This is to certify that the
dissertation entitled

MIXED ELEMENT FORMULATION FOR THE FINITE ELEMENT-BOUNDARY INTEGRAL METHOD

presented by

Jeffrey David Meese

has been accepted towards fulfillment
of the requirements for the

Ph.D. degree in Electrical and Computer Engineering

Major Professor's Signature

17 June 2003

Date

MIXED ELEMENT FORMULATION FOR THE FINITE ELEMENT-BOUNDARY INTEGRAL METHOD

By

Jeffrey David Meese

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Electrical and Computer Engineering

2003

ABSTRACT

MIXED ELEMENT FORMULATION FOR THE FINITE ELEMENT-BOUNDARY INTEGRAL METHOD

By

Jeffrey David Meese

The Finite Element-Boundary Integral (FE-BI) method for numerical simulation combines the computational and memory efficiency of finite element methods with the global enforcement of the transition conditions offered by the method of moments making it attractive for modeling three dimensional cavity backed apertures. Using a finite element formulation in the cavity region removes the need to provide a Green's function for the entire cavity and the resulting matrices are extremely sparse. For the aperture regions, using a moment method approach allows the Sommerfeld radiation condition to be satisfied, as well as an exact relationship between tangential magnetic fields across the aperture to be enforced, requiring only a specification of the Green's function for the two dimensional aperture. This moment method approach, however, leads to a fully populated matrix for the aperture which constitutes a large portion of the computational and memory demand of the formulation. Traditionally, FE-BI formulations have used a single type of element such as hexahedrals or prisms alone to discretize the geometry. Hexahedrals have a difficult time modeling highly irregular contours while prisms tend to produce a large number of free edges in the aperture region. This dissertation presents a method to use a combination of hexahedral and prism elements for FE-BI modeling. The accuracy and efficiency of this approach is demonstrated using some complex aperture examples. In addition, this dissertation presents a software implementation based on object oriented programming that yields more efficient and elegant solutions compared to previous programming techniques.

To my parents, who always believed in me

ACKNOWLEDGMENTS

While my name may be the only one on this dissertation there were many people directly and indirectly involved in its creation. These people were instrumental in producing this dissertation and it seems appropriate to credit them accordingly.

First, I must thank my academic advisor Dr. Leo Kempel. His tireless support and uncurbed enthusiasm for both electromagnetics and my enjoyment of it are astounding. I have learned a great many lessons from him that will guide me for the rest of my career and life. Next, I would like to thank my committee member Dr. Ed Rothwell. His knowledge of electromagnetics and his interest in the students attending his lectures were the original catalyst for my decision to pursue graduate studies in electromagnetics. He is a true mentor. Next, I would like to acknowledge my committee member Dr. Dennis Nyquist. His knowledge of electromagnetics is virtually unsurpassed and he has helped me solve numerous problems in my graduate studies on which I was hopelessly stuck. Finally, I would like to thank my final committee member Dr. Byron Drachman for his helpful suggestions throughout the writing of this dissertation.

Without my family and friends I would have never been able to find the will to finish this dissertation. Through many times of serious frustration and doubt my family and friends were always there with an encouraging word to keep me going. I would first like to thank my parents. They are my biggest cheerleaders and my largest source of inspiration. They've given me everything I've ever needed and I hope I have repayed them by achieving the goals they knew I could. I would also like

to thank my grandparents. Sadly, my grandfather passed away just as I was about to finish this dissertation. I was able to participate in commencement ceremonies before my dissertation was fully completed and he was adamite about attending them even though it took every ounce of strength that he had. I am truly grateful he was able to see me graduate. I would also like to thank my grandmother for being so amazing. She is a wonderful woman who always make sure that I know how much she loves me. I would also like to personally acknowledge my sister Amy. She is a great sister and a wonderful friend. I can not think of a better person to have as a sister. Finally, I would like to thank my girlfriend Candace. Her patience and understanding throughout the writing of this dissertation have been immense. She is a truly incredible person and I look forward to continuing our life together.

Last, but not least, I would like to thank all my friends and colleagues at the electromagnetics group here at Michigan State University. They have been a constant source of friendship and support throughout my many years of study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Three dimensional cavity-backed apertures, such as conformal or microstrip patch antennas, are of great interest in electromagnetics research due to their capabilities in areas including, but not limited to, direction finding (DF) and communication. Together with the push to integrate more capabilities on a single aperture, finding accurate yet efficient models for these devices is becoming paramount. Traditional closed form solutions do not exist for almost any practical antennas; hence to model these devices effectively some sort of numerical solution must be involved. Many such methods exist including modal and non-modal solutions, asymptotic wave solutions, the Finite Difference Time Domain Method (FDTD), the Method of Moments (MoM), and the Finite Element Method (FEM). All of these techniques have advantages and disadvantages and a great wealth of material is available on each technique; however, this dissertation will only concern itself with the MoM and the FEM.

1.1 Background

The Finite Element Method (FEM) has a history dating back to the 1940's and was first used in the field of airplane structural analysis [1]. Electrical engineering research based on FEM did not appear until the late 1960's where it was used for waveguide and cavity analysis. FEM had the advantage over boundary integral techniques such as MoM since it did not require the Green's function for the geometry. In addition, since FEM is based on a partial differential equation (PDE) approach as opposed to an integral equation (IE) approach, it resulted in very sparse matrices for linear systems, thereby reducing memory and computational demand.

One of the key concepts related to FEM is the choice of suitable elements in which to subdivide the problem domain. These elements are then assigned a suit-

able set of basis functions (sometimes called shape or expansion functions) which are used to approximate the unknown quantity of interest. Traditionally FEM has been implemented using node-based elements, where the unknown is approximated at the nodes of the finite elements. This worked well for scalar quantities such as those used in mechanical engineering and scalar wave equations in electromagnetics. Problems arose, however, when these node-based elements were used to model three dimensional vector electric or magnetic fields. Spurious modes [2] were observed during the modeling of cavities and special conditions had to be enforced at material discontinuities and geometry corners [3]. The introduction of edge-based elements, first described by Whitney [4], relieved these problems since they were able to more effectively represent a vector field.

Using edge-based elements, FEM was able to model three dimensional waveguide and cavity models effectively but it was still not suited for modeling open domain problems since the Sommerfeld radiation condition was not enforced. Two major techniques were used to extend the technique for open domain problems. The first was a combination of FEM and MoM, termed the Finite Element-Boundary Integral (FE-BI) technique [5]-[8]. This approach satisfies Sommerfeld radiation at infinity as well as the transition condition across the aperture, however, it does require the knowledge of the Green's function for the open domain. The second technique involved the use of absorbing boundary conditions (ABC's) [9]-[11] which were used to extend the mesh beyond the dimensions of the structure and to approximately enforce the radiation condition. This technique suffered from the fact that the accuracy of the solution was dependent on how good the absorbing boundary condition was able to enforce the radiation condition and how far the mesh was extruded beyond the problem domain. In this dissertation, the FE-BI technique will be used since the problems studied herein will consist of cavity-backed apertures cut into an infinite planar ground screen, for which a suitable closed form Green's function is well known.

1.2 Overview

Chapter 2 introduces the general formulation for the FE-BI method. The equations will be derived starting from Maxwell's equations and placed into a suitable form for linear system solution. General excitation formulations will be presented along with some general post processing formulations.

Chapter 3 extends the formulation developed in Chapter 2 for use with right prism elements. Right prism elements are a popular choice for modeling geometries such as those studied in this dissertation as they can readily be extruded from two dimensional triangular elements. Right prism elements also have the advantage that many of the integrals associated with the FE-BI formulation can be expressed in closed form.

Chapter 4 extends the formulation of Chapter 2 for use with distorted hexahedral elements. Hexahedral elements have the advantage of producing fewer unknowns than the right prism elements used in chapter 3, however, they are not as adept at modeling highly irregular geometries. This chapter discusses the techniques needed to use hexahedral elements in an FE-BI solution including the coordinate transformations needed to perform the necessary integrals.

Chapter 5 combines the two elements from chapters 3 and 4 into the same geometry. It will be shown that using a combination of hexahedral and prism elements in the same model can significantly reduce the number of unknowns both in the volume and on the surface while still maintaining the ability to model highly irregular surfaces. This allows the modeling of complex geometries that were too computationally expensive to be modeled with prisms alone. Some examples of these types of geometries will be shown in chapter 6.

Chapter 6 models some highly complex apertures using the mixed element approach developed in Chapter 5. Memory and computational demand for the modeling of these apertures with only prisms and using mixed elements will be compared and

contrasted.

Chapter 7 discusses the design of the software program developed for this dissertation. The use of the C++ programming language and an object oriented design approach is discussed and compared to the traditional procedural programming approach implemented in FORTRAN.

Chapter 8 presents some final thoughts and conclusions as well as some future considerations for the modeling technique presented herein.

CHAPTER 2

GENERAL FE-BI FORMULATION

This chapter introduces the Finite Element-Boundary Integral (FE-BI) method and develops the general equations needed to apply it to a cavity-backed aperture problem. Fields for the interior region and exterior regions are developed separately and then tangential field continuity is enforced across the boundary of these two regions.

2.1 The FE-BI Equation

A typical three dimensional planar cavity-backed antenna configuration is shown in Figure 2.1. All electric and magnetic fields are assumed to obey Maxwell's equations and are considered to be time harmonic in nature. Accordingly, an $e^{j\omega t}$ term is assumed and suppressed throughout the analysis. The material parameters inside the cavity ($z < 0$) are not required to be homogeneous but are assumed to be isotropic. The material parameters of the exterior region ($z > 0$) are assumed to be those of free space. In the figure $\mathbf{E}^{ext}$ and $\mathbf{H}^{ext}$ represent the external electric and magnetic field respectively, while $\mathbf{E}^{int}$ and $\mathbf{H}^{int}$ represent the electric and magnetic fields within the cavity.

2.1.1 The Interior Equation

Starting with Maxwell's equations for the interior fields

$$\nabla \times \mathbf{E}^{int} = -jk_0 Z_0 \mu_r(\mathbf{r}) \mathbf{H}^{int} - \mathbf{M} \tag{2.1}$$

$$\nabla \times \mathbf{H}^{int} = jk_0 Y_0 \epsilon_r(\mathbf{r}) \mathbf{E}^{int} + \mathbf{J} \tag{2.2}$$

In the above equations k_0 is the free space wave number, Z_0 is the intrinsic impedance of free space, Y_0 is the reciprocal of Z_0, $\mu_r(\mathbf{r})$ is the relative permeability of the

material inside the cavity, $\epsilon_r(\mathbf{r})$ is the relative permittivity of the material inside the cavity, and $\mathbf{J}$ and $\mathbf{M}$ are internal excitation sources of an electric or magnetic type respectively. For the sake of simplicity the spatial dependence of μ_r and ϵ_r will not be specifically shown from this point forward. From these two equations the vector wave equation can be derived in terms of either the electric field or the magnetic field. The former is referred to as an electric field formulation, while the latter is referred to as a magnetic field formulation. A electric field formulation is chosen here as it allows the boundary conditions on a perfectly electrically conducting surface to be easily satisfied. Assuming the use of edge-based finite elements the electric field contribution of a edge located on metal is set to zero. This amounts to the enforcement of the Dirichlet boundary condition. So for the case of a electric field formulation only non-metal or free edges need be considered. If a magnetic field formulation was used it would be necessary to enforce the Neumann boundary condition on all metal surfaces to satisfy the boundary conditions. Taking the curl of both sides of (2.1)

$$\nabla \times \frac{\nabla \times \mathbf{E}^{int}}{\mu_r} = -jk_0 Z_0 \left(\nabla \times \mathbf{H}^{int}\right) - \frac{\nabla \times \mathbf{M}}{\mu_r} \tag{2.3}$$

and utilize (2.2) to simplify (2.3), the vector wave equation for the interior total electric field is obtained

$$\nabla \times \frac{\nabla \times \mathbf{E}^{int}}{\mu_r} - k_0^2 \epsilon_r \mathbf{E}^{int} = -jk_0 Z_0 \mathbf{J} - \frac{\nabla \times \mathbf{M}}{\mu_r} \tag{2.4}$$

Defining the inner product as

$$< u, v >= \int u \cdot v \tag{2.5}$$

the finite element method is applied by performing an inner product of (2.4) with a vector sub-domain basis function $\mathbf{Q}_i$, creating the weighted residual

$$\int_V \mathbf{Q}_i \cdot \left[\nabla \times \frac{\nabla \times \mathbf{E}^{int}}{\mu_r} - k_0^2 \epsilon_r \mathbf{E}^{int}\right] dV = -\int_V \mathbf{Q}_i \cdot \left[j_0 k_0 Z_0 \mathbf{J} + \frac{\nabla \times \mathbf{M}}{\mu_r}\right] dV \quad (2.6)$$

It is assumed that all quantities on the right hand side of the equation are known *a priori*, so the right hand side may be written as

$$f_i^{int} = -\int_V \mathbf{Q}_i \cdot \left[j_0 k_0 Z_0 \mathbf{J} - \frac{\nabla \times \mathbf{M}}{\mu_r}\right] dV \quad (2.7)$$

In this, the superscript 'int' indicates that this forcing function is attributed to interior sources. Using (2.7), (2.6) becomes

$$\int_V \mathbf{Q}_i \cdot \left[\nabla \times \frac{\nabla \times \mathbf{E}^{int}}{\mu_r} - k_0^2 \epsilon_r \mathbf{E}^{int}\right] dV = f_i^{int} \quad (2.8)$$

Using a vector identity, the first term in brackets can be written as

$$\nabla \times \frac{\nabla \times \mathbf{E}^{int}}{\mu_r} = \frac{1}{\mu_r}\left(\nabla \times \nabla \times \mathbf{E}^{int}\right) - \nabla \times \mathbf{E}^{int} \times \nabla\left(\frac{1}{\mu_r}\right) \quad (2.9)$$

At this point it is assumed that the material is constant within each element. This assumption allows for the closed form solution of many of the integrals associated with this method for various elements. Without this assumption these integrals would have to be evaluated numerically. With this assumption the gradient term in (2.9) is zero. Therefore (2.8) is now written as

$$\int_V \mathbf{Q}_i \cdot \left[\frac{\nabla \times \nabla \times \mathbf{E}^{int}}{\mu_r} - k_0^2 \epsilon_r \mathbf{E}^{int}\right] dV = f_i^{int} \quad (2.10)$$

At a point later in the development, the electric field will be expanded as a set of sub-domain basis functions in accordance with the finite element method. As (2.10) stands now, a linear tangential-quadratic normal (LT/QN) basis function would be required to model the electric field. LT/QN basis functions are harder to implement than constant tangential-linear normal (CT/LN) basis functions which can be used if one of the curl operations is transferred from the unknown electric field to the basis function $\mathbf{Q}_i$. This also allows the use of CT/LN basis functions for both the testing function $\mathbf{Q}_i$ and electric field expansion. Using Green's first vector theorem [17], the first term of (2.10) can be written as

$$\frac{1}{\mu_r}\int_V \mathbf{Q}_i \cdot \left(\nabla \times \nabla \times \mathbf{E}^{int}\right) dV = \frac{1}{\mu_r}\int_V (\nabla \times \mathbf{Q}_i) \cdot \left(\nabla \times \mathbf{E}^{int}\right) dV - \frac{1}{\mu_r}\int_S \hat{\mathbf{n}} \cdot \left(\mathbf{Q}_i \times \nabla \times \mathbf{E}^{int}\right) dS \tag{2.11}$$

where S is the closed surface bounding V. Using (2.1) in (2.11), the functional becomes

$$\frac{1}{\mu_r}\int_V (\nabla \times \mathbf{Q}_i) \cdot \left(\nabla \times \mathbf{E}^{int}\right) dV - k_0^2 \epsilon_r \int_V \mathbf{Q}_i \cdot \mathbf{E}^{int} + jk_0 Z_0 \int_S \hat{\mathbf{n}} \cdot \left(\mathbf{Q}_i \times \mathbf{H}^{int}\right) dS = f_i^{int} \tag{2.12}$$

Equation (2.12) is termed the weak form of the equation since the PDE is represented weakly, or in an integrated sense and at most one derivative is applied to each term. Finally, it would be advantageous to manipulate (2.12) so that the normal vector is crossed with the magnetic field as this will simplify enforcing boundary conditions in future aspects of the development. Using another vector identity, (2.12) is written as

$$\frac{1}{\mu_r}\int_V (\nabla \times \mathbf{Q}_i) \cdot \left(\nabla \times \mathbf{E}^{int}\right) dV - k_0^2 \epsilon_r \int_V \mathbf{Q}_i \cdot \mathbf{E}^{int} - jk_0 Z_0 \int_S \mathbf{Q}_i \cdot \left(\hat{\mathbf{n}} \times \mathbf{H}^{int}\right) dS = f_i^{int} \tag{2.13}$$

2.1.2 The Exterior Equation

The equation shown above only describe the field quantities interior to the computational volume. In order to fully define the problem the fields exterior to the surface must also be modeled and these fields must match the fields on the interior as the surface is approached from either side. In the region exterior to the geometry the total magnetic field can be written as

$$\mathbf{H}^{ext} = \mathbf{H}^{inc} + \mathbf{H}^{ref} + \mathbf{H}^{scat} \tag{2.14}$$

where $\mathbf{H}^{inc}$ is the incident magnetic field, $\mathbf{H}^{ref}$ is the reflected magnetic field, and $\mathbf{H}^{scat}$ is the scattered magnetic field. The incident and reflected fields are those that exist without the presence of the antenna and are assumed to be known. Therefore, a relation must be determined between the scattered and total exterior magnetic fields. Following the development in [12] the scattered magnetic field for the exterior region can be written as

$$\mathbf{H}^{scat} = \int_{S'} \left(\hat{\mathbf{n}}' \times \mathbf{H}^{ext}\right) \cdot \left(\nabla \times \overline{\overline{\mathbf{G}}}\right) dS' - jk_0 Y_0 \int_{S'} \overline{\overline{\mathbf{G}}} \cdot \left(\hat{\mathbf{n}}' \times \mathbf{E}^{ext}\right) dS' \tag{2.15}$$

where $\overline{\overline{\mathbf{G}}}$ is a dyadic Green's function that satisfies the Sommerfeld radiation condition. Due to the third order singularity in dyadic Green's functions the integrals in (2.15) are not actually integrable. This form, however, is notationally convenient and later in the development the derivatives will be transferred off of the dyadic Green's function leaving only a first order singularity which is integrable. For more information on this dyadic Green's function singularity see [13]-[16].

