algorithm to maintain the Delaunay property. The effect is to split the segment in half, and the two resulting subsegments may appear in the mesh. If not, the procedure is repeated recursively until the original segment is represented by a linear sequence of constrained edges in the mesh.

The second choice is to simply use a constrained Delaunay triangulation (Figure 7). Each segment is inserted by deleting the triangles it overlaps, and retriangulating the regions on each side of the segment. No new vertices are inserted. For reasons explained in Section 3.1, Triangle uses the constrained Delaunay triangulation by default.

The third stage of the algorithm, which diverges from Ruppert [15], is to remove triangles from concavities and holes (Figure 8). A hole is simply a user-specified point in the plane where a "triangle-eating virus" is planted and spread by depth-first search until its advance is halted by segments. (This simple mechanism saves both the user and the implementation from a common outlook wherein one must define oriented curves whose insides are clearly distinguishable from their outsides. Triangle's method makes it easier to treat holes and internal boundaries in a unified manner.[2]) Concavities

[2] I imagine computational geometers replying, "Of course," engineers responding, "Hmm," and solid modeling specialists recoiling in horror.

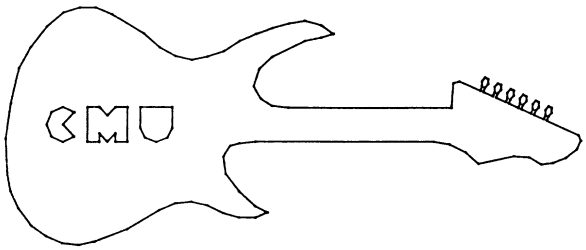


Figure 5: Electric guitar PSLG.

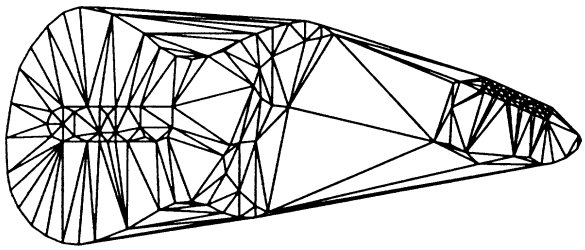


Figure 6: Delaunay triangulation of vertices of PSLG. The triangulation does not conform to all of the input segments.

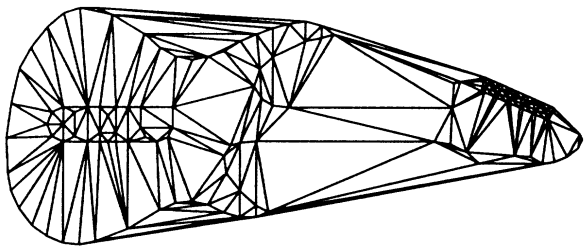


Figure 7: Constrained Delaunay triangulation of PSLG.

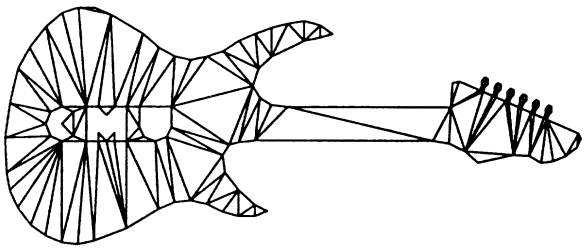


Figure 8: Triangles are removed from concavities and holes.

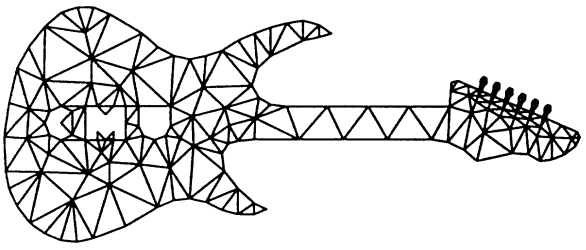


Figure 9: Conforming Delaunay triangulation with 20° minimum angle.

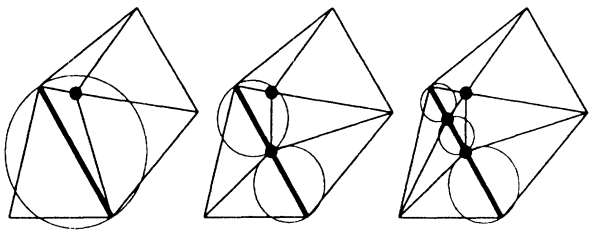


Figure 10: Segments are split recursively (while maintaining the Delaunay property) until no segments are encroached.

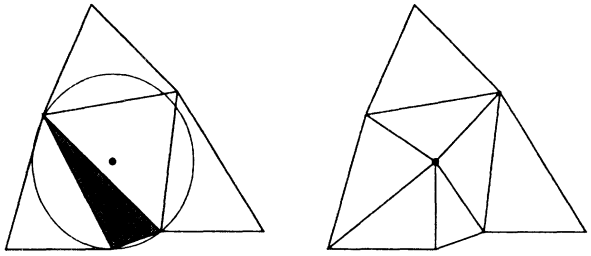


Figure 11: Each bad triangle is split by inserting a vertex at its circumcenter and maintaining the Delaunay property.

are recognized from unconstrained edges on the boundary of the mesh, and the same virus is used to hollow them out.