The tangential electric field on the ground plane is zero so the second integral reduces to an integration over the aperture. Choosing a Green's function that obeys

the Neumann boundary condition,

$$\hat{\mathbf{n}} \times \nabla \times \overline{\overline{\mathbf{G}}}_{e2} = 0 \tag{2.16}$$

manipulating the first term in (2.15) so that the normal vector is crossed with the $\nabla \times \overline{\overline{\mathbf{G}}}$, and solving for the tangential portion of the scattered magnetic field, the first integral disappears entirely and (2.15) can be written as

$$\hat{\mathbf{n}} \times \mathbf{H}^{scat} = -jk_0 Y_0 \int_{S'} \left(\hat{\mathbf{n}} \times \overline{\overline{\mathbf{G}}}_{e2}\right) \cdot \left(\hat{\mathbf{n}}' \times \mathbf{E}^{ext}\right) dS' \tag{2.17}$$

where

$$\overline{\overline{\mathbf{G}}}_{e2} = -\left(\bar{\bar{\mathbf{I}}} + \frac{\nabla\nabla}{k_0^2}\right) \frac{e^{-jk_0 R}}{2\pi R} \tag{2.18}$$

is the electric dyadic Green's function of the second kind [17], in which $R = \left|\mathbf{r} - \mathbf{r}'\right|$. This Green's function possesses the properties desired for this development. Re-writing the integrand as

$$\hat{\mathbf{n}} \times \overline{\overline{\mathbf{G}}}_{e2} \cdot \hat{\mathbf{n}}' \times \mathbf{E}^{ext} = \left(\hat{\mathbf{n}} \times \overline{\overline{\mathbf{G}}}_{e2} \times \hat{\mathbf{n}}'\right) \cdot \mathbf{E}^{ext} \tag{2.19}$$

(2.14) can now be written as

$$\hat{\mathbf{n}} \times \mathbf{H}^{ext} = \left[\hat{\mathbf{n}} \times \mathbf{H}^{inc} + \hat{\mathbf{n}} \times \mathbf{H}^{ref}\right] - jk_0 Y_0 \int_{S'} \left(\hat{\mathbf{n}} \times \overline{\overline{\mathbf{G}}}_{e2} \times \hat{\mathbf{n}}'\right) \cdot \mathbf{E}^{ext} dS' \tag{2.20}$$

Taking the inner product of (2.20) with $\mathbf{Q_i}$

$$\begin{aligned} \int_S \mathbf{Q_i} \cdot \left(\hat{\mathbf{n}} \times \mathbf{H}^{ext}\right) dS = & \int_S \mathbf{Q_i} \cdot \left[\hat{\mathbf{n}} \times \left(\mathbf{H}^{inc} + \mathbf{H}^{ref}\right)\right] dS \\ & -jk_0 Y_0 \int_S \int_{S'} \mathbf{Q_i} \cdot \left(\hat{\mathbf{n}} \times \overline{\overline{\mathbf{G}}}_{e2} \times \hat{\mathbf{n}}'\right) \cdot \mathbf{E}^{ext} dS' dS \end{aligned} \tag{2.21}$$

The first integral on the right hand side involves known quantities and is thus written

as

$$f_i^{ext} = \int_S \mathbf{Q}_i \cdot \left[\hat{\mathbf{n}} \times \left(\mathbf{H}^{inc} + \mathbf{H}^{ref}\right)\right] dS \tag{2.22}$$

where the 'ext' superscript indicates that this forcing function is attributed to exterior sources. Finally the exterior equation is found to be

$$\int_S \mathbf{Q}_i \cdot \left(\hat{\mathbf{n}} \times \mathbf{H}^{ext}\right) dS + jk_0 Y_0 \int_S \int_{S'} \mathbf{Q}_i \cdot \left(\hat{\mathbf{n}} \times \overline{\overline{\mathbf{G}}}_{e2} \times \hat{\mathbf{n}}'\right) \cdot \mathbf{E}^{ext} dS' dS = f_i^{ext} \tag{2.23}$$

One important thing to note about this equation is that through the use of the Green's function, the currents on the ground plane do not have to be explicitly modeled. This is fortunate since the ground plane is considered infinite in extent and modeling the currents on it exactly would be challenging indeed. To approximately model these currents, the ground plane would have to be truncated and techniques such as the Geometrical Theory of Diffraction (GTD) would have to be used to account for the finite extent of the ground plane.

Another important observation about this equation is that since the conductor is planar the Green's function can be written as a closed form expression. For cases where the conductor is not planar the Green's function would have to be solved numerically. Work has been done to this effect by [18] for right cylinders, by [19] for elliptical cylinders and [20] for prolate spheroids.

2.1.3 Matrix Formulation

In the previous sections the interior and exterior equation were derived separately. The next step is to combine these two equations in a way that satisfies tangential field continuity across the surface. In other words the following equations must be satisfied

$$\hat{\mathbf{n}} \times \mathbf{H}^{int} = \hat{\mathbf{n}} \times \mathbf{H}^{ext} \tag{2.24}$$

$$\hat{\mathbf{n}} \times \mathbf{E}^{int} = \hat{\mathbf{n}} \times \mathbf{E}^{ext} \tag{2.25}$$

The boundary condition associated with the magnetic field is often termed the natural boundary condition [2] and is enforced by setting $\mathbf{H}^{int} = \mathbf{H}^{ext}$. Upon enforcement of the natural boundary condition equation (2.23) is multiplied by jk_0Z_0 and the result is added to (2.13). The terms with the magnetic field are seen to cancel and the equation is written as

$$\frac{1}{\mu_r}\int_V (\nabla \times \mathbf{Q}_i)\cdot\left(\nabla \times \mathbf{E}^{int}\right)dV - k_0^2\epsilon_r\int_V \mathbf{Q}_i\cdot\mathbf{E}^{int}dV$$
$$-k_0^2\int_S\int_{S'}\mathbf{Q}_i\cdot\left(\hat{\mathbf{n}}\times\overline{\overline{\mathbf{G}}}_{e2}\times\hat{\mathbf{n}}'\right)\cdot\mathbf{E}^{ext}dS'dS \;= f_i^{int} + \tilde{f}_i^{ext} \qquad (2.26)$$

where

$$\tilde{f}_i^{ext} = jk_0Z_0f_i^{ext} \qquad (2.27)$$

Equation (2.25) is termed the essential boundary condition and it must be explicitly enforced in the formulation. The essential boundary condition can be satisfied in two ways. The first way is to use identical basis functions for the exterior and interior fields at the surface. The second way is to use a coupling equation to relate the two fields. In the first case continuity is implicitly enforced and this is the method by which electric field continuity will be enforced in this dissertation. In addition, the basis functions are chosen to be identical to the testing function $\mathbf{Q}$; an approach known as Galerkin's technique. Therefore the unknown electric field is expanded as

$$\mathbf{E} = \sum_j E_j\mathbf{Q}_j \qquad (2.28)$$

Inserting (2.28) into (2.26), the system of linear equations

$$\sum_j E_j\left\{\int_V\frac{(\nabla\times\mathbf{Q}_i)\cdot(\nabla\times\mathbf{Q}_j)}{\mu_r} - k_0^2\epsilon_r\int_V\mathbf{Q}_i\cdot\mathbf{Q}_j\right.$$
$$\left.-k_0^2\int_S\int_{S'}\mathbf{Q}_i\cdot\left(\hat{\mathbf{n}}\times\overline{\overline{\mathbf{G}}}_{e2}\times\hat{\mathbf{n}}'\right)\cdot\mathbf{Q}_jdS'dS\right\} \;= f_i^{int} + \tilde{f}_i^{ext} \qquad (2.29)$$

is obtained. This equation may be separated into two parts, one representing the finite element portion, written as I_{ij}^{FE}, and the other representing the boundary integral portion, written as I_{ij}^{BI},

$$I_{ij}^{FE} = \int_V \frac{(\nabla \times \mathbf{Q}_i) \cdot (\nabla \times \mathbf{Q}_j)}{\mu_r} dV - k_0^2 \epsilon_r \int_V \mathbf{Q}_i \cdot \mathbf{Q}_j dV \tag{2.30}$$

$$I_{ij}^{BI} = -k_0^2 \int_S \int_{S'} \mathbf{Q}_i \cdot \left(\hat{\mathbf{n}} \times \overline{\overline{\mathbf{G}}}_{e2} \times \hat{\mathbf{n}}' \right) \cdot \mathbf{Q}_j dS' dS \tag{2.31}$$

These integrals represent the matrix entries in the following linear system

$$\begin{pmatrix} I_{ij}^{FE} + I_{ij}^{BI} & I_{ij}^{FE} \\ I_{ij}^{FE} & I_{ij}^{FE} \end{pmatrix} \{\mathbf{E}\} = \{\mathbf{f}\} \tag{2.32}$$

where $\{\mathbf{f}\}$ is calculated from f^{int} and $\tilde{f}^{ext}$ and $\{\mathbf{E}\}$ is a set of electric field coefficients to be determined. In this work, $\{\bullet\}$ indicates a Nx1 data vector while $(\bullet)$ represents an NxN matrix with N being the number of unknowns or free edges in the mesh.

2.2 Green's Function Singularity

While equations (2.30) and (2.31) correctly implement the equations needed for the FE-BI formulation, there is a subtle problem with equation (2.31) as it is currently written. There is a singularity in the Green's function

$$\overline{\overline{\mathbf{G}}}_{e2} = -\left(\bar{\bar{\mathbf{I}}} + \frac{\nabla\nabla}{k_0^2} \right) \frac{e^{-jk_0 R}}{2\pi R} \tag{2.33}$$

as R approaches zero, that is, as the source and field points coalesce. Furthermore, there are two derivative operations on this singularity making it a third order singularity. Any numerical evaluation technique that encounters this singularity will inevitably fail. Special consideration must therefore be given to this integral. This is a well known challenge in numerical solutions and is commonly referred to as the

self-cell case. In finite element programming this case occurs when the two surface integrals of (2.31) are evaluated over the same element or if adjacent elements have common integration points. In order to evaluate this integral properly, it must be manipulated into a form more suitable for numerical evaluation. To begin, equation (2.31) is broken up into two separate integrals

$$I_{ij}^{BI(1)} = \frac{k_0^2}{2\pi}\int_S\int_{S'}\left[\mathbf{Q}_i\cdot\left(\hat{\mathbf{z}}\times\bar{\bar{\mathbf{I}}}\times\hat{\mathbf{z}}\right)\cdot\mathbf{Q}_j\right]\frac{e^{-jk_0R}}{R}dS'dS \tag{2.34}$$

$$I_{ij}^{BI(2)} = \frac{1}{2\pi}\int_S\int_{S'}\left[\mathbf{Q}_i\cdot\left(\hat{\mathbf{z}}\times\nabla\nabla\frac{e^{-jk_0R}}{R}\times\hat{\mathbf{z}}\right)\cdot\mathbf{Q}_j\right]dS'dS \tag{2.35}$$

where $\hat{\mathbf{n}}$ has been replaced with $\hat{\mathbf{z}}$ since the planar surface is assumed to lie in the x-y plane. Using the dyadic identity

$$\begin{aligned}\bar{\bar{\mathbf{I}}} &= (\hat{\mathbf{x}}\hat{\mathbf{x}}+\hat{\mathbf{y}}\hat{\mathbf{y}})\\ \hat{\mathbf{z}}\times(\hat{\mathbf{x}}\hat{\mathbf{x}}+\hat{\mathbf{y}}\hat{\mathbf{y}})\times\hat{\mathbf{z}} &= -\hat{\mathbf{x}}\hat{\mathbf{x}}-\hat{\mathbf{y}}\hat{\mathbf{y}}\end{aligned} \tag{2.36}$$

(2.11) becomes

$$I_{ij}^{BI(1)} = -\frac{k_0^2}{2\pi}\int_S\int_{S'}\left[\mathbf{Q}_i\cdot(\hat{\mathbf{x}}\hat{\mathbf{x}}+\hat{\mathbf{y}}\hat{\mathbf{y}})\cdot\mathbf{Q}_j\right]\frac{e^{-jk_0R}}{R}dS'dS \tag{2.37}$$

$I_{ij}^{BI(1)}$ is now in a form which allows substitution of any user defined basis function. $I_{ij}^{BI(2)}$ requires a bit more manipulation to transform the third order singularity into something more suitable for numerical evaluation. Noting that the following identities are useful

$$\begin{aligned}\mathbf{Q}_i\cdot\hat{\mathbf{z}}\times\nabla\nabla\frac{e^{-jk_0R}}{R} &= -(\hat{\mathbf{z}}\times\mathbf{Q}_i)\cdot\nabla\nabla\frac{e^{-jk_0R}}{R}\\ \nabla\nabla\frac{e^{-jk_0R}}{R}\times\hat{\mathbf{z}}\cdot\mathbf{Q}_j &= \nabla\nabla\frac{e^{-jk_0R}}{R}\cdot(\hat{\mathbf{z}}\times\mathbf{Q}_j)\\ \nabla\nabla\frac{e^{-jk_0R}}{R} &= -\nabla\nabla'\frac{e^{-jk_0R}}{R}\end{aligned} \tag{2.38}$$

(2.39)

(2.12) can be written as

$$I_{ij}^{BI(2)} = \frac{1}{2\pi}\int_S\int_{S'}\left[(\hat{\mathbf{z}}\times\mathbf{Q}_i)\cdot\nabla\nabla'\frac{e^{-jk_0R}}{R}\cdot(\hat{\mathbf{z}}\times\mathbf{Q}_j)\right]dS'dS \tag{2.40}$$

Since the first gradient is not in terms of the primed (or source) coordinates, the primed integral may be taken inside this derivative giving

$$I_{ij}^{BI(2)} = \frac{1}{2\pi}\int_S(\hat{\mathbf{z}}\times\mathbf{Q}_i)\cdot\nabla\left[\int_{S'}\nabla'\frac{e^{-jk_0R}}{R}\cdot(\hat{\mathbf{z}}\times\mathbf{Q}_j)\,dS'\right]dS \tag{2.41}$$

Using the following vector identity

$$\nabla'\cdot\left[(\hat{\mathbf{z}}\times\mathbf{Q}_j)\frac{e^{-jk_0R}}{R}\right] = (\hat{\mathbf{z}}\times\mathbf{Q}_j)\cdot\nabla'\frac{e^{-jk_0R}}{R} + \frac{e^{-jk_0R}}{R}\nabla'\cdot(\hat{\mathbf{z}}\times\mathbf{Q}_j) \tag{2.42}$$

the primed integral becomes

$$\int_{S'}\nabla'\cdot\left[(\hat{\mathbf{z}}\times\mathbf{Q}_j)\frac{e^{-jk_0R}}{R}\right]dS' - \int_{S'}\left[\nabla'\cdot(\hat{\mathbf{z}}\times\mathbf{Q}_j)\right]\frac{e^{-jk_0R}}{R}dS' \tag{2.43}$$

and applying the divergence theorem to the first integral in (2.43)

$$\int_{S'}\nabla'\cdot\left[(\hat{\mathbf{z}}\times\mathbf{Q}_j)\frac{e^{-jk_0R}}{R}\right]dS' = \oint_C(\hat{\mathbf{z}}\times\mathbf{Q}_j)\frac{e^{-jk_0R}}{R}\cdot\mathbf{dl} \tag{2.44}$$

is obtained. The contour integral in the above expression corresponds to the integral along the edge of an element. In the interior of the geometry, an edge will be shared by two different elements. Due to the standard rules for basis functions the sign of this edge will be positive on one element and negative on the other element and the vector incremental distance **dl** will be in the opposite direction for the two integrals along that edge. In addition, the value of the two basis functions will be identical

along this edge. This means that any contour integral over that edge on one element will be exactly cancelled by the contour integral of that same edge on the element which shares that edge. Therefore, the total contribution of this contour integral on the surface will be zero for all interior edges. Also, any edges not shared by two elements will by definition be boundary edges located on the metal walls of the cavity. Since the method being studied involves a total electric field formulation there should be no field on a metallic edge and the contour integral for these edges is also zero. Therefore, it is argued that there will be exactly zero contribution from the contour integral in (2.44) and (2.41) becomes

$$I_{ij}^{BI(2)} = \frac{1}{2\pi}\int_S (\hat{\mathbf{z}} \times \mathbf{Q}_j) \cdot \nabla \left[\int_{S'} \nabla' \cdot (\hat{\mathbf{z}} \times \mathbf{Q}_j) \frac{e^{-jk_0 R}}{R} dS' \right] dS \tag{2.45}$$

Bringing the unprimed gradient into the primed integral, swapping the order of integration and using the same argument as above (2.45) can be written

$$I_{ij}^{BI(2)} = \frac{1}{2\pi}\int_S \int_{S'} \nabla \cdot (\hat{\mathbf{z}} \times \mathbf{Q}_i) \nabla' \cdot (\hat{\mathbf{z}} \times \mathbf{Q}_j) \frac{e^{-jk_0 R}}{R} dS' dS \tag{2.46}$$

which is now in a form suitable for numerical evaluation. Though the singularity still exists in the integral it is now of first order, making it integrable. During actual computation of this integral the self-cell case must still be treated differently if a numerical integration technique is used that will allow R to become zero. Special techniques such as those shown in [21] can be used to perform the integration in a way that avoids the singularity problem.

In the interest of symmetry the FE equation is also broken up into two equations and the final form of the general FE-BI equations is shown below where $I_{ij}^{FE} =$

$I_{ij}^{FE(1)} + I_{ij}^{FE(2)}$ and $I_{ij}^{BI} = I_{ij}^{BI(1)} + I_{ij}^{BI(2)}$

$$I_{ij}^{FE(1)} = \frac{1}{\mu_r} \int_V (\nabla \times \mathbf{Q}_i) \cdot (\nabla \times \mathbf{Q}_j)\, dV \tag{2.47}$$

$$I_{ij}^{FE(2)} = -k_0^2 \epsilon_r \int_V \mathbf{Q}_i \cdot \mathbf{Q}_j dV \tag{2.48}$$

$$I_{ij}^{BI(1)} = -\frac{k_0^2}{2\pi} \int_S \int_{S'} [\mathbf{Q}_i \cdot (\hat{\mathbf{x}}\hat{\mathbf{x}} + \hat{\mathbf{y}}\hat{\mathbf{y}}) \cdot \mathbf{Q}_j] \frac{e^{-jk_0 R}}{R} dS' dS \tag{2.49}$$

$$I_{ij}^{BI(2)} = \frac{1}{2\pi} \int_S \int_{S'} \nabla \cdot (\hat{\mathbf{z}} \times \mathbf{Q}_i)\, \nabla' \cdot (\hat{\mathbf{z}} \times \mathbf{Q}_j) \frac{e^{-jk_0 R}}{R} dS' dS \tag{2.50}$$

2.3 Excitation

The equations above describe the FE-BI system itself. Solution of the electric field coefficients requires that the system be excited in some fashion. There are numerous ways to excite the FE-BI system. These excitation sources can be both internal and external. The most popular external source is the plane wave since any general electromagnetic wave can be modeled by a superposition of plane waves. Internal sources are composed of either electric or magnetic currents. Models such as probe feeds, voltage gap feeds, and coaxial cable feeds can be used as internal sources. For the sake of simplicity this thesis will present only the formulations for probe feed excitation and plane wave excitation. For a more complete description of different types of sources, both internal and external, see [2].