The fourth stage, and the heart of the algorithm, refines the mesh by inserting additional vertices into the mesh (using Lawson's algorithm to maintain the Delaunay property) until all constraints on minimum angle and maximum triangle area are met (Figure 9). Vertex insertion is governed by two rules.

- The *diametral circle* of a segment is the (unique) smallest circle that contains the segment. A segment is said to be *encroached* if a point lies within its diametral circle. Any encroached segment that arises is immediately split by inserting a vertex at its midpoint. The two resulting subsegments have smaller diametral circles, and may or may not be encroached themselves. See Figure 10.

- The *circumcircle* of a triangle is the unique circle that passes through all three vertices of the triangle. A triangle is said to be *bad* if it has an angle too small or an area too large to satisfy the user's constraints. A bad triangle is split by inserting a vertex at its *circumcenter* (the center of its circumcircle); the Delaunay property guarantees that the triangle is eliminated (see Figure 11). If the new vertex encroaches upon any segment, the vertex is deleted (reversing the insertion process) and all the segments it encroached upon are split.

Encroached segments are given priority over bad triangles. A queue of encroached segments and a queue of bad triangles are initialized at the beginning of the refinement stage and maintained throughout; every vertex insertion may add new members to either queue. The former queue rarely contains more than one segment except at the beginning of the refinement stage, when it may contain many.

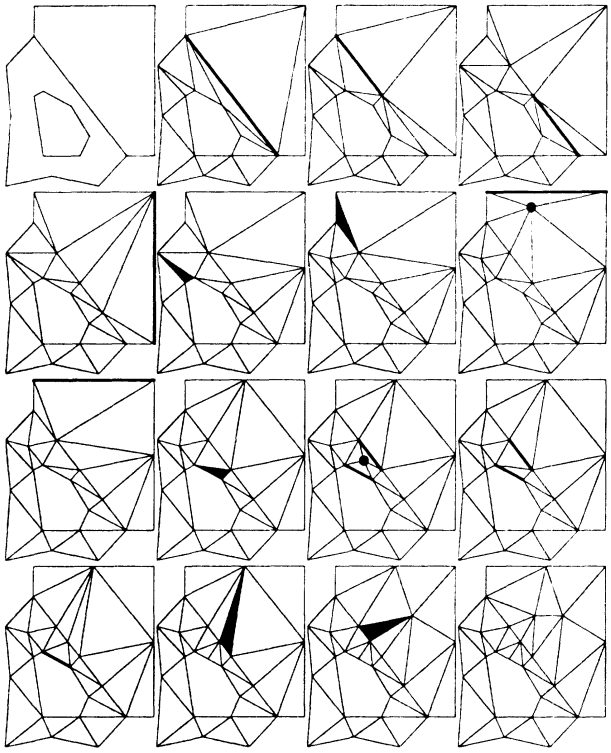


Figure 12: Demonstration of the refinement stage. The first two images are the input PSLG and its constrained Delaunay triangulation. In each image, highlighted segments or triangles are about to be split, and highlighted vertices are about to be deleted. Note that the algorithm easily accommodates internal boundaries and holes.

The refinement stage is illustrated in Figure 12. Ruppert [15] proves that this procedure halts for an angle constraint of up to 20.7°. In practice, the algorithm generally halts with an angle constraint of 33.8°, but often fails to terminate given an angle constraint of 33.9°. It would be interesting to discover why the cutoff falls there.

3.1 Selected Implementation Issues

Triangle removes extraneous triangles from holes and concavities before the refinement stage. This presents no problem for the refinement algorithm; the requirement that no segment be encroached and the Delaunay property together ensure that the circumcenter of every triangle lies within the mesh. (Roundoff error might perturb a circumcenter to just outside the mesh, but it is easy to identify the conflicting edge and treat it as encroached.) An advantage of removing triangles before refinement is that computation is not wasted refining triangles that will eventually be deleted.

A more important advantage is illustrated in Figure 13. If extraneous triangles remain during the refinement stage, overrefinement can occur if very small features outside the object being meshed cause the creation of small triangles inside the mesh. Ruppert suggests solving this problem by using the constrained Delaunay triangulation, and ignoring interactions that take place outside the region being triangulated. Early removal of triangles provides a nearly effortless

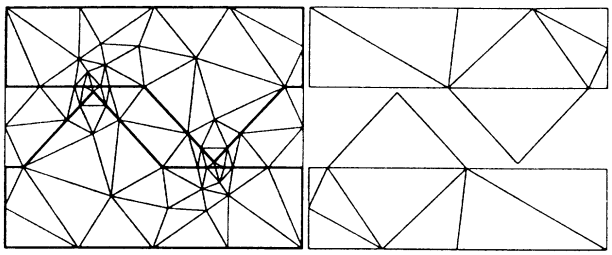


Figure 13: Two variations of the Delaunay refinement algorithm with a 20° minimum angle. Left: Mesh created using segment splitting and late removal of triangles. This illustration includes external triangles, just prior to removal, to show why overrefinement occurs. Right: Mesh created using constrained Delaunay triangulation and early removal of triangles.

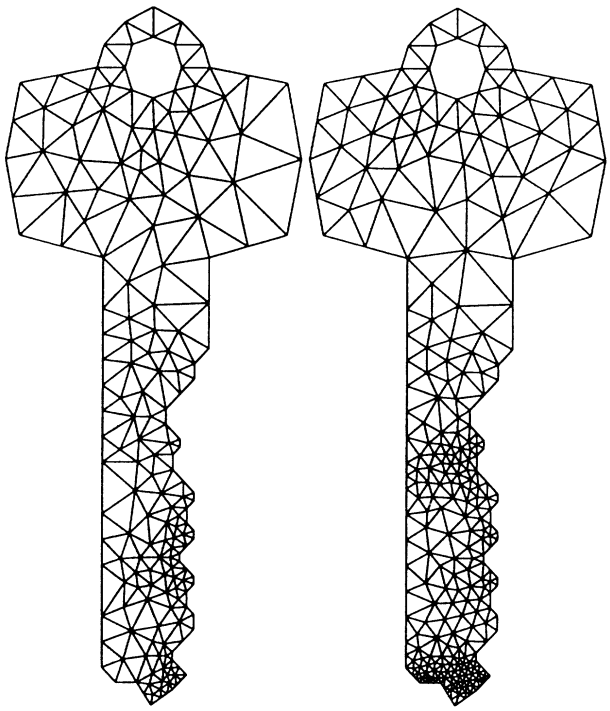


Figure 14: Two meshes with a 33° minimum angle. The left mesh, with 290 triangles, was formed by always splitting the worst existing triangle. The right mesh, with 450 triangles, was formed by using a first-come first-split queue of bad triangles.

way to accomplish this effect. Segments that would normally be considered encroached are ignored (Figure 13, right), because encroached segments are diagnosed by noticing that they occur opposite an obtuse angle in a triangle.

Another determinant of the number of triangles in the final mesh is the order in which bad triangles are split, especially when a strong angle constraint is used. Figure 14 demonstrates how sensitive the refinement algorithm is to the order. For this example with a 33° minimum angle, a heap of bad triangles indexed by their smallest angle confers a 35% reduction in mesh size over a first-in first-out queue. (This difference is typical for large meshes with a strong angle constraint, but thankfully disappears for small meshes and small

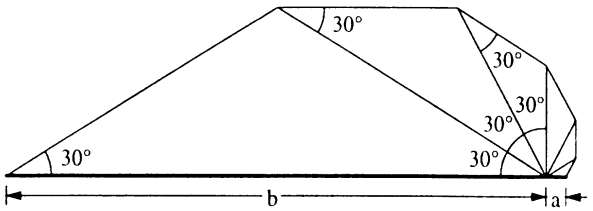


Figure 15: In any triangulation with no angles smaller than 30°, the ratio b/a cannot exceed 27.

constraints.) The discrepancy probably occurs because circumcenters of very bad triangles are likely to split more bad triangles than circumcenters of mildly bad triangles. Unfortunately, a heap is slow for large meshes, especially when small area constraints force all of the triangles into the heap. Delaunay refinement usually takes $\mathcal{O}(n)$ time in practice, but use of a heap increases the complexity to $\mathcal{O}(n \log n)$.

Triangle's solution, chosen experimentally, is to use 64 FIFO queues, each representing a different interval of angles. It is counterproductive (in practice) to order well-shaped triangles by their worst angle, so one queue is used for well-shaped but too-large triangles whose angles are all roughly larger than 39°. Triangles with smaller angles are partitioned among the remaining queues. When a bad triangle is chosen for splitting, it is taken from the "worst" nonempty queue. This method yields meshes comparable with those generated using a heap, but is only slightly slower than using a single queue. During the refinement phase, about 21,000 new vertices are generated per second on a DEC 3000/700. These vertices are inserted using the incremental Delaunay algorithm, but are inserted much more quickly than Table 1 would suggest because a triangle's circumcenter can be located quickly by starting the search at the triangle.

3.2 A Negative Result on Quality Triangulations

For any angle bound $\theta > 0$, there exists a PSLG $\mathcal{P}$ such that it is not possible to triangulate $\mathcal{P}$ without creating a new corner (not present in $\mathcal{P}$) having angle smaller than θ. Here, I discuss why this is true.

Ruppert's proof that his Delaunay refinement algorithm terminates makes use of the assumption that all interior angles are 90° or larger. This condition is often violated in practice, so he suggests handling small interior angles by surrounding each vertex of an acute angle with a ring of *shield edges*. As the negative result stated above suggests, there are PSLGs for which shield edges fail, and for which no construction can succeed. Fortunately, all such PSLGs I am aware of have an interior angle much smaller than θ, so failure is generally predictable.

The reasoning behind the result is as follows. Suppose a segment in a conforming triangulation has been split into two subsegments of lengths a and b, as illustrated in Figure 15. Mitchell [13] proves that if the triangulation has no angles smaller than θ, then the ratio b/a has an upper bound of $(2\cos\theta)^{180°/\theta}$. (This bound is tight if $180°/\theta$ is an integer;

Figure 15 offers an example where the bound is obtained.) Hence any bound on the smallest angle of a triangulation imposes a limit on the gradation of triangle sizes along a segment (or anywhere in the mesh).