2.3.1 Probe Feeds

Probe feeds are a simple model used to give the FE-BI system some impetus to which to respond. Although a somewhat limited model, probe feeds do a sufficient job of exciting the system and give a decent approximation to a real feed model such as a coaxial cable. Probe feeds are often used because they are exceedingly simple to

implement. Looking at the interior excitation equation,

$$f_i^{int} = -\int_V \mathbf{Q}_i \cdot \left[jk_0 Z_0 \mathbf{J} - \frac{\nabla \times \mathbf{M}}{\mu_r} \right] dV \tag{2.51}$$

probe feeds correspond to a **J** type current, therefore **M** is equal to zero. For this dissertation it is assumed that feeds will be z-directed only. Therefore

$$\mathbf{J} = \hat{\mathbf{z}} I_0 \tag{2.52}$$

It is also assumed that all feeds will be located directly on an element edge. Since the value of the basis function along an edge is one for that edge and zero for all other edges in the element the following is true

$$\int_V \mathbf{Q}_i \cdot \hat{\mathbf{z}} = l_i \tag{2.53}$$

where l_i is the length of the edge on which the feed is located. Using (2.52) and (2.53) in (2.51) the internal excitation can be written as

$$f_i^{int} = -jk_0 Z_0 I_0 l_i \tag{2.54}$$

2.3.2 Plane Waves

Recalling the definition of exterior excitation,

$$\tilde{f}^{ext} = jk_0 Z_0 \int_S \mathbf{Q}_i \cdot \left[\hat{\mathbf{n}} \times \left(\mathbf{H}^{inc} + \mathbf{H}^{ref} \right) \right] dS \tag{2.55}$$

A general plane wave can be written as

$$\mathbf{E}^{inc} = \hat{\mathbf{e}}^i e^{-jk_0(\hat{\mathbf{k}}^i \cdot \mathbf{r})}$$

$$\begin{aligned} \mathbf{H}^{inc} &= Y_0 \left[\hat{\mathbf{k}}^i \times \mathbf{E}^{inc}\right] \\ \mathbf{E}^{ref} &= \hat{\mathbf{e}}^r e^{-jk_0(\hat{\mathbf{k}}^r \cdot \mathbf{r})} \\ \mathbf{H}^{ref} &= Y_0 \left[\hat{\mathbf{k}}^r \times \mathbf{E}^{ref}\right] \end{aligned} \tag{2.56}$$

where $\hat{\mathbf{e}}^i$ is the polarization angle of the incident field, $\hat{\mathbf{k}}^i$ is the incident field direction of propagation, $\hat{\mathbf{e}}^r$ is the polarization angle of the reflected field, and $\hat{\mathbf{k}}^r$ is the reflected field direction of propagation. For the case of an aperture cut in an infinite ground plane located at z = 0, the sum of the incident and reflected magnetic fields is twice that of the incident field and $\hat{\mathbf{n}} = \hat{\mathbf{z}}$.

$$\tilde{f}^{ext} = j2k_0 \int_S \mathbf{Q}_i \cdot \hat{\mathbf{z}} \times \left[\left(\hat{\mathbf{k}}^i \times \hat{\mathbf{e}}^i\right) e^{-jk_0(\hat{\mathbf{k}}^i \cdot \mathbf{r})}\right] dS \tag{2.57}$$

Using the following relations

$$\hat{\mathbf{k}}^i = -\hat{\mathbf{r}}^i \tag{2.58}$$

$$\hat{\mathbf{e}}^i = \cos\alpha\hat{\theta}^i + \sin\alpha\hat{\phi}^i \tag{2.59}$$

$$\hat{\mathbf{k}}^i \times \hat{\mathbf{e}}^i = \sin\alpha\hat{\theta}^i - \cos\alpha\hat{\phi}^i \tag{2.60}$$

$$\begin{aligned} \hat{\mathbf{z}} \times \left(\hat{\mathbf{k}}^i \times \hat{\mathbf{e}}^i\right) = \sin\alpha &\left(\hat{\mathbf{y}}\cos\theta^i\cos\phi^i - \hat{\mathbf{x}}\cos\theta^i\sin\phi^i\right) \\ &+ \cos\alpha\left(\hat{\mathbf{y}}\sin\phi^i + \hat{\mathbf{x}}\cos\phi^i\right) \end{aligned} \tag{2.61}$$

$$\hat{\mathbf{k}}^i \cdot \mathbf{r} = \sin\theta^i\left(x\cos\phi^i + y\sin\phi^i\right) \tag{2.62}$$

the general equation for plane wave excitation can be written as

$$\begin{aligned} \tilde{f}^{ext} = j2k_0 \int_S \mathbf{Q}_i \cdot \Big[&\sin\alpha\left(\hat{\mathbf{y}}\cos\theta^i\cos\phi^i - \hat{\mathbf{x}}\cos\theta^i\sin\phi^i\right) \\ &+ \cos\alpha\left(\hat{\mathbf{y}}\sin\phi^i + \hat{\mathbf{x}}\cos\phi^i\right)\Big] e^{jk_0(\hat{\mathbf{r}}^i \cdot \mathbf{r})} dS \end{aligned} \tag{2.63}$$

2.4 Post Processing

While the main goal of the above development is to solve for the electric field coefficients for each edge in the FE-BI system, these coefficients by themselves do not represent commonly required information. Once the field coefficients are found they can be used to model a multitude of different antenna properties such as input impedance, radar cross section, antenna pattern, and antenna field pattern making it possible to model both active and passive antennas. This dissertation considers both passive and active antenna characteristics although the use Babinet's principle [23] allows passive antenna characteristics to be predicted from active antenna modeling and vice-versa.

2.4.1 Input Impedance

Input impedance is an important measure of an antenna's performance. No matter how well-behaved the antenna's radiation pattern is, if all the energy used to excite the antenna is reflected back into the power source the antenna has limited utility. Therefore, it is important to study the input impedance of an antenna so that the antenna's impedance may be matched to that of the excitation source. In section 2.4 it was stated that only probe feeds are going to be used as excitation devices for this dissertation, so in turn; only the input impedance of probe feeds need be modeled. Similar to the modeling of the excitation of a probe feed, input impedance modeling of a probe feed is quite simple. Once the electric field coefficients for each of the edges is found, the impedance of a probe feed located on the ith edge is found from this simple equation

$$Z_{in} = \frac{E_i l_i}{I_0} \tag{2.64}$$

where E_i is the electric field coefficient of the edge, l_i is the length of the edge, and I_0 is the excitation of the feed. The numerator of (2.64) is recognized as the voltage across the probe assuming the probe length l_i is small compared to a wavelength.

The reflection coefficient (Γ) and standing wave ratio (SWR) can then be found from

$$\Gamma = \frac{Z_{in} - 50.0}{Z_{in} + 50.0} \tag{2.65}$$

$$SWR = \frac{1.0 + |\Gamma|}{1.0 - |\Gamma|} \tag{2.66}$$

assuming a feed port resistance of 50Ω.

2.4.2 Radar Cross Section and Radiation Pattern

The analysis discussed thus far can only predict the near field distributions. These near field distributions, however, can be used to obtain certain far field characteristics, such as radar cross section (RCS) and radiation pattern. RCS is a measure of how a passive antenna scatters and directs electromagnetic energy while radiation pattern is a measure of how well an active antenna directs energy in a specified direction, with respect to an isotropic radiator. RCS and radiation pattern are developed in three dimensional spherical coordinates θ and ϕ and are closely linked. In fact both can be computed from the definition of RCS [11].

$$\sigma = \lim_{r \to \infty} 4\pi r^2 \frac{|\mathbf{E}^{sc}(r)|^2}{|\mathbf{E}^{inc}(r)|^2} \tag{2.67}$$

The far-zone magnetic field can be written as

$$\mathbf{H}^{sc}(r) = jk_0 Y_0 \frac{e^{-jk_0 r}}{2\pi r} \int_S \left(\hat{\theta}\hat{\theta} + \hat{\phi}\hat{\phi}\right) \cdot \left[\hat{\mathbf{z}} \times \mathbf{E}(x', y')\right] e^{jk_0(\hat{\mathbf{r}} \cdot \mathbf{r}')} dS \tag{2.68}$$

Let

$$\mathbf{I} = \int_S \left(\hat{\theta}\hat{\theta} + \hat{\phi}\hat{\phi}\right) \cdot \left[\hat{\mathbf{z}} \times \mathbf{E}(x', y')\right] e^{jk_0(\hat{\mathbf{r}} \cdot \mathbf{r}')} dS = I_\theta \hat{\theta} + I_\phi \hat{\phi} \tag{2.69}$$

Taking the magnitude of (2.68)

$$|\mathbf{H}^{sc}| = \frac{k_0 Y_0}{2\pi r} |\mathbf{I}| = \frac{k_0 Y_0}{2\pi r} \sqrt{\mathbf{I} \cdot \mathbf{I}^*} \tag{2.70}$$

where the (*) indicates the complex conjugate. Assuming an incident plane wave of unit amplitude

$$\left|\mathbf{H}^{inc}\right| = Y_0 \tag{2.71}$$

Without loss of generality, (2.70) and (2.71) may be inserted into (2.67)

$$\sigma = \frac{k_0^2}{\pi}\left(\mathbf{I}\cdot\mathbf{I}^*\right) \tag{2.72}$$

In practice, two components of RCS are defined

$$\sigma_\theta = \frac{k_0^2}{\pi}\left|I_\theta^2\right| \tag{2.73}$$

$$\sigma_\phi = \frac{k_0^2}{\pi}\left|I_\phi^2\right| \tag{2.74}$$

When the polarization of the incident plane wave is in the θ direction (i.e. $\mathbf{E}^i = \hat{\theta}^i E_i$), $\sigma_{\theta\theta}$ is termed the co-polarized component of the RCS and $\sigma_{\theta\phi}$ is termed the cross-polarized component of the RCS. Similarly, when the polarization of the incident plane wave is in the ϕ direction (i.e. $\mathbf{E}^i = \hat{\phi}^i E_i$), the co-polarized component is computed from $\sigma_{\phi\phi}$ while the cross-polarized component is computed from $\sigma_{\phi\theta}$.

To compute the radiation pattern of the antenna the RCS is normalized by the input power of the antenna.

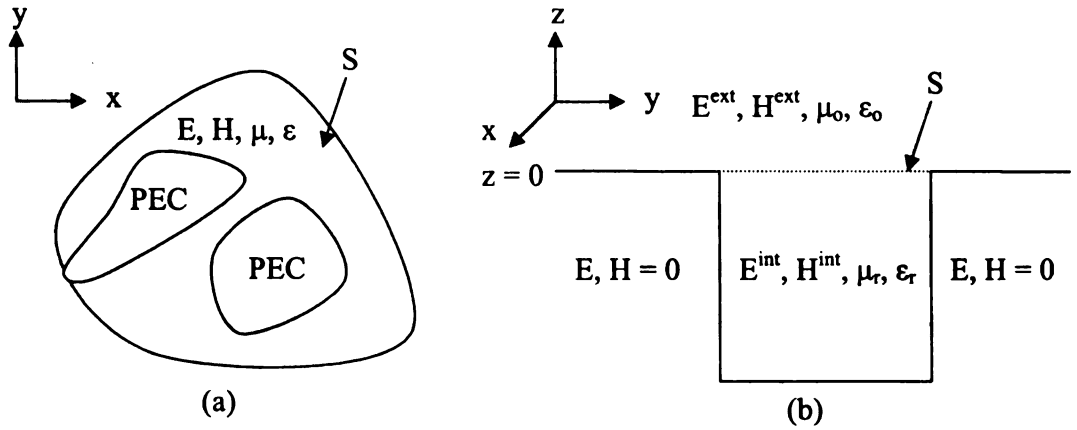

Figure 2.1. Cavity backed aperture in a ground plane (a) top view, (b) side view

CHAPTER 3

RIGHT PRISM ELEMENTS

In this chapter a set of sub-domain basis functions are defined for right prism elements and the equations from chapter 2 are specialized to these functions. These equations will represent the final form of the equations needed to correctly implement the FE-BI method using prism elements in a computer program.

3.1 FE-BI Formulation

The equations derived in chapter 2 are general in a sense that they don't relate to a specific element type. To continue the formulation, an element type must be chosen and a set of sub-domain basis functions must be defined for that element. The basis functions need to possess a few properties in order to be usable. They must at least have constant tangential/linear normal (CT/LN) expansion properties, and they must enforce tangential field continuity across element faces. Expansion functions that at CT/LN have a constant curl within the domain of the element while lower order expansions have a curl necessitating the use of Dirac delta functions that are not suitable for numerical implementation. They also must be divergence free and curl conforming. Divergence free basis functions are necessary to avoid the build up of fictitious charge within the element, and curl conforming basis functions are needed to avoid spurious eigenvalue solutions. A formulation for prism elements has previously been done by [24].

3.1.1 Basis Functions

A right prism, usually just called a prism, is defined as an element that has a constant cross-sectional area throughout its extrusion. Prisms have the ability to provide great flexibility in modeling geometries that are irregular in two dimensions but regular in

the third dimension, such as cavities recessed in ground planes. Prisms also have the advantage of making it fairly simple to create a three dimensional mesh from a two dimensional mesh composed of triangles. Creating the two dimensional mesh for an arbitrarily shaped surface is quite difficult and beyond the scope of this dissertation but fortunately many commercial two dimensional mesh generation software packages exist and can be used without much difficulty. Prisms do, however, have the disadvantage of possibly over sampling the geometry, especially in the aperture area.

Figure 3.1 depicts a right prism element where the edges for the prisms are defined in Table 3.1.

Edge	Node 1	Node 2
1	2	3
2	3	1
3	1	2
4	5	6
5	6	4
6	4	5
7	1	4
8	2	5
9	3	6

Table 3.1. Local edge numbering scheme for a right prism element.

Notice that in the table the edge numbers for the horizontal edges are defined as being directly across from the node numbers. This is a bookkeeping procedure and will facilitate the use of the basis functions soon to be defined. Also, in the figure, the nodes are ordered in a counter clockwise direction. Numbering in this way is necessary in order to ensure that the normal vector of the element points towards the top of the element and therefore out of the cavity model. The edge based expansion functions for the prism element are derived from the standard Rao-Wilton-Glisson

(RWG) basis functions [25] for triangles and will be represented by the letter **W** instead of **Q** as in chapter 2. This change of notation will be important when the basis functions for prisms are coupled with the basis functions for hexahedrals in chapter 5. Multiplying the RWG basis functions by a function of the prism height gives the following equations

$$\begin{aligned}
\mathbf{W}_{\chi i} = \mathbf{V}_i &= \frac{(z - z_l)}{c} \frac{l_i s_i}{2S^e} \left[(x - x_i)\,\hat{\mathbf{y}} - (y - y_i)\,\hat{\mathbf{x}}\right] \quad \chi = 1,2,3 \\
\mathbf{W}_{\chi i} = \mathbf{M}_i &= \frac{(z_u - z)}{c} \frac{l_i s_i}{2S^e} \left[(x - x_i)\,\hat{\mathbf{y}} - (y - y_i)\,\hat{\mathbf{x}}\right] \quad \chi = 4,5,6 \\
\mathbf{W}_{\chi i} = \mathbf{K}_i &= \frac{\left[(x_{k1} y_{k2} - x_{k2} y_{k1}) + (y_{k1} - y_{k2})\,x + (x_{k2} - x_{k1})\,y\right]}{2S^e} \hat{\mathbf{z}} \quad \chi = 7,8,9
\end{aligned} \tag{3.1}$$

where l_i = length of the ith edge, s_i = sign of the ith edge , (x_i, y_i) = node positions across from the ith edge, (z_l, z_u) = bottom and top positions of the prism, c = height of the prism, S^e = Area of the triangle that forms the top (or bottom) of the prism, and k_1 and k_2 are defined in Table 3.2.

χ	k_1	k_2
7	2	3
8	3	1
9	1	2

Table 3.2. Vertical edge constants for right prisms

The curl of each of these basis functions are found to be

$$\begin{aligned}
\nabla \times \mathbf{V}_i &= -\frac{l_i s_i}{2cS^e} \left[(x - x_i)\,\hat{\mathbf{x}} + (y - y_i)\,\hat{\mathbf{y}} - 2\,(z - z_l)\,\hat{\mathbf{z}}\right] \\
\nabla \times \mathbf{M}_i &= \frac{l_i s_i}{2cS^e} \left[(x - x_i)\,\hat{\mathbf{x}} + (y - y_i)\,\hat{\mathbf{y}} + 2\,(z_u - z)\,\hat{\mathbf{z}}\right] \\
\nabla \times \mathbf{K}_i &= \frac{\left[(x_{k2} - x_{k1})\,\hat{\mathbf{x}} + (y_{k1} - y_{k2})\,\hat{\mathbf{y}}\right]}{2S^e}
\end{aligned} \tag{3.2}$$

3.1.2 FE Matrix Entries

The FE matrix entries are defined by equations (2.47) and (2.49) which are reproduced here, with the vector basis function now represented with the symbol $\mathbf{W}$.

$$I_{ij}^{FE(1)} = \frac{1}{\mu_r} \int_V (\nabla \times \mathbf{W}_i) \cdot (\nabla \times \mathbf{W}_j)\, dV \tag{3.3}$$

$$I_{ij}^{FE(2)} = -k_0^2 \epsilon_r \int_V \mathbf{W}_i \cdot \mathbf{W}_j dV \tag{3.4}$$

Using the basis functions and their curls defined in (3.1) and (3.2) closed form expressions can be defined for the integrals shown in (3.3) and (3.4). First, some integrals over triangles are defined [26], for convenience:

$$\begin{aligned}
S^x &= \int_S x dS = S^y = \int_S y dS = 0 \\
S^e &= \int_S dS = \frac{1}{2}\left[x_2(y_3 - y_1) - x_1(y_3 - y_2) - x_3(y_2 - y_1)\right] \\
S^{xx} &= \int_S x^2 dS = \frac{S^e}{12}\left[x_1^2 + x_2^2 + x_3^2\right] \\
S^{yy} &= \int_S y^2 dS = \frac{S^e}{12}\left[y_1^2 + y_2^2 + y_3^2\right] \\
S^{xy} &= \int_S xy dS = \frac{S^e}{12}\left[x_1 y_1 + x_2 y_2 + x_3 y_3\right]
\end{aligned} \tag{3.5}$$

Also defining the following integrals will be useful

$$\begin{aligned}
I_s^x &= I_s^y = \int_S (x - a) dS = \int_S (y - a) dS = -aS^e \\
I_s^{xx} &= \int_S (x - a)(x - b) dS = S^{xx} + abS^e \\
I_s^{yy} &= \int_S (y - a)(y - b) dS = S^{yy} + abS^e \\
I_l^{zz} &= \int_{z_l}^{z_u} (z - a)(z - b) dl = \left(\frac{z_u^3 - z_l^3}{3}\right) - \left(\frac{z_l^2 - z_u^2}{2}\right) + ab(z_u - z_l)
\end{aligned} \tag{3.6}$$

With the integrals defined in (3.5) and (3.6), closed form expressions can be found readily for the integrals shown in (3.3) and (3.4). The closed form expressions for