A problem can arise if a small angle ϕ occurs at the intersection point o of two segments of a PSLG, as illustrated in Figure 16 (top). The small angle cannot be improved, of course, but one does not wish to create any new small angles. Assume that one of the segments is split by a point p, which may be present in the input or may be inserted to help achieve the angle constraint elsewhere in the triangulation. The insertion of p forces part of the region between the two segments to be triangulated (Figure 16, center), which can cause a new point q to be inserted on the segment containing p. Let $a = |\overline{pq}|$ and $b = |\overline{op}|$ as illustrated. If the angle bound is maintained, the length a cannot be large; the ratio a/b is bounded below

$$\frac{\sin\phi}{\sin\theta}\left(\cos(\theta+\phi) + \frac{\sin(\theta+\phi)}{\tan\theta}\right).$$

If the region above the segments is part of the interior of the PSLG, the fan effect demonstrated in Figure 15 may necessitate the insertion of another vertex r between o and p (Figure 16, bottom); this circumstance is unavoidable if the product of the bounds on b/a and a/b given above is less than one. For an angle constraint of $\theta = 30°$, this condition occurs when ϕ is about six tenths of a degree. Unfortunately, the new vertex r creates the same conditions as the vertex p, but closer to o; the process will cascade, eternally creating smaller and smaller triangles in an attempt to satisfy the angle constraint. No algorithm can produce a finite triangulation of such a PSLG without violating the angle constraint. (It is amusing to consider whether the angle constraint can be met if one is allowed an infinite number of triangles.)

If some PSLGs do not have quality triangulations, what are the implications for shielding? Triangle implements a variant of shielding known as "modified segment splitting using concentric circular shells" (see Ruppert [15] for details), which is generally effective in practice for PSLGs that have small angles greater than 5°, and often for smaller angles. Shielding is useful even though it cannot solve all problems. On the other hand, the Delaunay refinement algorithm does not know to use careful arrangements of triangles as in Figure 15 to manage small input angles, and therefore can fail to terminate even on PSLGs for which a quality triangulation exists. Hence, Triangle prints a warning message when angles smaller than five degrees appear between input segments. The smaller an angle is, and the greater the number of small angles in a PSLG, the less likely Triangle is to terminate. An interesting question for future work is how to determine when and where it is wise to weaken the angle constraint so that termination can be ensured.

This problem presents another motivation for removing triangles from holes and concavities prior to applying the Delaunay refinement algorithm. Holes with small angles might cause the algorithm to fail if triangles are not removed until after refinement. Concave objects can be particularly dastardly, because a very small angle may occur between a defining segment of the object and an edge of the convex hull. The user, unaware of the effect of the convex hull edge, would be mystified why the Delaunay refinement algorithm fails to terminate on what appears to be a simple PSLG. (In fact, this is how the issues described in this section first became evident to me.) Early removal of triangles from concavities avoids this problem.

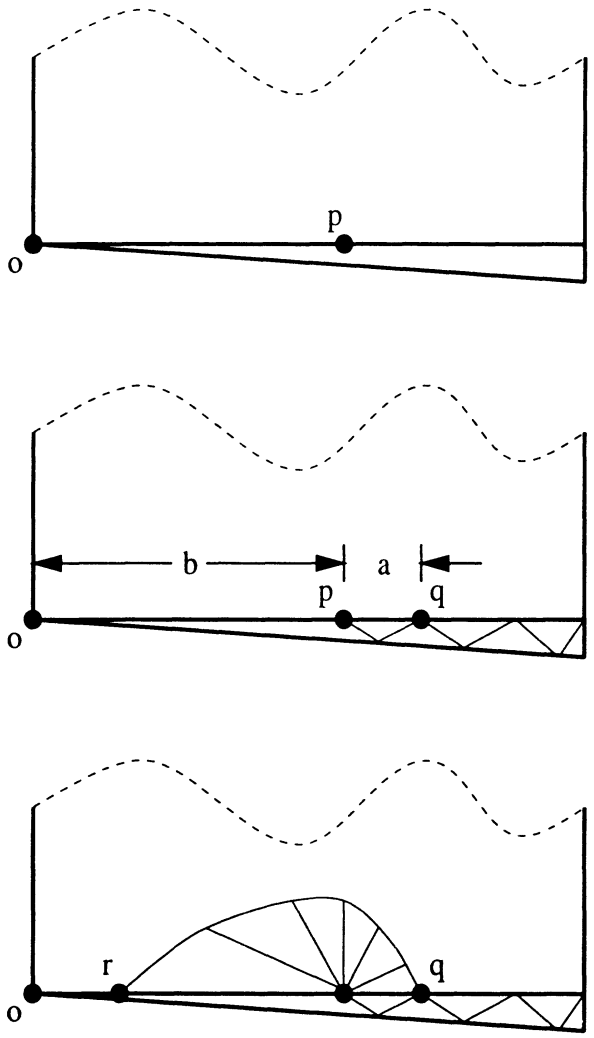


Figure 16: Top: A difficult PSLG with a small interior angle ϕ. Center: The point p and the angle constraint necessitate the insertion of the point q. Bottom: The point q and the angle constraint necessitate the insertion of the point r. The process repeats eternally.

4 Correct Adaptive Tests

The correctness of the incremental and divide-and-conquer algorithms depends on reliable *orientation* and *incircle* tests. The orientation test determines whether a point lies to the left of, to the right of, or on a line; it is used in many (perhaps most) geometric algorithms. The incircle test determines whether a point lies inside, outside, or on a circle. Inexact versions

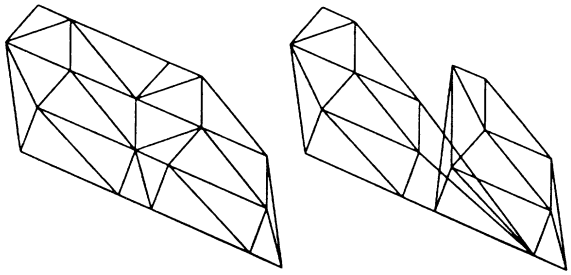


Figure 17: Left: A Delaunay triangulation (two of the guitar's tuning screws). Right: An invalid triangulation created by Triangle with exact arithmetic disabled.

of these tests are vulnerable to roundoff error, and the wrong answers they produce can cause geometric algorithms to hang, crash, or produce incorrect output. Figure 17 demonstrates a real example of the failure of Triangle's divide-and-conquer algorithm.

The easiest solution to many of these robustness problems is to use software implementations of exact arithmetic, albeit often at great expense. It is common to hear reports of implementations being slowed by factors of ten or more as a consequence. The goal of improving the speed of correct geometric calculations has received much recent attention [4, 8, 1], but the most promising proposals take integer or rational inputs, often of limited precision. These methods do not appear to be usable if it is convenient or necessary to use ordinary floating-point inputs.

Triangle includes fast correct implementations of the orientation and incircle tests that take floating-point inputs. They owe their speed to two features. First, they employ new fast algorithms for arbitrary precision arithmetic that have a strong advantage over other software techniques in computations that manipulate values of extended but small precision. Second, they are adaptive; their running time depends on the degree of uncertainty of the result, and is usually small. For instance, the adaptive orientation test is slow only if the points being tested are nearly or exactly collinear.

The orientation and incircle tests both work by computing the sign of a determinant. Fortune and Van Wyk [8] take advantage of the fact that only the sign is needed by using a *floating-point filter*: the determinant is first evaluated approximately, and only if forward error analysis indicates that the sign of the approximate result cannot be trusted does one use an exact test. Triangle's adaptive implementations carry this suggestion to its logical extreme by computing a sequence of successively more accurate approximations to the determinant, stopping only when the accuracy of the sign is assured. To reduce computation time, some of these approximations can reuse previous, less accurate computations. Shewchuk [16] presents details of the arbitrary precision arithmetic algorithms and the adaptivity scheme, and provides empirical evidence that multiple-stage adaptivity can significantly improve on two-stage adaptivity when difficult point sets are triangulated.

Using the adaptive tests, Triangle computes Delaunay triangulations, constrained Delaunay triangulations, and convex hulls exactly, roundoff error notwithstanding. Table 1 shows that the robust tests usually incur only a 10% to 30% overhead, though more time may be needed for points sets with many near-degeneracies. One exception is the divide-and-conquer algorithm with vertical cuts. Because this algorithm repeatedly merges tall, thinly separated triangulations, it performs many orientation tests on nearly-collinear points, and hence the robust version is much slower than the non-robust version. The variant that uses alternating cuts encounters nearly-collinear points less often; hence, its robust version suffers a smaller speed handicap, and its non-robust version is less likely to fail.

Of course, adaptive tests do not solve all robustness problems. Geometric computations that produce new vertices, including circumcenters and segment intersections, could be performed exactly in principle, but the results would have large bit complexity and would be inconvenient to manipulate and expensive to store. Worse, vertices of arbitrarily large bit complexity could eventually be produced in a cascading effect when the Delaunay refinement algorithm inserts circumcenters of triangles whose vertices were themselves circumcenters. Hence, it is infeasible to make the algorithm perfectly robust. Fortunately, the Delaunay refinement algorithm is naturally stable with regard to floating-point roundoff error. Problems arise only when triangles are refined to so small a size that it is no longer possible to construct a circumcenter that is distinct from its triangle's vertices.

I have not produced a robust version of the sweepline algorithm for a somewhat technical reason. The sweepline algorithm maintains a priority queue (normally implemented as a heap) containing two types of events: *site events*, where the sweepline passes over an input point, and *circle events*, where the sweepline reaches the top of a circle defined by three consecutive vertices on the boundary of the triangulation. Unfortunately, the y-coordinate of such a circle top is expensive to compute exactly, may be irrational, and has a somewhat complicated exact representation. A robust implementation must keep the events correctly ordered, and hence must replace the simple comparisons normally used to maintain a priority queue with a test that correctly compares two circle tops. Even a fast adaptive version of such a test would be so much slower than simple comparisons that event queue maintenance, which is a dominant cost of the sweepline algorithm, would become prohibitively expensive.

A Additional Implementation Notes

The sweepline and incremental Delaunay triangulation implementations compared by Su and Drysdale [18] each use some variant of uniform bucketing to locate points. Bucketing yields fast implementations on uniform point sets, but is easily defeated; a small, dense cluster of points in a large, sparsely populated region may all fall into a single bucket. I have not used bucketing in Triangle, preferring algorithms

that exhibit good performance with any distribution of input points. As a result, Triangle may be slower than necessary when triangulating uniformly distributed point sets, but will not exhibit asymptotically slower running times on difficult inputs.