(3.3) are written as

$$I_{ij}^{FE} = \frac{l_i l_j s_i s_j}{\mu_r (2cS^e)^2}[cI_s^{xx} + cI_s^{yy} + 4S^e I_l^{zz}] \qquad Top - Top \tag{3.7}$$

$$I_{ij}^{FE} = -\frac{l_i l_j s_i s_j}{\mu_r (2cS^e)^2}[cI_s^{xx} + cI_s^{yy} + 4S^e I_l^{zz}] \qquad Top - Bottom \tag{3.8}$$

$$I_{ij}^{FE} = -\frac{l_i s_i}{\mu_r (2S^e)^2}[(x_{k2}^j - x_{k1}^j)I_s^x + (y_{k2}^j - y_{k1}^j)I_s^y] \qquad Top - Side \tag{3.9}$$

$$I_{ij}^{FE} = -\frac{l_i l_j s_i s_j}{\mu_r (2cS^e)^2}[cI_s^{xx} + cI_s^{yy} + 4S^e I_l^{zz}] \qquad Top - Top \tag{3.10}$$

$$I_{ij}^{FE} = \frac{l_i l_j s_i s_j}{\mu_r (2cS^e)^2}[cI_s^{xx} + cI_s^{yy} + 4S^e I_l^{zz}] \qquad Top - Bottom \tag{3.11}$$

$$I_{ij}^{FE} = \frac{l_i s_i}{\mu_r (2S^e)^2}[(x_{k2}^j - x_{k1}^j)I_s^x + (y_{k2}^j - y_{k1}^j)I_s^y] \qquad Bottom - Side \tag{3.12}$$

$$I_{ij}^{FE} = -\frac{l_j s_j}{\mu_r (2S^e)^2}[(x_{k2}^i - x_{k1}^i)I_s^x + (y_{k2}^i - y_{k1}^i)I_s^y] \qquad Side - Top \tag{3.13}$$

$$I_{ij}^{FE} = \frac{l_j s_j}{\mu_r (2S^e)^2}[(x_{k2}^i - x_{k1}^i)I_s^x + (y_{k2}^i - y_{k1}^i)I_s^y] \qquad Side - Bottom \tag{3.14}$$

$$I_{ij}^{FE} = \frac{c}{\mu_r (4S^e)}[(x_{k2}^i - x_{k1}^i)(x_{k2}^j - x_{k1}^j) + (y_{k2}^i - y_{k1}^i)(y_{k2}^j - y_{k1}^j)] \quad Side - Side \tag{3.15}$$

In these, the notation 'Top-Top' refers to the interactions between a testing edge on the top of the prism and a source edge on the top face of the prism. The notation 'Top-Bottom', etc. have similar interpretations. The closed form integrals for equation (3.4) are written as

$$I_{ij}^{FE} = \frac{\epsilon_r l_i l_j s_i s_j}{(2cS^e)^2}[I_s^{xx} I_l^{zz} + I_s^{yy} I_l^{zz}] \qquad Top - Top \tag{3.16}$$

$$I_{ij}^{FE} = -\frac{\epsilon_r l_i l_j s_i s_j}{(2cS^e)^2}[I_s^{xx} I_l^{zz} + I_s^{yy} I_l^{zz}] \qquad Top - Bottom \tag{3.17}$$

$$I_{ij}^{FE} = 0 \qquad Top - Side \tag{3.18}$$

$$I_{ij}^{FE} = -\frac{\epsilon_r l_i l_j s_i s_j}{(2cS^e)^2}[I_s^{xx} I_l^{zz} + I_s^{yy} I_l^{zz}] \qquad Bottom - Top \tag{3.19}$$

$$I_{ij}^{FE} = \frac{\epsilon_r l_i l_j s_i s_j}{(2cS^e)^2}[I_s^{xx} I_l^{zz} + I_s^{yy} I_l^{zz}] \qquad Bottom - Bottom \tag{3.20}$$

$$I_{ij}^{FE} = 0 \qquad Bottom - Top \tag{3.21}$$

$$I_{ij}^{FE} = 0 \qquad Bottom - Side \tag{3.22}$$

$$I_{ij}^{FE} = 0 \qquad Side - Top \tag{3.23}$$

$$I_{ij}^{FE} = 0 \qquad Side - Bottom \tag{3.24}$$

$$I_{ij}^{FE} = \frac{c}{(2S^e)^2}[C_iC_jS^e + (X_iY_j + X_jY_i)S^{xy} + Y_iY_jS^{xx} + X_iX_jS^{yy}] \; Side - Side \tag{3.25}$$

where

$$C_i = (x_{k1}^i y_{k2}^i - x_{k2}^i y_{k1}^i)$$

$$C_j = (x_{k1}^j y_{k2}^j - x_{k2}^j y_{k1}^j)$$

$$X_i = (x_{k2}^i - x_{k1}^i)$$

$$X_j = (x_{k2}^j - x_{k1}^j)$$

$$Y_i = (y_{k2}^i - y_{k1}^i)$$

$$Y_i = (y_{k2}^j - y_{k1}^j)$$

3.1.3 BI Matrix Entries

The equations for the BI matrix entries were shown in (2.49) and (2.50) which are reproduced here, with the basis function now represented by **W**.

$$I_{ij}^{BI(1)} = -\frac{k_0^2}{2\pi}\int_S\int_{S'}[\mathbf{W}_i \cdot (\hat{\mathbf{x}}\hat{\mathbf{x}} + \hat{\mathbf{y}}\hat{\mathbf{y}}) \cdot \mathbf{W}_j]\frac{e^{-jk_0R}}{R}dS'dS \tag{3.26}$$

$$I_{ij}^{BI(2)} = \frac{1}{2\pi}\int_S\int_{S'}\nabla \cdot (\hat{\mathbf{z}} \times \mathbf{W}_i)\,\nabla' \cdot (\hat{\mathbf{z}} \times \mathbf{W}_j)\frac{e^{-jk_0R}}{R}dS'dS \tag{3.27}$$

In order to enforce tangential electric field continuity across the surface, the basis functions for the interior and exterior fields must be identical at the surface. Therefore, the three dimensional basis functions for the prisms must be compressed into two dimensional triangular basis functions. By removing the axial (z) dependence of

the **V** basis functions in (3.1), the two dimensional basis functions are found to be

$$\mathbf{W}_i = \frac{l_i s_i}{2S^e}\left[(x - x_i)\,\hat{\mathbf{y}} - (y - y_i)\,\hat{\mathbf{x}}\right] \tag{3.28}$$

Inserting in the definition of the two dimensional basis functions into the BI equations, (2.49) becomes

$$I_{ij}^{BI(1)} = -\frac{l_i l_j s_i s_j k_0^2}{8\pi S_i^e S_j^e}\int_S\int_{S'}[(x - x_i)(x' - x_j) + (y - y_i)(y' - y_j)]\frac{e^{-jk_0R}}{R}dS'dS \tag{3.29}$$

and after computing

$$\nabla \cdot (\hat{\mathbf{z}} \times \mathbf{W}_i) = -\frac{l_i s_i}{S_i^e} \tag{3.30}$$

(2.50) becomes

$$I_{ij}^{BI(2)} = \frac{l_i l_j s_i s_j}{2\pi}\int_S\int_{S'}\frac{e^{-jk_0R}}{R}dS'dS \tag{3.31}$$

Unlike the equations for the FE matrix these integrals do not have closed form expressions, they must be evaluated numerically. Any suitable numerical integration technique could be used and some well known algorithms can be found in [27]. The only consideration needed when using these integration routines is that the self-cell case must be treated differently than other cases as the singularity in the Green's function will cause these routines to fail.

3.1.4 Plane Wave Excitation

Inserting in the definition of the two-dimensional basis functions into the plane wave excitation equation from chapter 2, (2.63) becomes

$$\begin{aligned}\tilde{f}_i^{ext} = \frac{jk_0 d_i s_i}{S^e}\int_S \Big[&(x - x_i)\left(\sin\alpha\cos\theta^i\cos\phi^i + \cos\alpha\sin\phi^i\right)\\ &+ (y - y_i)\left(\sin\alpha\cos\theta^i\sin\phi^i - \cos\alpha\cos\phi^i\right)\Big]\, e^{jk_0\left(\hat{\mathbf{r}}^i\cdot\mathbf{r}\right)}dS\end{aligned} \tag{3.32}$$

This equation, like the BI equations, must be computed numerically. Unlike the BI equations however, there is no singularity to worry about so its implementation is fairly straightforward.

3.1.5 Radiation Pattern

Using the expansion of the electric field in terms of the prism basis functions in (2.69)

$$\mathbf{I} = \sum_j \left\{ -\frac{E_j l_j s_j}{2S_j^e} \int_{S'} \left(\hat{\theta}\hat{\theta} + \hat{\phi}\hat{\phi}\right) \cdot \left[(x' - x_j)\hat{\mathbf{y}} - (y' - y_j)\hat{\mathbf{x}}\right] e^{jk_0(\hat{\mathbf{r}}\cdot\mathbf{r}')} dS \right\} \tag{3.33}$$

Noticing

$$\begin{aligned}
(\hat{\theta} \cdot \hat{\mathbf{x}}) &= \cos(\theta)\cos(\phi) \\
(\hat{\phi} \cdot \hat{\mathbf{x}}) &= -\sin(\phi) \\
(\hat{\theta} \cdot \hat{\mathbf{y}}) &= \cos(\theta)\sin(\phi) \\
(\hat{\phi} \cdot \hat{\mathbf{y}}) &= \cos(\phi)
\end{aligned}$$

The radiation pattern is then written as

$$I_\theta = \sum_j \left\{ -\frac{E_j l_j s_j}{2S_j^e} \int_{S'} \left[(x' - x_j)\cos\theta\cos\phi + (y' - y_j)\cos\theta\sin\phi\right] e^{jk_0(\hat{\mathbf{r}}\cdot\mathbf{r}')} dS \right\} \tag{3.34}$$

$$I_\phi = \sum_j \left\{ -\frac{E_j l_j s_j}{2S_j^e} \int_{S'} \left[(x' - x_j)\sin\phi - (y' - y_j)\cos\phi\right] e^{jk_0(\hat{\mathbf{r}}\cdot\mathbf{r}')} dS \right\} \tag{3.35}$$

where

$$e^{jk_0(\hat{\mathbf{r}}\cdot\mathbf{r}')} = e^{jk_0\sin\theta(x'\cos\phi + y'\sin\phi)}$$

3.2 Examples

To validate the formulation a software program, codename Morfeus, was developed and used to model a couple simple geometries. The results were compared to existing results generated the Prism program (courtesy of Dr. Leo Kempel).

3.2.1 Rectangular Slot Antenna

To begin, the radar cross section of a 6cm x 5cm x 2cm cavity with a centered 3cm x 2cm slot aperture was computed. The cavity was assumed to be filled with a material having a dielectric constant of $\epsilon_r = 2.17$. Figure 3.2 and Figure 3.3 show the radar cross section of the slot aperture at normal incidence. The first figure shows the results where the incident electric field has no ϕ component while the second figure shows the results where the incident electric field has no θ component.

3.2.2 Rectangular Patch Antenna

Next a 6cm x 4cm x 0.0762cm cavity with a centered 3cm x 2cm patch antenna was modeled. The cavity was filled with a material having a dielectric constant of $\epsilon_r = 3.2$. The patch was excited with a probe feed at approximately x = 3.0cm, y = 2.1cm. Both the radiation pattern at 5.4 GHz and the input impedance for 4-7 GHz were modeled and again compared to results generated by the Prism program. Figure 3.4 shows the radiation pattern results while Figure 3.5 shows the results of the input impedance calculations.

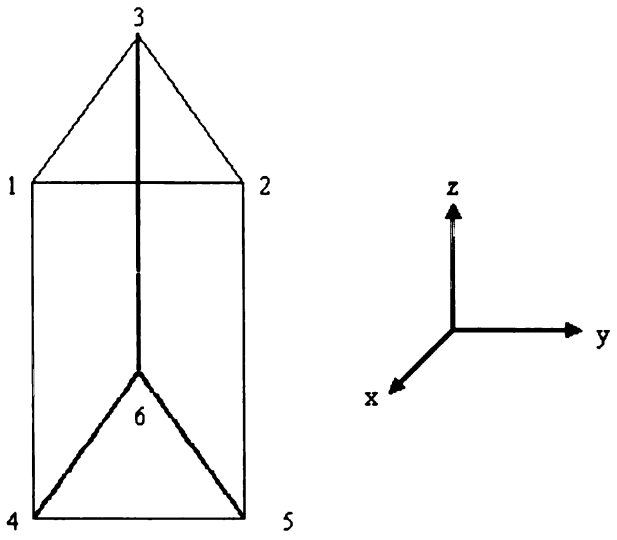

Figure 3.1. Right prism shown with its defined node numbering

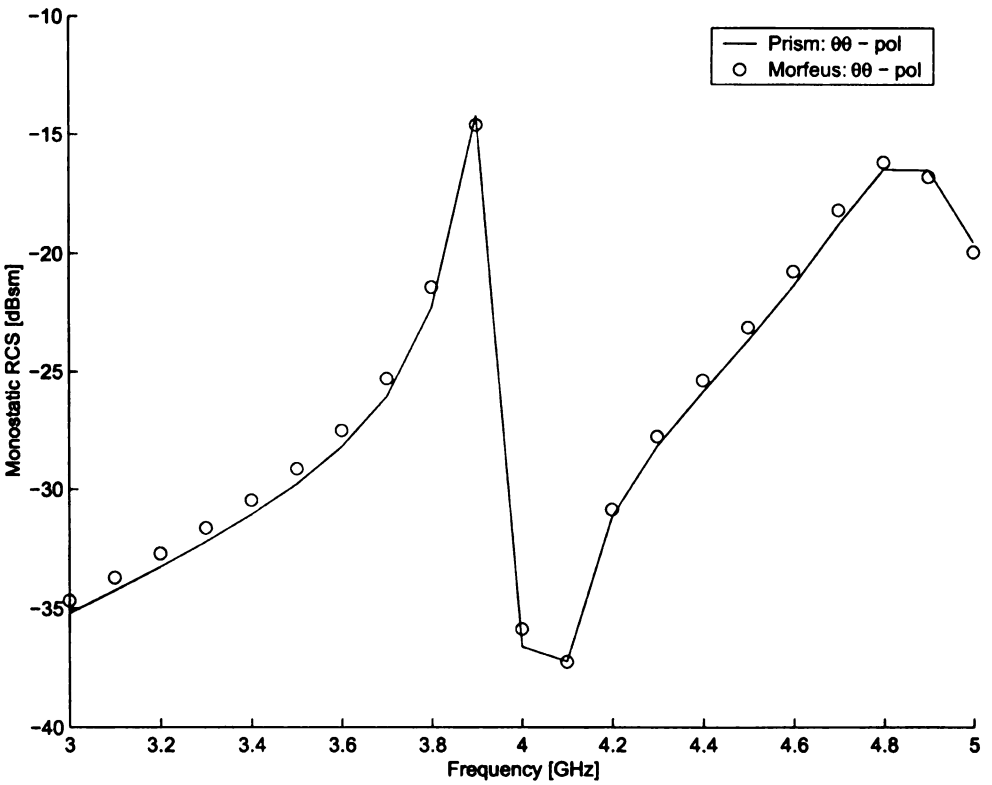

Figure 3.2. Radar cross section of a 6cm x 5cm x 2cm cavity with a 3cm x 2cm slot aperture, $\theta\theta$-pol

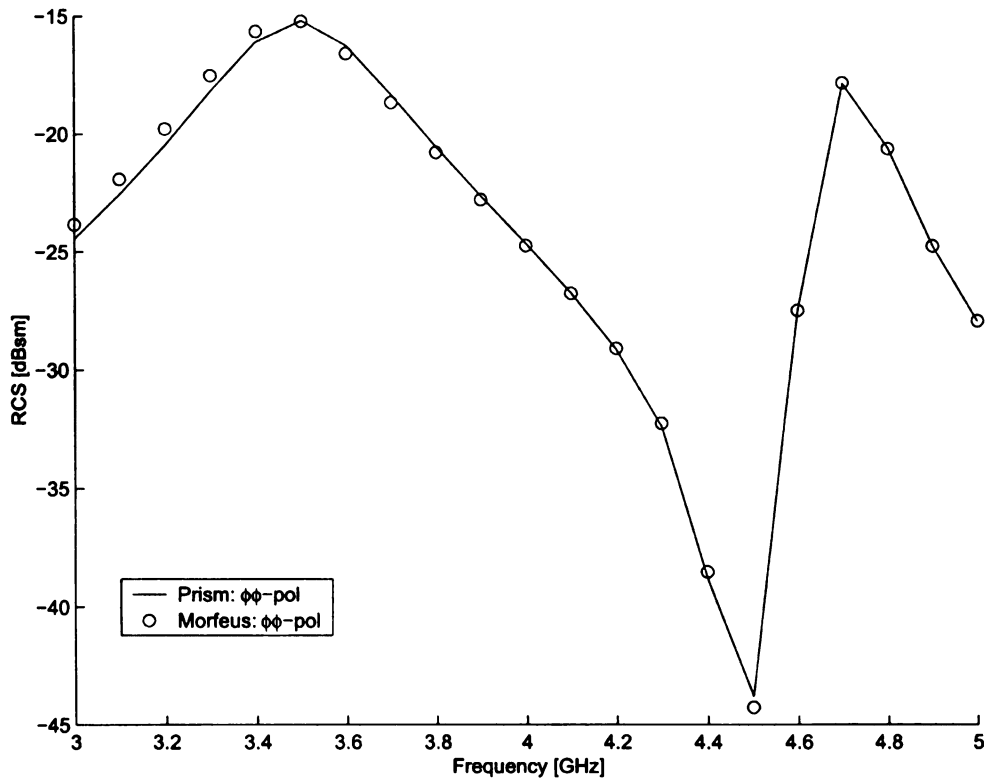

Figure 3.3. Radar cross section of a 6cm x 5cm x 2cm cavity with a 3cm x 2cm slot aperture, $\phi\phi$-pol

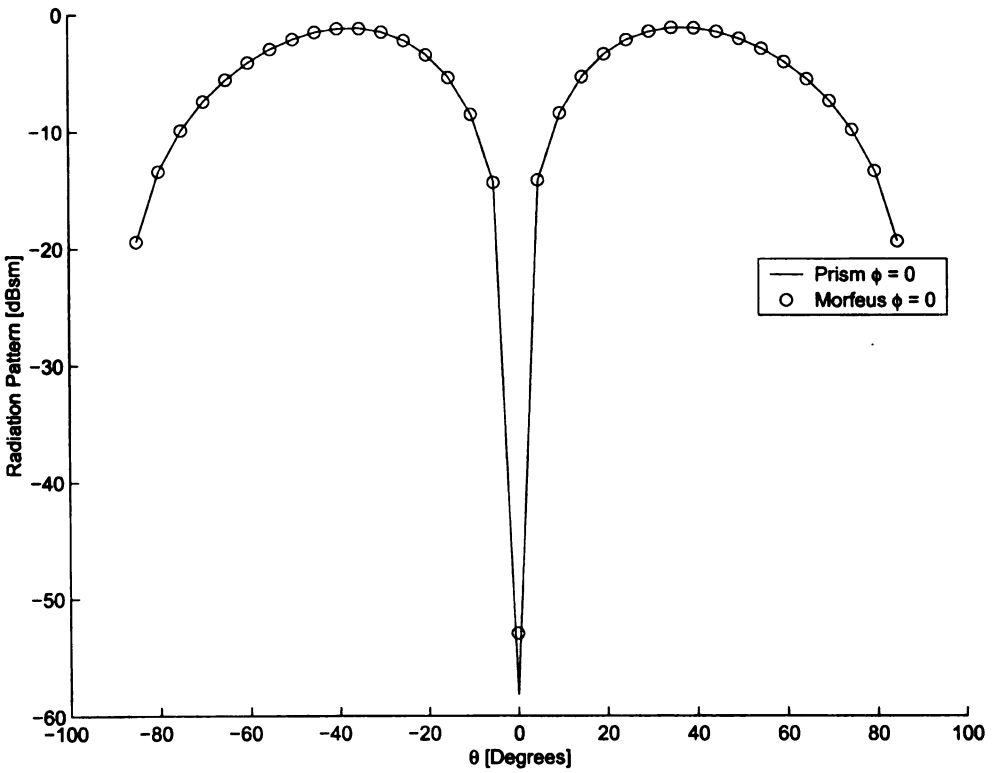

Figure 3.4. Radiation patten of a 6cm x 4cm x 0.0762cm cavity with a 3cm x 2cm patch