Fortune's sweepline algorithm uses two nontrivial data structures in addition to the triangulation: a priority queue to store events, and a balanced tree data structure to store the sequence of edges on the boundary of the mesh. Fortune's own implementation, available from Netlib, uses bucketing to perform both these functions; hence, an $O(n \log n)$ running time is not guaranteed, and Su and Drysdale [18] found that the original implementation exhibits $O(n^{3/2})$ performance on uniform random point sets. By modifying Fortune's code to use a heap to store events, they obtained $O(n \log n)$ running time and better performance on large point sets (having more than 50,000 points). However, bucketing outperforms a heap on small point sets.

Triangle's implementation uses a heap as well, and also uses a splay tree [17] to store mesh boundary edges, so that an $O(n \log n)$ running time is attained, regardless of the distribution of points. Not all boundary edges are stored in the splay tree; when a new edge is created, it is inserted into the tree with probability 0.1. (The value 0.1 was chosen empirically to minimize the triangulation time for uniform random point sets.) At any time, the splay tree contains a random sample of roughly one tenth of the boundary edges. When the sweepline sweeps past an input point, the point must be located relative to the boundary edges; this point location involves a search in the splay tree, followed by a search on the boundary of the triangulation itself.

Splay trees adjust themselves so that frequently accessed items are near the top of the tree. Hence, a point set organized so that many new vertices appear at roughly the same location on the boundary of the mesh is likely to be triangulated quickly. This effect partly explains why Triangle's sweepline implementation triangulates points on the boundary of a circle more quickly than the other point sets, even though there are many more boundary edges in the cocircular point set and the splay tree grows to be much larger (containing $O(n)$ boundary edges instead of $O(\sqrt{n})$).

Triangle's incremental insertion algorithm for Delaunay triangulation uses the point location method proposed by Mücke, Saias, and Zhu [14]. Their *jump-and-walk* method chooses a random sample of $O(n^{1/3})$ vertices from the mesh (where n is the number of nodes *currently* in the mesh), determines which of these vertices is closest to the query point, and walks through the mesh from the chosen vertex toward the query point until the triangle containing that point is found. Mücke et al. show that the resulting incremental algorithm takes expected $O(n^{4/3})$ time on uniform random point sets. Table 1 appears to confirm this analysis. Triangle uses a sample size of $0.45n^{1/3}$; the coefficient was chosen empirically to minimize the triangulation time for uniform random point sets. Triangle also checks the previously inserted point, because in many practical point sets, any two consecutive points have a high likelihood of being near each other.

A more elaborate point location scheme such as that suggested by Guibas, Knuth, and Sharir [9] could be used (along with randomization of the insertion order) to obtain an expected $O(n \log n)$ triangulation algorithm, but the data structure used for location is likely to take up as much memory as the triangulation itself, and unlikely to surpass the performance of the divide-and-conquer algorithm; hence, I do not intend to pursue it.

Note that all discussion in this paper applies to Triangle version 1.2; earlier versions lack the sweepline algorithm and many optimizations to the other algorithms.

References

[1] Francis Avnaim, Jean-Daniel Boissonnat, Olivier Devillers, Franco P. Preparata, and Mariette Yvinec. *Evaluating Signs of Determinants Using Single-Precision Arithmetic.* 1995.

[2] Marshall Bern and David Eppstein. *Mesh Generation and Optimal Triangulation.* Computing in Euclidean Geometry (Ding-Zhu Du and Frank Hwang, editors), Lecture Notes Series on Computing, volume 1, pages 23–90. World Scientific, Singapore, 1992.

[3] L. Paul Chew. *Guaranteed-Quality Mesh Generation for Curved Surfaces.* Proceedings of the Ninth Annual Symposium on Computational Geometry, pages 274–280. Association for Computing Machinery, May 1993.

[4] Kenneth L. Clarkson. *Safe and Effective Determinant Evaluation.* 33rd Annual Symposium on Foundations of Computer Science, pages 387–395. IEEE Computer Society Press, October 1992.

[5] Rex A. Dwyer. *A Faster Divide-and-Conquer Algorithm for Constructing Delaunay Triangulations.* Algorithmica 2(2):137–151, 1987.

[6] Steven Fortune. *A Sweepline Algorithm for Voronoï Diagrams.* Algorithmica 2(2):153–174, 1987.

[7] ———. *Voronoï Diagrams and Delaunay Triangulations.* Computing in Euclidean Geometry (Ding-Zhu Du and Frank Hwang, editors), Lecture Notes Series on Computing, volume 1, pages 193–233. World Scientific, Singapore, 1992.

[8] Steven Fortune and Christopher J. Van Wyk. *Efficient Exact Arithmetic for Computational Geometry.* Proceedings of the Ninth Annual Symposium on Computational Geometry, pages 163–172. Association for Computing Machinery, May 1993.

[9] Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. *Randomized Incremental Construction of Delaunay and Voronoï Diagrams.* Algorithmica 7(4):381–413, 1992.

[10] Leonidas J. Guibas and Jorge Stolfi. *Primitives for the Manipulation of General Subdivisions and the Computation of Voronoï Diagrams.* ACM Transactions on Graphics **4**(2):74–123, April 1985.

[11] C. L. Lawson. *Software for C^1 Surface Interpolation.* Mathematical Software III (John R. Rice, editor), pages 161–194. Academic Press, New York, 1977.