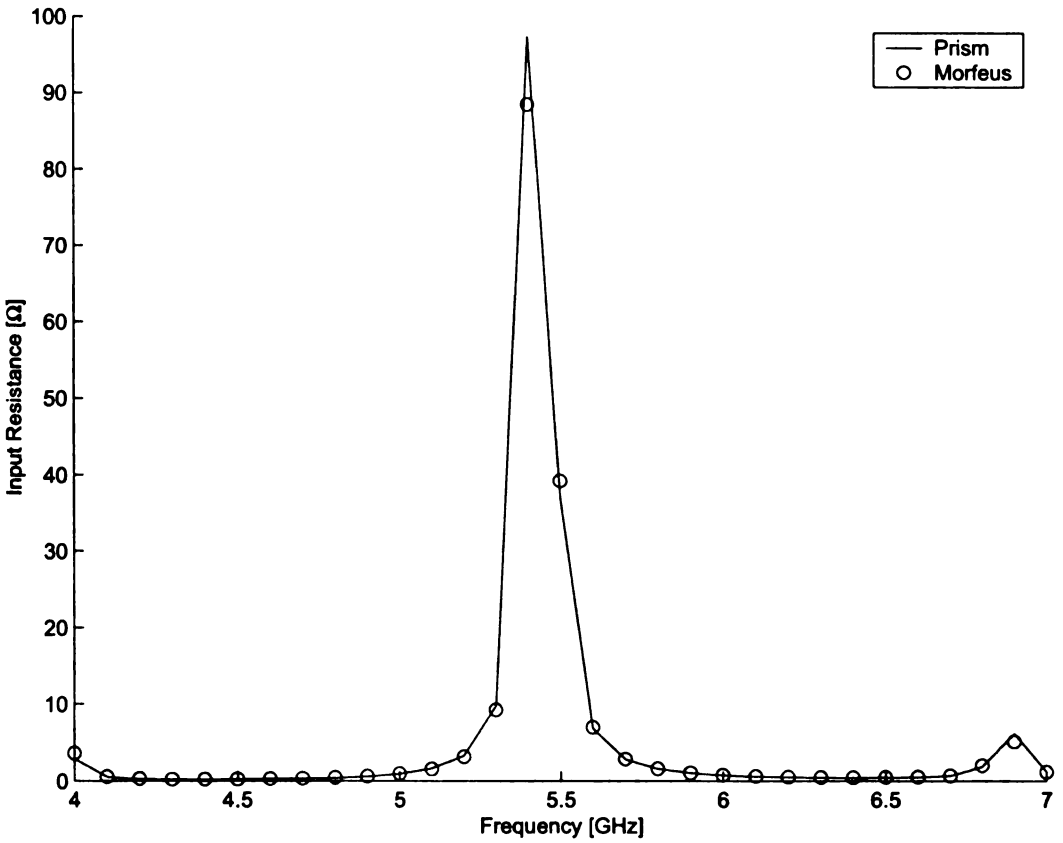

Figure 3.5. Input resistance of a 6cm x 4cm x 0.0762cm cavity with a 3cm x 2cm patch

CHAPTER 4

DISTORTED HEXAHEDRAL ELEMENTS

This chapter introduces basis functions for distorted hexahedral elements and extends the equations in chapter 2 for use with these basis functions. This chapter also discusses some of the difficulties that are associated with the use of distorted elements and how to overcome these difficulties.

4.1 FE-BI Formulation

Distorted hexahedral elements are important in finite element programming because they have the ability to model slightly irregular geometries with fewer edges than prisms. This leads to less computational and memory demand for the same geometry modeled with hexahedral elements as opposed to prism elements. Hexahedral elements, however, have the disadvantage of not being able to model geometries that have very sharp corners and severely irregular contours without using an excessive amount of elements, thereby defeating the purpose of using hexahedral elements. A formulation for curved hexahedral elements has previously been done by [28].

4.1.1 Basis Functions

The basis functions used for hexahedral elements are the so-called rooftop functions. Rooftop basis functions are most often used in brick elements [1], [2]. Brick elements are very easy to use but suffer from the fact that they can only model shoebox type geometries effectively. In order to model more irregular geometries hexahedral elements must be used. Due to the distortion of the elements, however, the integrals in (2.30) and (2.31) can be very difficult to compute numerically directly in cartesian coordinates. On the other hand, numerical integration over brick shaped volumes is very easy to implement. Through the use of a coordinate transformation, it is possible

to express the basis functions and the integral operations for the hexahedral elements in terms of a local coordinate system in which the hexahedral element becomes a cube. Figure 3 shows the transformation of a distorted hexahedral element in the (x, y, z) coordinate system to a unit cube in the (ξ, η, ζ) coordinate system, where the edges of the hexahedral and the brick are defined in Table 4.1

Edge	Node 1	Node 2
1	1	2
2	3	4
3	1	4
4	2	3
5	5	6
6	7	8
7	5	8
8	6	7
9	1	5
10	2	6
11	3	7
12	2	6

Table 4.1. Edge numbering for a distorted hexahedral element

Such a transformation can be described by the using the following relation at the eight nodes of the hexahedron where (a-h) are a set of unknown coefficients

$$\begin{aligned} x &= a + b\xi + c\eta + d\zeta + e\xi\eta + f\xi\zeta + g\eta\zeta + h\xi\eta\zeta \\ y &= a' + b'\xi + c'\eta + d'\zeta + e'\xi\eta + f'\xi\zeta + g'\eta\zeta + h'\xi\eta\zeta \\ z &= a'' + b''\xi + c''\eta + d''\zeta + e''\xi\eta + f''\xi\zeta + g''\eta\zeta + h''\xi\eta\zeta \end{aligned} \tag{4.1}$$

Using (4.1) the equations at each node of the element can be written

$$
\begin{aligned}
x_1 &= a - b - c + d + e - f - g + h \\
x_2 &= a + b - c + d - e + f - g - h \\
x_3 &= a + b + c + d + e + f + g + h \\
x_4 &= a - b + c + d - e - f + g - h \\
x_5 &= a - b - c - d + e + f + g - h \\
x_6 &= a + b - c - d - e - f + g + h \\
x_7 &= a + b + c - d + e - f - g - h \\
x_8 &= a - b + c - d - e + f - g + h
\end{aligned}
\tag{4.2}
$$

$$
\begin{aligned}
y_1 &= a' - b' - c' + d' + e' - f' - g' + h' \\
y_2 &= a' + b' - c' + d' - e' + f' - g' - h' \\
y_3 &= a' + b' + c' + d' + e' + f' + g' + h' \\
y_4 &= a' - b' + c' + d' - e' - f' + g' - h' \\
y_5 &= a' - b' - c' - d' + e' + f' + g' - h' \\
y_6 &= a' + b' - c' - d' - e' - f' + g' + h' \\
y_7 &= a' + b' + c' - d' + e' - f' - g' - h' \\
y_8 &= a' - b' + c' - d' - e' + f' - g' + h'
\end{aligned}
\tag{4.3}
$$

$$
\begin{aligned}
z_1 &= a'' - b'' - c'' + d'' + e'' - f'' - g'' + h'' \\
z_2 &= a'' + b'' - c'' + d'' - e'' + f'' - g'' - h'' \\
z_3 &= a'' + b'' + c'' + d'' + e'' + f'' + g'' + h''
\end{aligned}
$$

$$
\begin{aligned}
z_4 &= a'' - b'' + c'' + d'' - e'' - f'' + g'' - h'' \\
z_5 &= a'' - b'' - c'' - d'' + e'' + f'' + g'' - h'' \\
z_6 &= a'' + b'' - c'' - d'' - e'' - f'' + g'' + h'' \\
z_7 &= a'' + b'' + c'' - d'' + e'' - f'' - g'' - h'' \\
z_8 &= a'' - b'' + c'' - d'' - e'' + f'' - g'' + h''
\end{aligned} \tag{4.4}
$$

Solving for the unknown coefficients, the global coordinates are given by

$$
\begin{aligned}
x &= \sum_{i=1}^{8} N_i(\xi, \eta, \zeta) x_i \\
y &= \sum_{i=1}^{8} N_i(\xi, \eta, \zeta) y_i \\
z &= \sum_{i=1}^{8} N_i(\xi, \eta, \zeta) z_i
\end{aligned} \tag{4.5}
$$

where

$$
N_i(\xi, \eta, \zeta) = \frac{1}{8}(1 + \xi_i \xi)(1 + \eta_i \eta) + (1 + \zeta_i \zeta) \tag{4.6}
$$

and (ξ_i, η_i, ζ_i) represent the coordinate value of (ξ, η, ζ) along the i^{th} edge. Equation (4.6) represents the scalar nodal basis functions of the element. Following the same procedure as in [1], the vector edge-based basis functions can be written as

$$
\mathbf{N}_i = \frac{l_i s_i}{8}(1 + \eta_i \eta)(1 + \zeta_i \zeta) \nabla \xi \tag{4.7}
$$

for the edges parallel to the ξ axis

$$
\mathbf{N}_i = \frac{l_i s_i}{8}(1 + \xi_i \xi)(1 + \zeta_i \zeta) \nabla \eta \tag{4.8}
$$

for the edges parallel to the η axis, and finally

$$\mathbf{N}_i = \frac{l_i s_i}{8}(1 + \xi_i \xi)(1 + \eta_i \eta)\nabla\zeta \tag{4.9}$$

for the edges parallel to the ζ axis, where l_i denotes the length of the i^{th} edge, and s_i denotes the sign of the i^{th} edge. Noticing that

$$\begin{aligned}
\nabla\xi &= \hat{\xi}\frac{\partial\xi}{\partial\xi} + \hat{\eta}\frac{\partial\xi}{\partial\eta} + \hat{\zeta}\frac{\partial\xi}{\partial\zeta} = \hat{\xi} \\
\nabla\eta &= \hat{\xi}\frac{\partial\eta}{\partial\xi} + \hat{\eta}\frac{\partial\eta}{\partial\eta} + \hat{\zeta}\frac{\partial\eta}{\partial\zeta} = \hat{\eta} \\
\nabla\zeta &= \hat{\xi}\frac{\partial\zeta}{\partial\xi} + \hat{\eta}\frac{\partial\zeta}{\partial\eta} + \hat{\zeta}\frac{\partial\zeta}{\partial\zeta} = \hat{\zeta}
\end{aligned} \tag{4.10}$$

the basis functions can alternatively be written as

$$\begin{aligned}
\mathbf{N}_i &= \frac{l_i s_i}{8}(1 + \eta_i \eta)(1 + \zeta_i \zeta)\hat{\xi} \\
\mathbf{N}_i &= \frac{l_i s_i}{8}(1 + \xi_i \xi)(1 + \zeta_i \zeta)\hat{\eta} \\
\mathbf{N}_i &= \frac{l_i s_i}{8}(1 + \xi_i \xi)(1 + \eta_i \eta)\hat{\zeta}
\end{aligned} \tag{4.11}$$

The basis functions for hexahedral elements are now defined in the mapped coordinate system where numerical integration can be performed over a cube rather than a hexahedral volume.

4.1.2 Coordinate Transformation

Although the basis functions derived in the previous section are in terms of mapped coordinates, the integrands in (2.30) and (2.31) are still in terms of Cartesian coor-

dinates. Defining the Jacobian matrix as

$$\mathbf{J} = \begin{pmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta} \end{pmatrix} \tag{4.12}$$

the elemental volume of the integral can be written as [1]

$$dV = dxdydz = \det \mathbf{J} d\xi d\eta d\zeta \tag{4.13}$$

There are also some vector operations that will be important to evaluating the integrals in (2.30) and (2.31) in mapped coordinates. The dot product of a normal vector in Cartesian coordinates with a normal vector in mapped coordinates can be defined by using the definition of the gradient

$$(\hat{\mathbf{x}} \cdot \hat{\xi}) = (\hat{\mathbf{x}} \cdot \nabla \xi) = \hat{\mathbf{x}} \cdot \left[\hat{\mathbf{x}} \frac{\partial \xi}{\partial x} + \hat{\mathbf{y}} \frac{\partial \xi}{\partial y} + \hat{\mathbf{z}} \frac{\partial \xi}{\partial z} \right] = \frac{\partial \xi}{\partial x} \tag{4.14}$$

Using the same method the entire set of dot products can be written as

$$\begin{array}{lll} (\hat{\mathbf{x}} \cdot \hat{\xi}) = \dfrac{\partial \xi}{\partial x} & (\hat{\mathbf{x}} \cdot \hat{\eta}) = \dfrac{\partial \eta}{\partial x} & (\hat{\mathbf{x}} \cdot \hat{\zeta}) = \dfrac{\partial \zeta}{\partial x} \\ (\hat{\mathbf{y}} \cdot \hat{\xi}) = \dfrac{\partial \xi}{\partial y} & (\hat{\mathbf{y}} \cdot \hat{\eta}) = \dfrac{\partial \eta}{\partial y} & (\hat{\mathbf{y}} \cdot \hat{\zeta}) = \dfrac{\partial \zeta}{\partial y} \\ (\hat{\mathbf{z}} \cdot \hat{\xi}) = \dfrac{\partial \xi}{\partial z} & (\hat{\mathbf{z}} \cdot \hat{\eta}) = \dfrac{\partial \eta}{\partial z} & (\hat{\mathbf{z}} \cdot \hat{\zeta}) = \dfrac{\partial \zeta}{\partial z} \end{array} \tag{4.15}$$

From [29] the following vector operations can be defined

$$\hat{\mathbf{z}} \cdot (\nabla \times \mathbf{N}) = \frac{1}{\det \mathbf{J}} \left[\frac{\partial N^{\eta}}{\partial \xi} - \frac{\partial N^{\xi}}{\partial \eta} \right] \tag{4.16}$$

$$(\nabla \times \mathbf{N}) = \frac{1}{\det \mathbf{J}} \left[\hat{\xi} \left(\frac{\partial N^{\zeta}}{\partial \eta} - \frac{\partial N^{\eta}}{\partial \zeta} \right) + \hat{\eta} \left(\frac{\partial N^{\xi}}{\partial \zeta} - \frac{\partial N^{\zeta}}{\partial \xi} \right) + \hat{\zeta} \left(\frac{\partial N^{\eta}}{\partial \xi} - \frac{\partial N^{\xi}}{\partial \eta} \right) \right] \tag{4.17}$$

4.1.3 FE Matrix Entries

The FE matrix equations for hexahedral elements are different from the FE matrix equations derived for prism elements in section 3.1.2 due to the fact that they cannot be evaluated in closed form, they must be evaluated numerically. This fact makes evaluating these integrals much slower since each numerical integration point must be mapped into a different coordinate system. To map the integrands to the correct coordinate system, the development from [29] for mapping vector basis functions to curvilinear cells can be used. Although this method was shown to map basis functions to curved cells it works just as well here, due to the fact that it simply presents mapping of a function from one coordinate system to another. Defining the vector

$$\mathbf{V}_i^T = \left[\hat{\xi}\left(\frac{\partial N^\varsigma}{\partial \eta} - \frac{\partial N^\eta}{\partial \varsigma}\right) + \hat{\eta}\left(\frac{\partial N^\xi}{\partial \varsigma} - \frac{\partial N^\varsigma}{\partial \xi}\right) + \hat{\varsigma}\left(\frac{\partial N^\eta}{\partial \xi} - \frac{\partial N^\xi}{\partial \eta}\right)\right] \tag{4.18}$$

the two integrals in (2.30) are written as

$$I_{ij}^{FE(1)} = \int_{-1}^{1} \frac{1}{\det \mathbf{J}} \mathbf{V}_i^T \mathbf{J}\mathbf{J}^T \mathbf{V}_j d\xi d\eta d\varsigma \tag{4.19}$$

$$I_{ij}^{FE(2)} = \int_{-1}^{1} \left[\hat{\xi}N_i^\xi + \hat{\eta}N_i^\eta + \hat{\varsigma}N_i^\varsigma\right] \mathbf{J}^{-T}\mathbf{J}^{-1} \left[\hat{\xi}N_j^\xi + \hat{\eta}N_j^\eta + \hat{\varsigma}N_j^\varsigma\right]^T \det \mathbf{J} d\xi d\eta d\varsigma \tag{4.20}$$

By using the mapping shown in section 4.1.2 the entries of the Jacobian matrix can be found directly. The inverse Jacobian and the determinant can then be computed numerically from the Jacobian. Since the inverse and determinant must be computed numerically, the integrals in (4.19) and (4.20) can take a significant amount of time to compute. Looking at equation (2.30), however, it can be seen that there is no frequency dependency in the integrands. This means that the integral quantities can be computed once and stored independent of frequency. These quantities can then be scaled by the correct frequency constants.

4.1.4 BI Matrix Entries

The BI matrix equations for the hexahedral elements follow closely from the BI matrix equations derived for the prism elements in section 3.1.3. The basis functions must once again be compressed into two dimensions to enforce tangential electric field continuity at the surface, making the hexahedrals into quadrilaterals. Following a similar method used in section 4.1.1 for two dimensions, also shown in [1], the quadrilateral basis functions can be written as

$$\mathbf{N} = \frac{l_i s_i}{4}(1 + \eta_i \eta)\hat{\xi} \tag{4.21}$$

for the ξ directed edges and

$$\mathbf{N} = \frac{l_i s_i}{4}(1 + \xi_i \xi)\hat{\eta} \tag{4.22}$$

for the η directed edges. As in the three dimensional case the two dimensional elemental areas may be written in terms of a Jacobian matrix

$$dS' dS = dx' dy' dx dy = \det \mathbf{J_i} \det \mathbf{J}_j d\xi' d\eta' d\xi d\eta \tag{4.23}$$

where the Jacobian matrix is now defined in two dimensions

$$\mathbf{J} = \begin{pmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{pmatrix} \tag{4.24}$$

Expanding the divergence terms in (2.31) in terms of a vector identity

$$\nabla \cdot (\hat{\mathbf{z}} \times \mathbf{N}) = -\hat{\mathbf{z}} \cdot (\nabla \times \mathbf{N}) \tag{4.25}$$

Using (4.16) and (4.23), the BI equations become

$$I_{ij}^{BI(1)} = -\frac{k_0^2}{2\pi}\int_{-1}^{1}[\mathbf{N}_i \cdot (\hat{\mathbf{x}}\hat{\mathbf{x}} + \hat{\mathbf{x}}\hat{\mathbf{x}}) \cdot \mathbf{N}_j]\frac{e^{-jk_0R}}{R}\det \mathbf{J}_i \det \mathbf{J}_j d\xi' d\eta' d\xi d\eta \tag{4.26}$$

$$I_{ij}^{BI(1)} = \frac{1}{2\pi}\int_{-1}^{1}\left(\frac{\partial N_i^{\eta}}{\partial \xi} - \frac{\partial N_i^{\xi}}{\partial \eta}\right)\left(\frac{\partial N_j^{\eta}}{\partial \xi'} - \frac{\partial N_j^{\xi}}{\partial \eta'}\right)\frac{e^{-jk_0R}}{R}d\xi' d\eta' d\xi d\eta \tag{4.27}$$

The derivatives needed in (4.27) are

$$\frac{\partial N_i^{\xi}}{\partial \eta} = \frac{l_i s_i}{4}\eta_i \tag{4.28}$$

$$\frac{\partial N_i^{\eta}}{\partial \xi} = \frac{l_i s_i}{4}\xi_i \tag{4.29}$$

$$\frac{\partial N_j^{\xi}}{\partial \eta} = \frac{l_i s_i}{4}\eta_j \tag{4.30}$$

$$\frac{\partial N_j^{\eta}}{\partial \xi} = \frac{l_i s_i}{4}\xi_j \tag{4.31}$$

Using the derivatives in (4.28)-(4.31) along with the dot products shown in (4.10) the integrals in (4.26) can be written

$$I_{ij(\xi\xi)}^{BI(1)} = -\frac{k_0^2 l_i l_j s_i s_j}{32\pi}\int_{-1}^{1}(1+\eta_i\eta)(1+\eta_j\eta')\left[\frac{\partial \xi}{\partial x}\frac{\partial \xi'}{\partial x} + \frac{\partial \xi}{\partial y}\frac{\partial \xi'}{\partial y}\right] \frac{e^{-jk_0R}}{R}(\det \mathbf{J}_i)(\det \mathbf{J}_i)d\xi' d\eta' d\xi d\eta \tag{4.32}$$

$$I_{ij(\xi\eta)}^{BI(1)} = -\frac{k_0^2 l_i l_j s_i s_j}{32\pi}\int_{-1}^{1}(1+\eta_i\eta)(1+\xi_j\xi')\left[\frac{\partial \xi}{\partial x}\frac{\partial \eta'}{\partial x} + \frac{\partial \xi}{\partial y}\frac{\partial \eta'}{\partial y}\right] \frac{e^{-jk_0R}}{R}(\det \mathbf{J}_i)(\det \mathbf{J}_i)d\xi' d\eta' d\xi d\eta \tag{4.33}$$

$$I_{ij(\eta\xi)}^{BI(1)} = -\frac{k_0^2 l_i l_j s_i s_j}{32\pi}\int_{-1}^{1}(1+\xi_i\xi)(1+\eta_j\eta')\left[\frac{\partial \eta}{\partial x}\frac{\partial \xi'}{\partial x} + \frac{\partial \eta}{\partial y}\frac{\partial \xi'}{\partial y}\right]$$

$$\frac{e^{-jk_0R}}{R}(\det \mathbf{J}_i)(\det \mathbf{J}_i)d\xi' d\eta' d\xi d\eta \tag{4.34}$$

$$I_{ij(\eta\eta)}^{BI(1)} = -\frac{k_0^2 l_i l_j s_i s_j}{32\pi}\int_{-1}^{1}(1+\xi_i\xi)(1+\xi_j\xi')\left[\frac{\partial\eta}{\partial x}\frac{\partial\eta'}{\partial x}+\frac{\partial\eta}{\partial y}\frac{\partial\eta'}{\partial y}\right]$$
$$\frac{e^{-jk_0R}}{R}(\det \mathbf{J}_i)(\det \mathbf{J}_i)d\xi' d\eta' d\xi d\eta \tag{4.35}$$

and the integrals in (4.27) can be written

$$I_{ij(\xi\xi)}^{BI(2)} = \frac{l_i l_j s_i s_j \eta_i \eta_j}{32\pi}\int_{-1}^{1}\frac{e^{-jk_0R}}{R}d\xi' d\eta' d\xi d\eta \tag{4.36}$$

$$I_{ij(\xi\eta)}^{BI(2)} = \frac{l_i l_j s_i s_j \eta_i \xi_j}{32\pi}\int_{-1}^{1}\frac{e^{-jk_0R}}{R}d\xi' d\eta' d\xi d\eta \tag{4.37}$$

$$I_{ij(\eta\xi)}^{BI(2)} = \frac{l_i l_j s_i s_j \xi_i \eta_j}{32\pi}\int_{-1}^{1}\frac{e^{-jk_0R}}{R}d\xi' d\eta' d\xi d\eta \tag{4.38}$$

$$I_{ij(\eta\eta)}^{BI(2)} = \frac{l_i l_j s_i s_j \xi_i \xi_j}{32\pi}\int_{-1}^{1}\frac{e^{-jk_0R}}{R}d\xi' d\eta' d\xi d\eta \tag{4.39}$$

4.1.5 Plane Wave Excitation

Inserting in the definition of the two dimensional basis functions into the plane wave excitation equation from chapter 2, (2.63) becomes

$$\tilde{f}_i^{ext(\xi)} = \frac{jk_0 d_i s_i}{2}\int_{-1}^{1}(1+\eta_i\eta)\left[\frac{\partial\xi}{\partial x}C_1 - \frac{\partial\xi}{\partial y}C_2\right]e^{jk_0C_0}\det \mathbf{J}_i d\xi d\eta \tag{4.40}$$

$$\tilde{f}_i^{ext(\eta)} = \frac{jk_0 d_i s_i}{2}\int_{-1}^{1}(1+\xi_i\xi)\left[\frac{\partial\eta}{\partial x}C_1 - \frac{\partial\eta}{\partial y}C_2\right]e^{jk_0C_0}\det \mathbf{J}_i d\xi d\eta \tag{4.41}$$

where

$$\begin{aligned} C_0 &= \sin\theta\left[x\cos\phi^i + y\sin\phi^i\right] \\ C_1 &= \sin\alpha\cos\theta^i\sin\phi^i - \cos\alpha\cos\phi^i \end{aligned} \tag{4.42}$$

$$C_2 = \sin\alpha\cos\theta^i\cos\phi^i + \cos\alpha\sin\phi^i$$

Again, as in chapter 3, these equations must be computed numerically.

4.1.6 Radiation Pattern

Using the expansion of the electric field in terms of the hexahedral basis functions in (2.69)

$$\mathbf{I} = \sum_j \left\{ E_j \int_S \left(\hat{\theta}\hat{\theta} + \hat{\phi}\hat{\phi}\right) \cdot \left[\hat{\mathbf{z}} \times \mathbf{N}(\xi', \eta')\right] e^{jk0(\hat{\mathbf{r}}\cdot\mathbf{r}')} dS \right\} \tag{4.43}$$

Expanding the integrand as

$$\hat{\mathbf{z}} \times \left[\hat{\xi} N_j^{\xi} + \hat{\eta} N_j^{\eta}\right] = N_j^{\xi}\left(\hat{\mathbf{y}}\frac{\partial\xi}{\partial x} - \hat{\mathbf{x}}\frac{\partial\xi}{\partial y}\right) + N_j^{\eta}\left(\hat{\mathbf{y}}\frac{\partial\eta}{\partial x} - \hat{\mathbf{x}}\frac{\partial\eta}{\partial y}\right) \tag{4.44}$$

$$\mathbf{I} = \sum_j \left\{ E_j \int_S \left(\hat{\theta}\hat{\theta} + \hat{\phi}\hat{\phi}\right) \cdot \left[N_j^{\xi}\left(\hat{\mathbf{y}}\frac{\partial\xi}{\partial x} - \hat{\mathbf{x}}\frac{\partial\xi}{\partial y}\right) + N_j^{\eta}\left(\hat{\mathbf{y}}\frac{\partial\eta}{\partial x} - \hat{\mathbf{x}}\frac{\partial\eta}{\partial y}\right)\right] e^{jk0(\hat{\mathbf{r}}\cdot\mathbf{r}')} dS \right\} \tag{4.45}$$

Noticing that

$$\begin{aligned}
(\hat{\theta}\cdot\hat{\mathbf{x}}) &= \cos(\theta)\cos(\phi) \\
(\hat{\phi}\cdot\hat{\mathbf{x}}) &= -\sin(\phi) \\
(\hat{\theta}\cdot\hat{\mathbf{y}}) &= \cos(\theta)\sin(\phi) \\
(\hat{\phi}\cdot\hat{\mathbf{y}}) &= \cos(\phi)
\end{aligned}$$

and utilizing in the two dimensional basis functions shown in (4.21) and (4.22), the radiation pattern can be written as

$$I_{\theta}^{\xi} = \sum_j \frac{E_j l_j s_j}{4}\left[\int_{-1}^{1} \cos\theta(1+\eta_j\eta)\left(\sin\phi\frac{\partial\xi}{\partial x} - \cos\phi\frac{\partial\xi}{\partial y}\right) e^{jk0(\hat{\mathbf{r}}\cdot\mathbf{r}')} d\xi' d\eta'\right] \tag{4.46}$$

$$I_{\theta}^{\eta} = \sum_j \frac{E_j l_j s_j}{4}\left[\int_{-1}^{1} \cos\theta(1+\xi_j\xi)\left(\sin\phi\frac{\partial\eta}{\partial x} - \cos\phi\frac{\partial\eta}{\partial y}\right) e^{jk0(\hat{\mathbf{r}}\cdot\mathbf{r}')} d\xi' d\eta'\right] \tag{4.47}$$

$$I_{\phi}^{\xi} = \sum_{j} \frac{E_j l_j s_j}{4} \left[\int_{-1}^{1} (1 + \eta_j \eta) \left(\cos\phi \frac{\partial \xi}{\partial x} + \sin\phi \frac{\partial \xi}{\partial y} \right) e^{jk0(\hat{\mathbf{r}} \cdot \mathbf{r}')} d\xi' d\eta' \right] \tag{4.48}$$

$$I_{\phi}^{\eta} = \sum_{j} \frac{E_j l_j s_j}{4} \left[\int_{-1}^{1} (1 + \xi_j \xi) \left(\cos\phi \frac{\partial \eta}{\partial x} + \sin\phi \frac{\partial \eta}{\partial y} \right) e^{jk0(\hat{\mathbf{r}} \cdot \mathbf{r}')} d\xi' d\eta' \right] \tag{4.49}$$

where

$$e^{jk_0(\hat{\mathbf{r}} \cdot \mathbf{r}')} = e^{jk_0 \sin\theta (x' \cos\phi + y' \sin\phi)}$$

4.2 Examples

To validate the hexahedral element formulation the same geometries from chapter 3 were modeled using only hexahedral elements. The results were compared to the case where only prisms were used.

4.2.1 Rectangular Slot Antenna

As in chapter 3 the radar cross section of a 6cm x 5cm x 2cm cavity with a centered 3cm x 2cm slot aperture was computed. Figure 4.2 shows the $\theta\theta$-pol results while Figure 4.3 shows the $\phi\phi$-pol results. It is seen that the RCS calculations are in very good agreement with the results computed using only prisms.

4.2.2 Rectangular Patch Antenna

The rectangular patch antenna from chapter 3 was also modeled using the hexahedral formulation and the results were compared to the results from the prism formulation. Figure 4.4 shows the radiation pattern from $\theta = 90°$ to $-90°$ at $\phi = 0°$. Figure 4.5 shows the input resistance from 4-7 GHz.

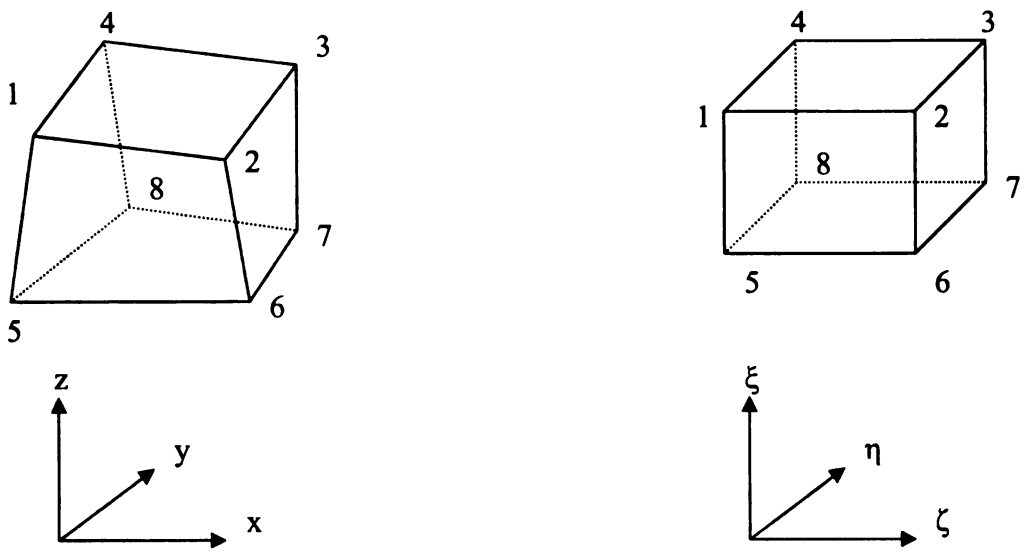

Figure 4.1. Hexahedral in (x,y,z) coordinates mapped into a cube in (ξ, η, ζ) coordinates. The numbers shown correspond to the local node numbering scheme.

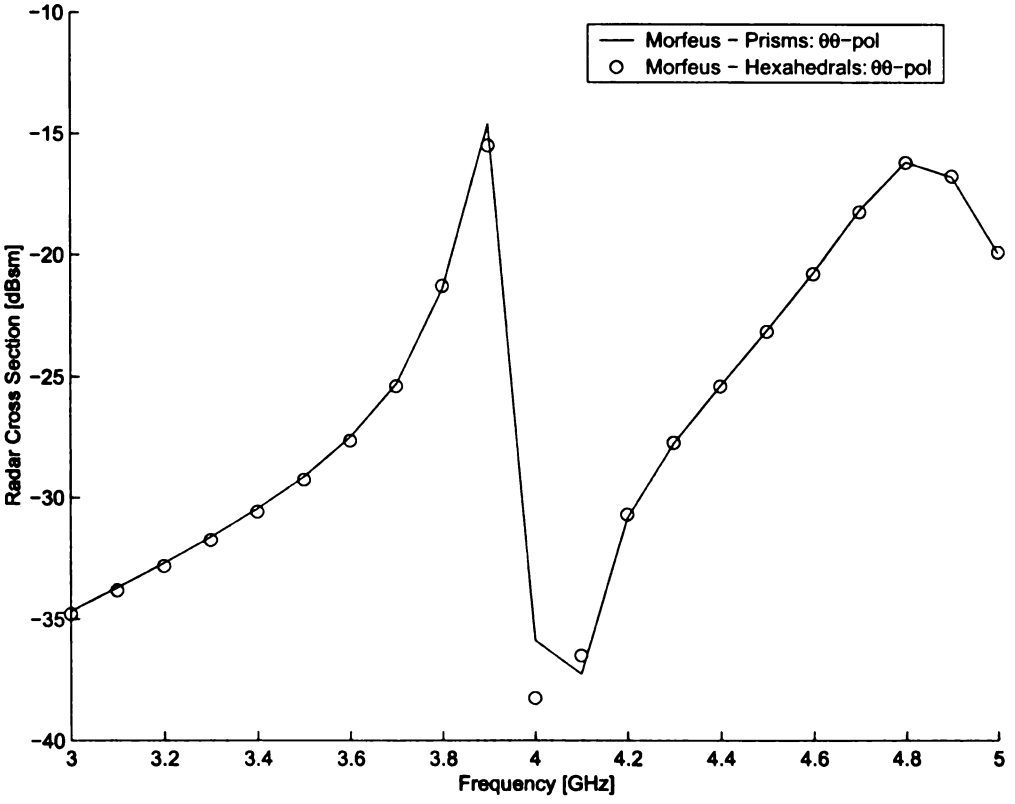

Figure 4.2. Radar cross section of a 6cm x 5cm x 2cm cavity with a 3cm x 2cm slot aperture, $\theta\theta$-pol.

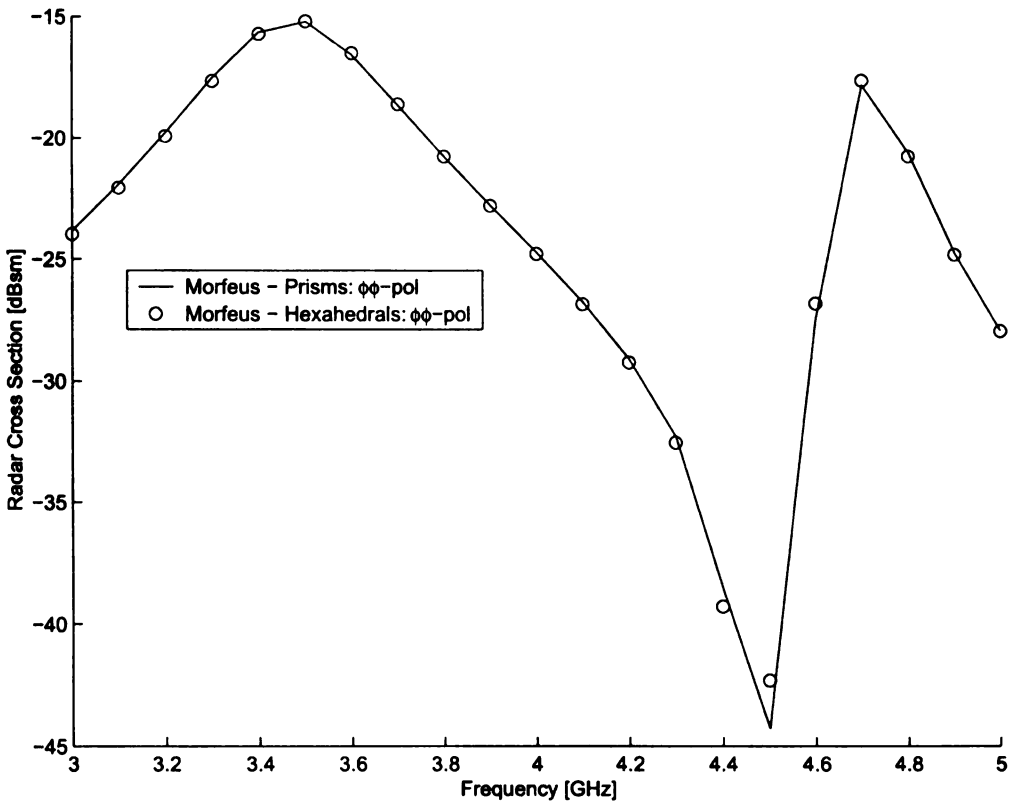

Figure 4.3. Radar cross section of a 6cm x 5cm x 2cm cavity with a 3cm x 2cm slot aperture, $\phi\phi$-pol.

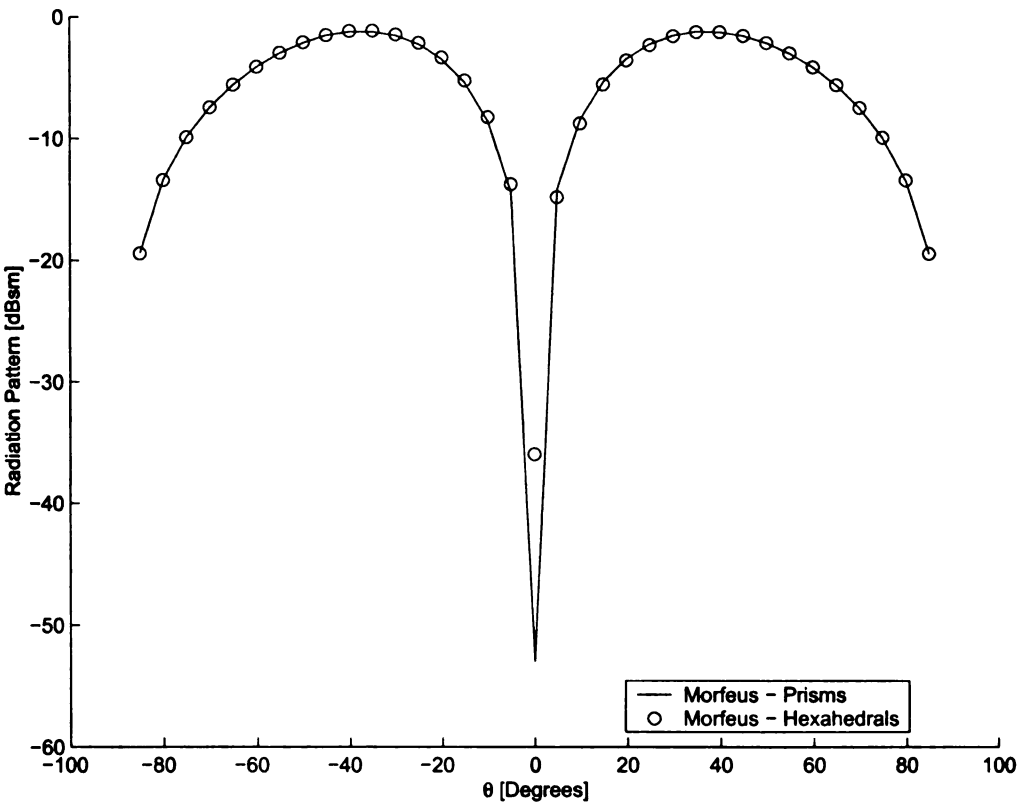

Figure 4.4. Radiation patten of a 6cm x 4cm x 0.0762cm cavity with a 3cm x 2cm patch.