[12] D. T. Lee and B. J. Schachter. *Two Algorithms for Constructing a Delaunay Triangulation.* International Journal of Computer and Information Sciences **9**(3):219–242, 1980.

[13] Scott A. Mitchell. *Cardinality Bounds for Triangulations with Bounded Minimum Angle.* Sixth Canadian Conference on Computational Geometry, 1994.

[14] Ernst P. Mücke, Isaac Saias, and Binhai Zhu. *Fast Randomized Point Location Without Preprocessing in Two- and Three-dimensional Delaunay Triangulations.* Proceedings of the Twelfth Annual Symposium on Computational Geometry. Association for Computing Machinery, May 1996.

[15] Jim Ruppert. *A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation.* Journal of Algorithms **18**(3):548–585, May 1995.

[16] Jonathan Richard Shewchuk. *Robust Adaptive Floating-Point Geometric Predicates.* Proceedings of the Twelfth Annual Symposium on Computational Geometry. Association for Computing Machinery, May 1996.

[17] Daniel Dominic Sleator and Robert Endre Tarjan. *Self-Adjusting Binary Search Trees.* Journal of the Association for Computing Machinery **32**(3):652–686, July 1985.

[18] Peter Su and Robert L. Scot Drysdale. *A Comparison of Sequential Delaunay Triangulation Algorithms.* Proceedings of the Eleventh Annual Symposium on Computational Geometry, pages 61–70. Association for Computing Machinery, June 1995.

AUTOMATED DESIGN OF LOG-PERIODIC FOLDED DIPOLE AND FOLDED SLOT ANTENNAS

Michael Carr (mcarr@umich.edu)
John Volakis (volakis@umich.edu)
December 15, 1996

INTRODUCTION

While log-periodic folded dipole antennas can be completeley characterized with only four basic parameters, construction or synthesis of a LPFD antenna can be time-consuming because of the many geometry points that must be calculated using these parameters. Design iterations are in turn time consuming, because the slightest change in design parameters requires a recalculation of each point.

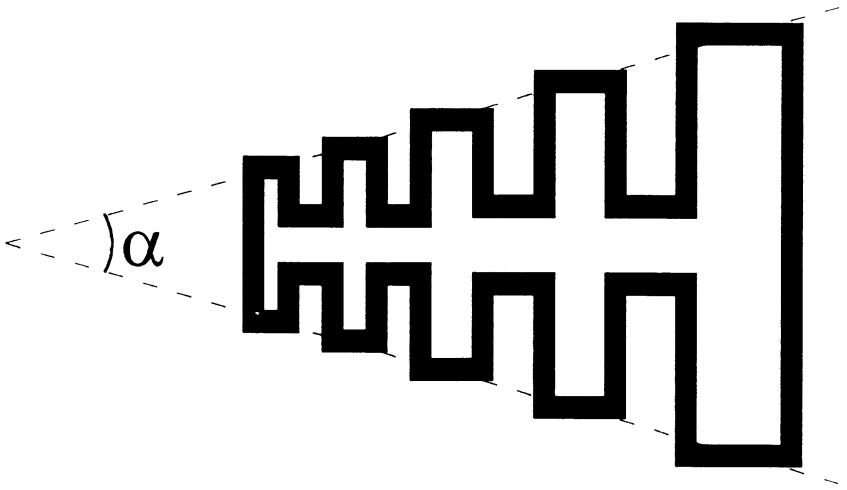


Figure 1: Basic LPFD Geometry

The 32-bit Windows application developed in this project uses four basic parameters to design a LPFD or LPFS antenna. It saves a DXF-format file containing the antenna geometry suitable for import into AutoCAD or one of many other CAD packages supporting DXF.

USER INTERFACE

The program's user interface (shown in Figure 2) is comprised of a single dialog box prompting for entry of the antenna's design parameters.