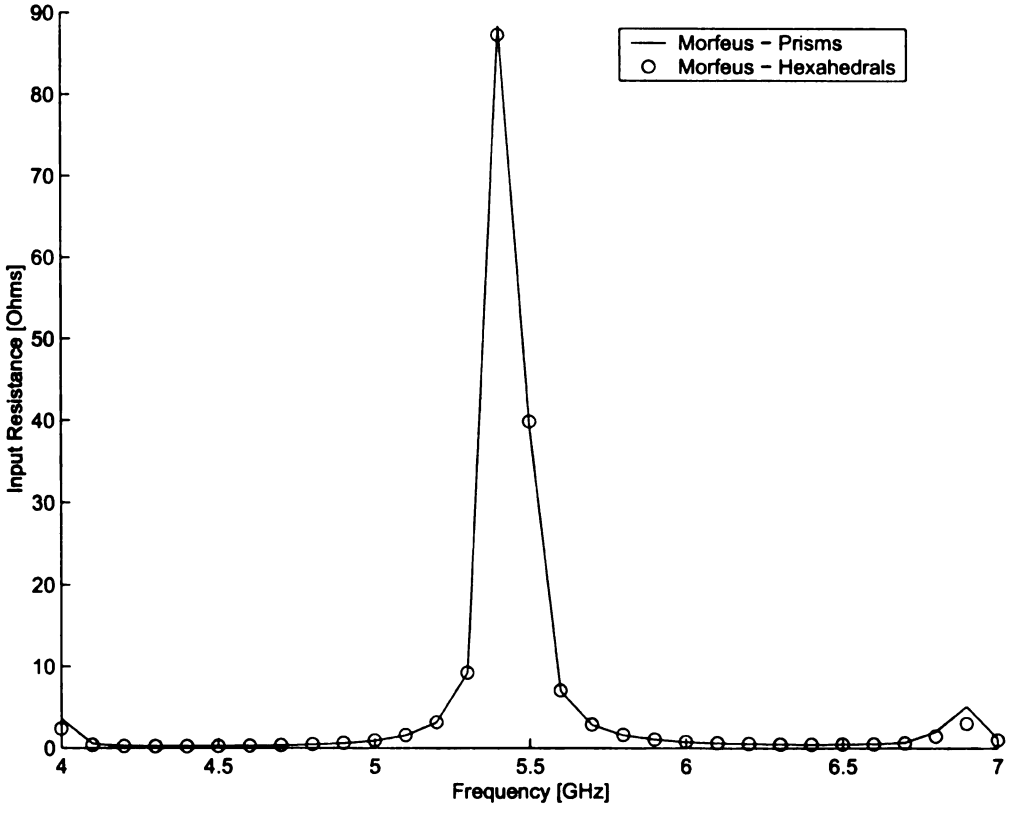

Figure 4.5. Input resistance of a 6cm x 4cm x 0.0762cm cavity with a 3cm x 2cm patch.

CHAPTER 5

MIXED ELEMENTS

This chapter presents the mixed element formulation for the FE-BI method. This formulation combines prism and hexahedral elements into the same geometry and it will be shown that the computational and memory efficiency of the corresponding solution is improved while accuracy is maintained.

5.1 FE-BI Formulation

Up to this point in the formulation, it has been implicitly assumed that all elements in the geometry are the same shape and are modeled by the same basis functions. This works well in practice and many implementations of this technique have been successful. In chapter 3, prism elements were used to model the geometry. Prism elements are quite adept at modeling irregular geometries due to the fact that in two dimensions they compress into triangles, which are very flexible and able to accurately model many two dimensional surfaces. Triangles, however, have the distinct disadvantage of producing a large number of unknowns. This does not heavily affect the FE portion of the system but can have a drastic affect on the BI portion. Since a large amount of the time solving the system is spent computing and solving the BI portion of the system, the ability to reduce the number of unknowns on the surface is of great interest. Hexahedral elements were discussed in chapter 4 and are seen to model fairly regular geometries as effectively as prisms. Hexahedral elements also use many fewer unknowns, almost a third fewer [1], in fact. However, hexahedral elements cannot model severely irregular geometries as effectively as prisms. This is attributed to the fact that the coordinate transformation process described in chapter 4 cannot accommodate, accurately, severe distortion. Specifically, the Jacobian matrix becomes unsuitable and its inverse becomes susceptible to error. To get the

best of both worlds, the flexibility of prism elements and the unknown saving hexahedral elements, the two types of elements can be combined. This allows the use of prism elements where their flexibility is needed, such as at sharp contours and areas of rapid varying fields, while still allowing the use of hexahedral elements where fewer unknowns are needed, such as areas of slowly varying fields. Within the development itself, only a few additional equations need to be developed in order to make this element mixing possible. To be more specific, only the boundary integral terms where a prism edge is interacting with a hexahedral edge need to be derived. All other equations, such as the FE matrix equations and antenna pattern equations, are still valid in their current form due to the local nature of their coupling.

5.1.1 BI Matrix Entries

The BI matrix equations for mixed elements can be derived using the same methods as those used in the computation of the BI matrix equations for both prism and hexahedral elements. The only major difference is that now the basis functions in (2.31) will be represented as two distinct functions. The integrals shown in (2.49) and (2.50) are now written as

$$I_{ij}^{BI(1)} = -\frac{k_0^2}{2\pi}\int_S\int_{-1}^{1}\left[\mathbf{W}_i\cdot(\hat{\mathbf{x}}\hat{\mathbf{x}}+\hat{\mathbf{x}}\hat{\mathbf{x}})\cdot\mathbf{N}_j\right]\frac{e^{-jk_0R}}{R}\det\mathbf{J}_j d\xi' d\eta' dS \tag{5.1}$$

$$I_{ij}^{BI(2)} = \frac{1}{2\pi}\int_S\int_{-1}^{1}\nabla\cdot(\hat{\mathbf{z}}\times\mathbf{W}_i)\,\nabla'\cdot(\hat{\mathbf{z}}\times\mathbf{N}_j)\frac{e^{-jk_0R}}{R}\det\mathbf{J}_j d\xi' d\eta' dS \tag{5.2}$$

when prisms are used as test elements and hexahedrals are used as source elements, or conversely,

$$I_{ij}^{BI(1)} = -\frac{k_0^2}{2\pi}\int_{-1}^{1}\int_{S'}\left[\mathbf{N}_i\cdot(\hat{\mathbf{x}}\hat{\mathbf{x}}+\hat{\mathbf{x}}\hat{\mathbf{x}})\cdot\mathbf{W}_j\right]\frac{e^{-jk_0R}}{R}\det\mathbf{J}_i d\xi d\eta dS' \tag{5.3}$$

$$I_{ij}^{BI(2)} = \frac{1}{2\pi}\int_{-1}^{1}\int_{S'}\nabla\cdot(\hat{\mathbf{z}}\times\mathbf{N}_i)\,\nabla'\cdot(\hat{\mathbf{z}}\times\mathbf{W}_j)\frac{e^{-jk_0R}}{R}\det\mathbf{J}_i d\xi d\eta dS' \tag{5.4}$$

when hexahedrals are used as test elements and prisms are used as source elements. Recalling that

$$\hat{\mathbf{z}} \cdot (\nabla \times \mathbf{N}) = \frac{1}{\det \mathbf{J}} \left[\frac{\partial N^{\eta}}{\partial \xi} - \frac{\partial N^{\xi}}{\partial \eta} \right] \tag{5.5}$$

and

$$\nabla \cdot (\hat{\mathbf{z}} \times \mathbf{W}_i) = -\frac{l_i s_i}{S_i^e} \tag{5.6}$$

Using the dot products shown in (4.15) and the basis functions defined in chapter 3 for triangle elements and chapter 4 for quadrilateral elements respectively, the BI matrix equations can be written as

$$I_{ij(\xi p)}^{BI(1)} = -\frac{k_0^2 l_i l_j s_i s_j}{16\pi S_j^e} \int_{-1}^{1} \int_{S'} (1+\eta_i \eta) \left[(x' - x_j) \frac{\partial \xi}{\partial y} - (y' - y_j) \frac{\partial \xi}{\partial x} - \right] \frac{e^{-jk_0 R}}{R} \det \mathbf{J}_i dS' d\xi d\eta \tag{5.7}$$

$$I_{ij(\eta p)}^{BI(1)} = -\frac{k_0^2 l_i l_j s_i s_j}{16\pi S_j^e} \int_{-1}^{1} \int_{S'} (1+\xi_i \xi) \left[(x' - x_j) \frac{\partial \eta}{\partial y} - (y' - y_j) \frac{\partial \eta}{\partial x} - \right] \frac{e^{-jk_0 R}}{R} \det \mathbf{J}_i dS' d\xi d\eta \tag{5.8}$$

$$I_{ij(p\xi)}^{BI(1)} = -\frac{k_0^2 l_i l_j s_i s_j}{16\pi S_i^e} \int_{S} \int_{-1}^{1} (1+\eta_j \eta) \left[(x - x_i) \frac{\partial \xi'}{\partial y} - (y - y_i) \frac{\partial \xi'}{\partial x} - \right] \frac{e^{-jk_0 R}}{R} \det \mathbf{J}_j d\xi' d\eta' dS \tag{5.9}$$

$$I_{ij(p\eta)}^{BI(1)} = -\frac{k_0^2 l_i l_j s_i s_j}{16\pi S_i^e} \int_{S} \int_{-1}^{1} (1+\xi_j \xi) \left[(x - x_i) \frac{\partial \eta'}{\partial y} - (y - y_i) \frac{\partial \eta'}{\partial x} - \right] \frac{e^{-jk_0 R}}{R} \det \mathbf{J}_j d\xi' d\eta' dS \tag{5.10}$$

$$I_{ij(\xi p)}^{BI(2)} = -\frac{l_i l_j s_i s_j \eta_i}{8\pi S_j^e} \int_{-1}^{1} \int_{S'} \frac{e^{-jk_0 R}}{R} dS' d\xi d\eta \tag{5.11}$$

$$I_{ij(\eta p)}^{BI(2)} = \frac{l_i l_j s_i s_j \xi_i}{8\pi S_j^e} \int_{-1}^{1} \int_{S'} \frac{e^{-jk_0 R}}{R} dS' d\xi d\eta \tag{5.12}$$

$$I_{ij(p\xi)}^{BI(2)} = -\frac{l_i l_j s_i s_j \eta_j}{8\pi S_i^e} \int_{S} \int_{-1}^{1} \frac{e^{-jk_0 R}}{R} d\xi' d\eta' dS \tag{5.13}$$

$$I_{ij(p\eta)}^{BI(2)} = \frac{l_i l_j s_i s_j \xi_j}{8\pi S_i^e} \int_{S} \int_{-1}^{1} \frac{e^{-jk_0 R}}{R} d\xi' d\eta' dS \tag{5.14}$$

The equations in (5.7)-(5.14) constitute the only additional equations needed to develop the mixed element formulation.

5.2 Examples

To validate the mixed element formulation, the same geometries from chapters 3 and 4 were modeled using a mixture of hexahedrals and prism elements. The results were compared to the case where only prisms were used.

5.2.1 Rectangular Slot Antenna

As in chapter 3 the radar cross section of a 6cm x 5cm x 2cm cavity with a centered 3cm x 2cm slot aperture was computed. Figure 5.1 shows the $\theta\theta$-pol results while Figure 5.2 shows the $\phi\phi$-pol results. It is seen that the RCS calculations are in very good agreement with the results computed using only prisms.

5.2.2 Rectangular Patch Antenna

The rectangular patch antenna from chapter 3 was also modeled using the mixed element formulation and the results were compared to the results from the prism formulation. Figure 5.3 shows the radiation pattern from $\theta = 90°$ to $-90°$ at $\phi = 0°$. Figure 5.4 shows the input resistance from 4-7 GHz. The slight discrepancy in the impedance plot can be attributed to the fact that using the mixed element formulation it is not possible to place the probe feed at the exact same location in both geometries. Near resonance the input impedance is very sensitive and moving the probe feed a slight amount can cause a significant difference in impedance values.

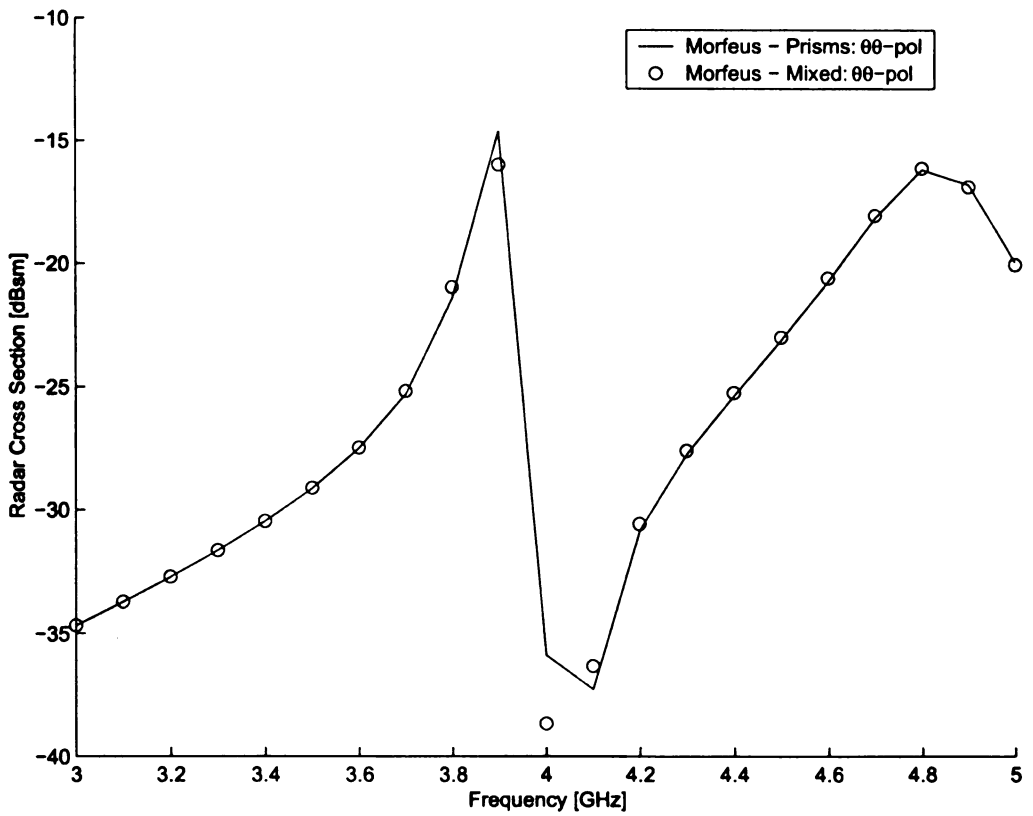

Figure 5.1. Radar cross section of a 6cm x 5cm x 2cm cavity with a 3cm x 2cm slot aperture, $\theta\theta$-pol

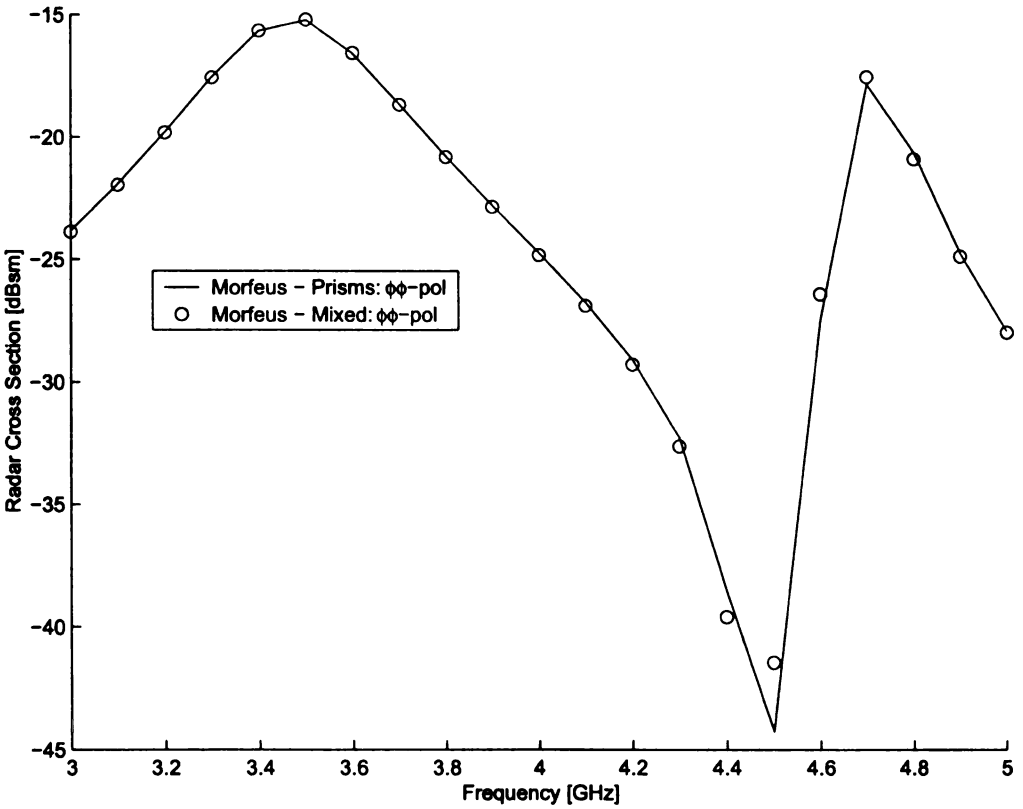

Figure 5.2. Radar cross section of a 6cm x 5cm x 2cm cavity with a 3cm x 2cm slot aperture, $\phi\phi$-pol