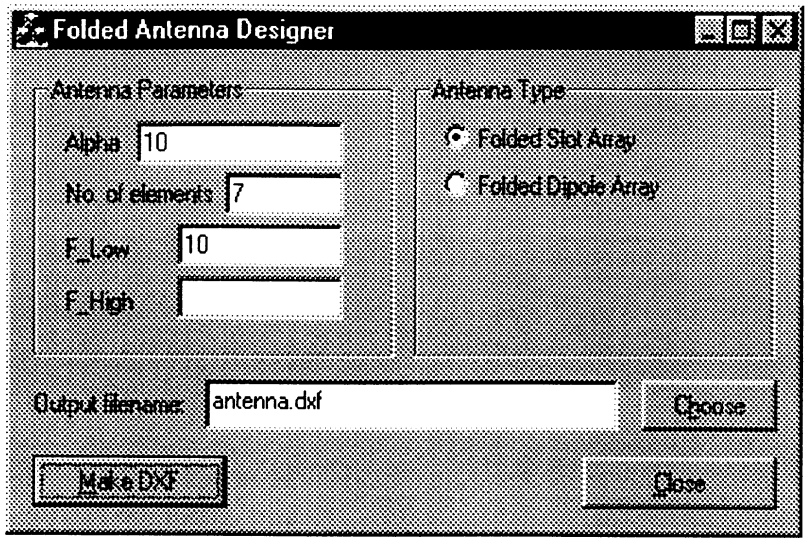


Figure 2: User Interface

- Alpha
  This parameter establishes the angle between the top and bottom of any given element in the antenna.
- No. of elements
  The antenna may have anywhere from one to 999 dipole elements.
- F_Low and F_High
  These parameters specify the lowest and highest desired frequencies. They do not insure that the antenna will operate at these frequencies, however. The values are only used to calculate the shortest and longest elements of the antenna. Middle elements are interpolated from these two calculated elements.
- Antenna Type
  These radio buttons select between a folded dipole, an antenna which is constructed of metal, or a folded slot, an antenna which is etched out of metal.
- Output filename
  This box lets you choose a name for the resulting .DXF file. By default, it will create a file named antenna.dxf. Click the "Choose" button to use a file requester or simply type the name into the text box.
- Make DXF
  After all parameters have been specified, this button will generate the desired .DXF for the antenna and exit the program.
- Cancel
  This button exits the program without generating the .DXF file.

CONCLUSIONS

This pre-alpha release of the code has the general program framework implemented, but does not generate antennas of the correct dimension, nor of the correct shape. Future releases will correct these problems. Please feel free to offer suggestions as to the user interface and program operation.

This is a listing of all directories in this delivery.

Directory Listing:

DIRECTORY bin:

This directory contains all executables (compiled on SGI) including;

Acad.in	acadproc input file.
Acad.in.bak	Commented acadproc input file.
Tri.in	triproc input file.
Tri.in.bak	Commented triproc input file.
acadproc	Converts AutoCAD .dxf file to Triangle .poly input file.
antenna.exe	Experimental DOS executable to automatically generate Log-Periodic .dxf line file.
farfield	Generates far-field data from FEMA-PRISM output magnetic current file EqvCur.
fema	FEMA-PRISM executable.
showme	Triangle Viewing tool.
triangle	2D mesher.
triproc	Converts Triangle output .1.node and .1.ele to FEMA-PRISM "surfmesh" input file.

DIRECTORY doc:

This directory contains documentation for routines delivered.

LOG-PERI.MIF.ps	Postscript copy of final report.
Manualv1.ps	FEMA-PRISM user's manual
cover.ps	Cover page for report.
meshman.ps	AutoCAD Interface Manual (i.e. .dxf to Triangle using AcadProc.f and Triangle to surfmesh using TriProc.f
minp1.ps/minp2.ps	Updated FEMA_PRISM Input File format including conducting pins, layers, holes, etc.
trianglev1_3.doc	Triangle meshing package user's manual.

DIRECTORY examples:

This directory contains example input files, meshes, and data for the single folded slot (slot subdirectory) and the 7-element LPFSA (lpfsa subdirectory).

The subdirectory for each antenna contains:

an execute directory containing:

1) The AutoCAD .dwg drawing file.
2) The AutoCAD .dxf file.
3) The .poly file created by AcadProc.f.
4) The .1.node and .1.ele files created by Triangle (Note that the end of these files shows the Triangle command and options used to create the files).
5) The FEMA_PRISM input "surfmesh" mesh file created by TriProc.f
6) MainInput, the FEMA_PRISM input parameter file.
7) Acad.in, Input file for AcadProc.f.
8) Tri.in, Input file for TriProc.f.

a data directory containing FEMA_PRISM and FarField.f output files:

1) EdgeUnk - FEMA_PRISM edge (field) unknowns.
2) EqvCur - FEMA_PRISM magnetic current file (used by FarField.f).
3) Imp - FEMA_PRISM probe impedence.
4) RUNDATA - Run data output of FEMA_PRISM.
5) *E.pat, and *H.pat FarField output patterns. (Note that these patterns are combined outputs from FarField. 2 FarField runs were made for each pattern, i.e., for E-Cut single slot runs were phi=180, theta=90,0,5 and phi=0, theta=0,90,5. After being combined aspects were renumbered to go from 0 to 180 for each cut.

DIRECTORY src:

This directory and subdirectories contain all source code and make files.

Subdirectory fema_prism contains:

This subdirectory contains all FEMA_PRISM subroutines and makefiles as well as FarField.f source code for farfield pattern generation.

In addition to FEMA_PRISM subroutines there is also:

Farfield.f	Source code for farfield pattern generation routine.
Makefile	SGI makefile including O3 optimization.
Makefile_SGI	Same as above.
MakefileNOOPT	SGI makefile with no optimization.

Subdirectory preproc contains:

This subdirectory contains preprocessing routines (converting AutoCAD to FEMA_PRISM input) as well as appropriate makefiles.

AcadProc.f	Routine to convert AutoCAD .dxf files to Triangle .poly input file.
AcadProc.inc	Include file for above.
TriProc.f	Routine to convert Triangle output .1.node and .1.ele files to FEMA_PRISM "surfmesh" file.

Subdirectory triangle contains:

This subdirectory contains all Triangle meshing C source code as well as Showme viewing tool C source code.

A.poly	Example .poly boundary line file.

makefile	Triangle and Showme make file.
showme.c	C-source code for Showme viewing package.
triangle.c	C-source code for Triangle 2D meshing package.
triangle.h	Include file for triangle.c
triangle.shar	Archived triangle/showme package.
tricall.c	Example program on how to call Triangle.