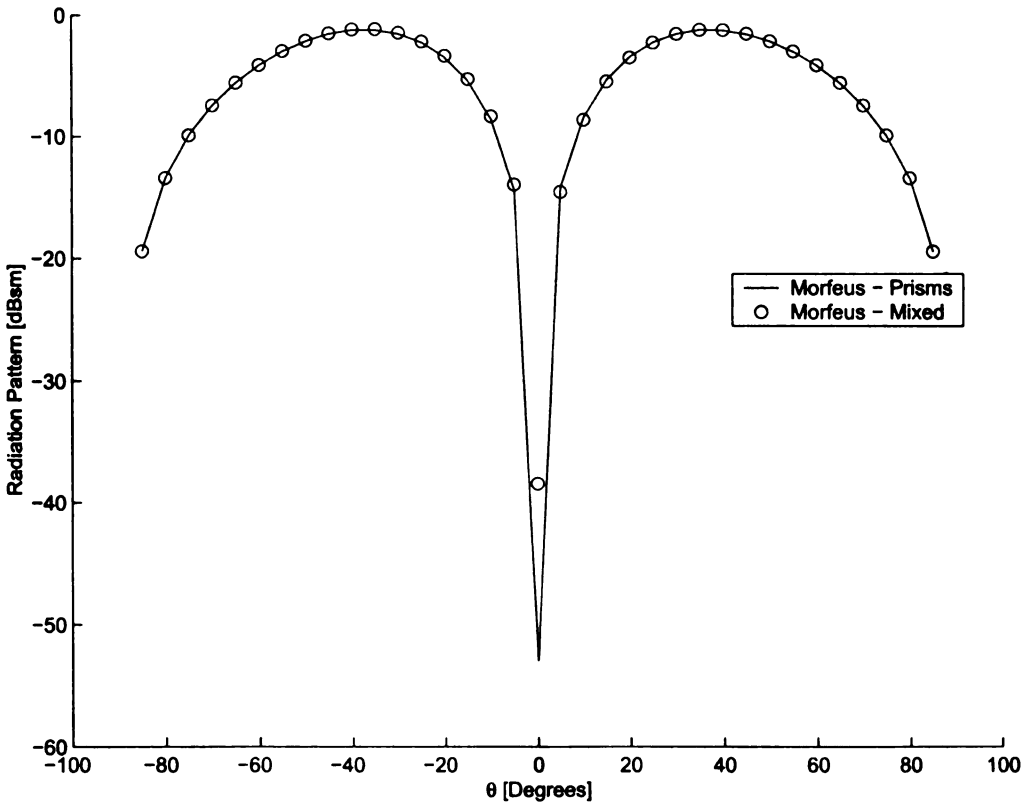

Figure 5.3. Radiation patten of a 6cm x 4cm x 0.0762cm cavity with a 3cm x 2cm patch

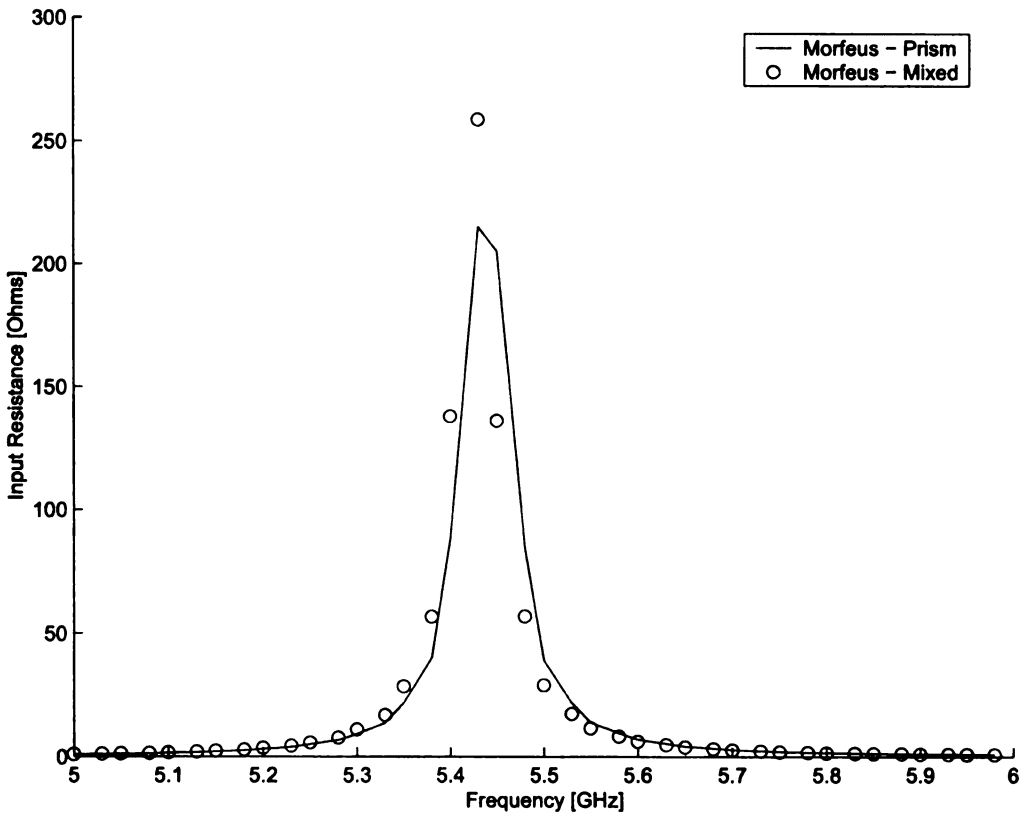

Figure 5.4. Input resistance of a 6cm x 4cm x 0.0762cm cavity with a 3cm x 2cm patch

CHAPTER 6

COMPLEX APERTURES

In this chapter a couple of complex apertures will be studied to demonstrate the validity of the mixed element formulation developed in chapter 5. Comparative results will be shown as well as a comparison of the time and memory it takes to compute these results in order to show that using the mixed element formulation can handle complex geometries in a more efficient manner than using prisms alone.

6.1 Four Arm Spiral Antenna

The spiral antenna has been around since 1954 when Edwin Turner at Wright Patterson Air Force base decided to wind the arms of a dipole antenna [30]. Spiral antennas demonstrate nearly constant impedance and beamwidth independent of frequency making them attractive for direction wide bandwidth finding applications among others.

6.1.1 Introduction

Historically, the four arm spiral antenna has been chosen for direction finding (DF) applications; rather than the two, three, or n-arm spiral antennas since it offers sufficient modes for phase-based DF while the required mode formers can be constructed using 90° and 180° phase shifters. Hence, this dissertation will consider only four arm spiral antennas. Theoretically, the number of useful modes of a spiral antenna is equal to its number of arms but in practice only one fewer mode can be used. Mode zero suffers from feed line radiation caused by exciting the feeds of the spiral arms with in phase currents. This feed line radiation limits the usefulness of the spiral antenna and is not easily suppressed. Therefore, a four arm spiral antenna has three useful modes. These modes are often termed mode 1, mode 2, and mode 3. For

transmission, the modes are realized by varying the phase of the feed currents in each of the spiral arms. Also, in practice mode 3 is the conjugate of mode 1 so only modes 1 and 2 are used. For a four arm spiral antenna, mode 1 is realized by exciting the feeds with a 90 degree phase progression between each feed current. Mode 2 is realized using a 180 degree phase progression and mode 3 is realized using a 270 degree phase progression. For reception, a mode former acts as a mode filter extracting the modes for an incident signal. Since computationally, transmission is more efficient to compute than reception (due to one right hand side in the system) and reciprocity applies, only transmission will be considered. For this dissertation only mode 1 and mode 2 will be considered since this is sufficient to demonstrate the validity of the mixed element formulation.

The frequencies at which a spiral antenna can operate are determined by two factors. The upper limit of frequency is determined by the radius of the feed ring. The lower limit of frequency is determined by the circumference of the outer ring the the spiral antenna. Spiral antennas can be enlarged by adding additional turns to the spiral arms. The more turns that are introduced the lower in frequency the spiral will operate. Increasing the number of spiral turns, however, greatly increases the number of unknowns present in the computational model since disparity in triangle dimensions lead to ill-conditioned systems. Hence, the spiral antenna is a geometrically constrained problem as opposed to a electrically constrained one.

6.1.2 Results

To determine how well the mixed element formulation would scale in terms of a geometrically constrained problem, a four arm spiral with one, two, and three turns was modeled. Table 6.1 shows the dimensions used for the various spirals.

A frequency range of 12-15 GHz was chosen at which to generate computational results. This frequency range was chosen because it was in the range of all spiral antennas studied and being at the higher end of the aperture, spiral termination

Turns	Circumference (cm)	Feed Radius (cm)	Mode 1 (GHz)	Mode 2 (GHz)
1	1.58	0.3	3.00-16	6.0-30.0
2	2.83	0.3	1.50-16	3.0-30.0
3	4.10	0.3	1.16-16	2.3-30.0

Table 6.1. Dimensions of various four arm spiral antennas

is not a major factor in radiation performance. Antenna radiation patterns were computed at 12 GHz for $\phi = 0°$ and $\phi = 90°$ and impedance measurements were computed for all four spiral feeds from 12-15 GHz. Each spiral antenna was placed on a 30 mill dielectric substrate ($\epsilon_r = 3.2 - j0.095$) and backed by 2 cm of absorber ($\epsilon_r = 6.0 - j6$) used to suppress the back wave of the antenna.

One challenge in modeling the performance of an antenna is the strong dependence of the impedance on local mesh details. In theory, the impedance of each feed should be identical. However, in practice, the mesh generation creates some mesh irregularities near each feed. The result is slightly different impedances. The could be compensated by massive over-sampling; however doing so makes the tool impractical for antenna study.

Figure 6.1 depicts a four arm, one turn spiral antenna. Pattern and impedance results for this antenna were computed at the frequencies and angles described in the above paragraph. Figure 6.2-Figure 6.5 illustrate the pattern and impedance results for the one turn spiral excited in mode 1. It is seen that the results for the mixed element formulation are in good agreement with the results for the prism element formulation. Figure 6.6-Figure 6.9 illustrate the pattern and impedance results for the one turn spiral antenna excited in mode 2. Once again, the mixed element results are in good agreement with the prism element results. In both cases, the slight discrepancy in the impedance results may be attributed to the slightly different mesh density around the feed points.

Figure 6.10 depicts a four arm, two turn spiral antenna. As with the one turn spiral, pattern and impedance results were computed for this antenna. Figure 6.11-Figure 6.14 illustrate the results for the antenna excited in mode 1 while Figure 6.15-Figure 6.18 illustrate the results for the antenna excited in mode 2. Similar to the one turn spiral, the results for the two formulations are in good agreement for the two turn spiral. As before, there are slight discrepancies in the impedance results caused by different mesh densities around the feed points. These impedance differences cause different input powers to be present at the feed points which can subtly affect the pattern results. While this effect was negligible for the one turn spiral it is a bit more noticeable for the two turn. While the shape of the pattern matches well, the peak values can vary slightly.

Figure 6.19 depicts a four arm, three turn spiral antenna. Figure 6.20-Figure 6.23 illustrate the results for the antenna excited in mode 1 while Figure 6.24-Figure 6.27 illustrate the results for the antenna excited in mode 2. The results are similar to the two turn spiral with good agreement between the two formulations including slight variations in impedance and pattern.

6.2 I-Dipole Array Antenna

This section discusses and provides results for I-dipole array antennas. I-dipole array antennas are another example of a geometrically constrained problem similar to the four-arm spiral antennas discussed on the previous section.

6.2.1 Introduction

I-dipole array antennas are essentially the converse of spiral antennas. While spiral antennas exhibit wide-band frequency characteristics, I-dipole array antennas are designed to resonate at a very specific point in the frequency spectrum making them very narrow-band. This resonant frequency is determined by many design factors. These factors can be grouped into two categories, individual dipole parameters and

array parameters. The dipole parameters include dipole height and dipole width along with the dimensions of the metallic portions of the dipole. The array parameters include the separation between the centers of two dipoles in the horizontal direction and the separation between the centers of two dipoles in the vertical direction along with the distance from the outer dipoles to the surrounding cavity. It is noted that the bandwidth of the array can be substantially increased by tightly packing the array and hence utilizing the mutual coupling. The fact that the dipoles are end loaded produces a larger effective area for the dipole in a smaller amount of physical space compared to a standard dipole. This end loading allows the dipoles to be packed very tightly together in a minimum amount of space giving a larger bandwidth.

6.2.2 Results

To study the validity of the mixed element formulation I-dipole arrays of one, four, and nine dipoles were created. The various parameters used for the array are shown in Table 6.2.

W_d	0.640 cm
H_d	0.750 cm
W_{dd}	0.800 cm
H_{dd}	1.000 cm
T_d	0.050 cm
G_d	0.100 cm
W_{dc}	0.375 cm
H_{dc}	0.375 cm

Table 6.2. Parameters for I-dipole array antenna.

W_d is the width of the dipole, H_d is the height of the dipole, W_{dd} is the dipole to dipole separation in the horizontal direction, H_{dd} is the dipole to dipole separation in the vertical direction, T_d is the dimension of the metallic portions of the dipole,

G_d is the dimension of the feed gap of the dipole, W_{dc} is the horizontal distance from the outer dipole to the surrounding cavity, and H_{dc} is the vertical distance from the outer dipole to the surrounding cavity.

Figure 6.28-Figure 6.30 show the one, four and nine dipole array antennas while Figure 6.31 shows the backscatter radar cross section of the arrays with the excitation being an incident plane wave at an angle of $\phi = 0°$ from 3-5 GHz. It can be seen from the figure that the results for the mixed element formulation and the prism element formulation are in very good agreement. Using the available computational resources a nine element array was the limit using prism elements for comparison purposes with the mixed element formulation.

6.3 Mixed Elements vs. Prism Elements

The previous section show that the mixed element formulation produces approximately the same results as the prism formulation. Now it is important to show that the mixed element formulation is more efficient in terms of memory and computational demand and that the mixed element formulation scales better for geometrically constrained problems.

6.3.1 Memory Demand

The most important parameter used to determine the memory demand of any general FE-BI computational problem is the number of surface unknowns present in the model. Reducing the number of surface unknowns is one of the major goals of the mixed element formulation discussed in this dissertation. To compare the memory demand used by the two different formulations the spiral antenna and I-dipole array antenna were used as a benchmark to determine how much the mixed element formulation was able to reduce the number of surface unknowns in addition to how well the mixed element formulation scaled as the antenna dimensions were increased.

Figure 6.32 shows the number of surface unknowns needed to model a spiral

antenna versus the number of turns for both the mixed element and prism element formulations. Figure 6.33 shows the number of surface unknowns needed to model the I-dipole antenna array versus the number of dipoles for both the mixed element and prism element formulations. As seen in both figures, the number of surface unknowns is decreased for the mixed element formulation at each step. Also, as the dimensions of the antenna is increased the number of surface unknowns mixed element formulation rises more slowly than the prism element formulation. Therefore, the mixed element formulation scales better for geometrically constrained problems.

6.3.2 Computational Demand

The previous section showed that the mixed element formulation produces far fewer surface unknowns for the same geometry as opposed to a formulation using only prisms. This reduction in surface unknowns should therefore lead to more efficiency in terms of computation. In other words, the same problem should run faster using the mixed element formulation. To quantify these results a number of benchmarks were run for various spiral and I-dipole array antennas. The following results were computed on a quad-processor linux machine with processor speeds of 450 MHz and 2 Gb of RAM. The program was run in serial mode.

Table 6.3 shows the comparative results of a pattern cut computation for a spiral antenna with an increasing number of turns using both the mixed element formulation and the prism element formulation. Figure 6.34 shows a graph of the same data. It can be seen from the table and the figure that the mixed element formulation runs faster than the corresponding prism formulation and that the mixed element formulation scales better as the number of turns is increased. It is also important to note that the four turn spiral could only be modeled with the mixed element formulation because the prism element formulation required too much memory.

Table 6.4 shows the comparative results of a pattern cut computation for an I-dipole array antenna with an increasing number of dipoles. Figure 6.35 shows a graph

Turns	Mixed (hh:mm:ss)	Prism (hh:mm:ss)
1	00:14:29	00:18:48
2	00:47:15	01:09:47
3	02:16:49	03:46:48
4	03:41:14	too big to model

Table 6.3. Pattern computation time for various 4-arm spiral antennas.

of the same data. As with the spiral antenna, the mixed element formulation is more efficient in terms of computation time for each case and scales better as the number of dipoles is increased.

Dipoles	Mixed (hh:mm:ss)	Prism (hh:mm:ss)
1	00:05:07	00:08:40
4	00:41:19	01:14:37
9	02:54:34	05:18:44
16	09:00:24	18:02:15

Table 6.4. Pattern computation time for various I-dipole array antennas.

6.4 Remarks

From the above results it can be seen that the mixed element formulation works well for computationally complex geometries. The results are in good agreement with the prism element formulation and are computed in a more efficient manner. The slight discrepancies in impedance modeling are attributed to differing mesh densities around the feed points causing different input powers to be present in the feeds.

Figure 6.1. Four arm, one turn spiral antenna.

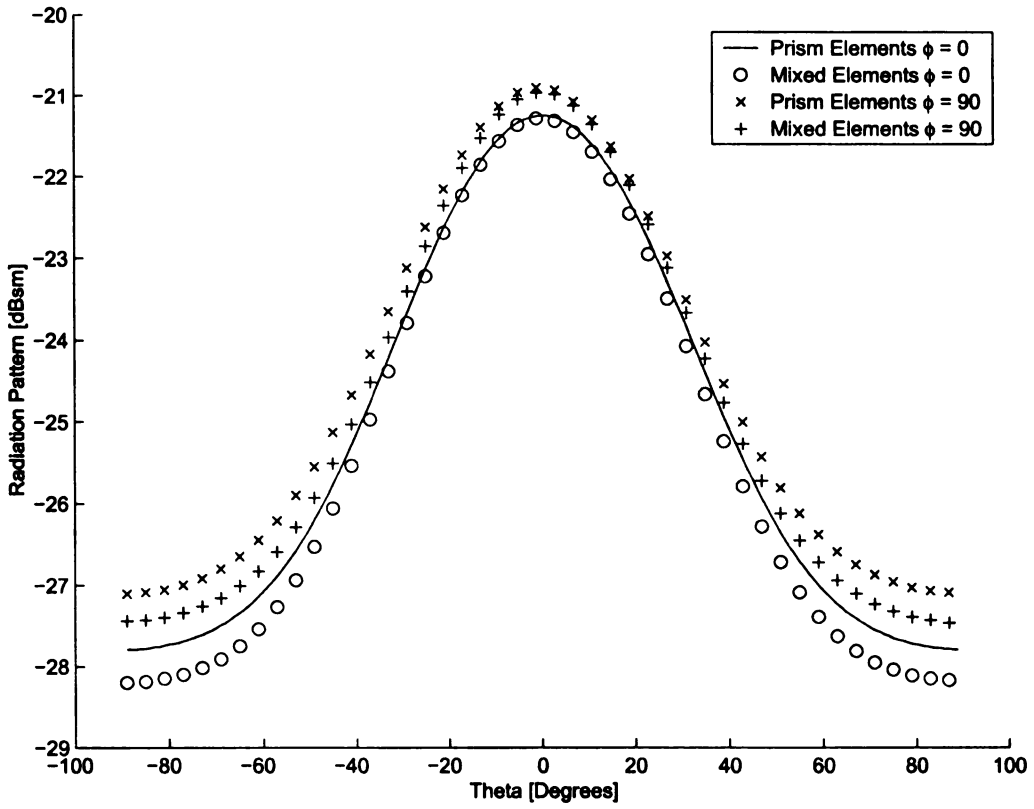

Figure 6.2. Mode 1 radiation pattern (σ_θ) of a 4-arm 1-turn spiral antenna.

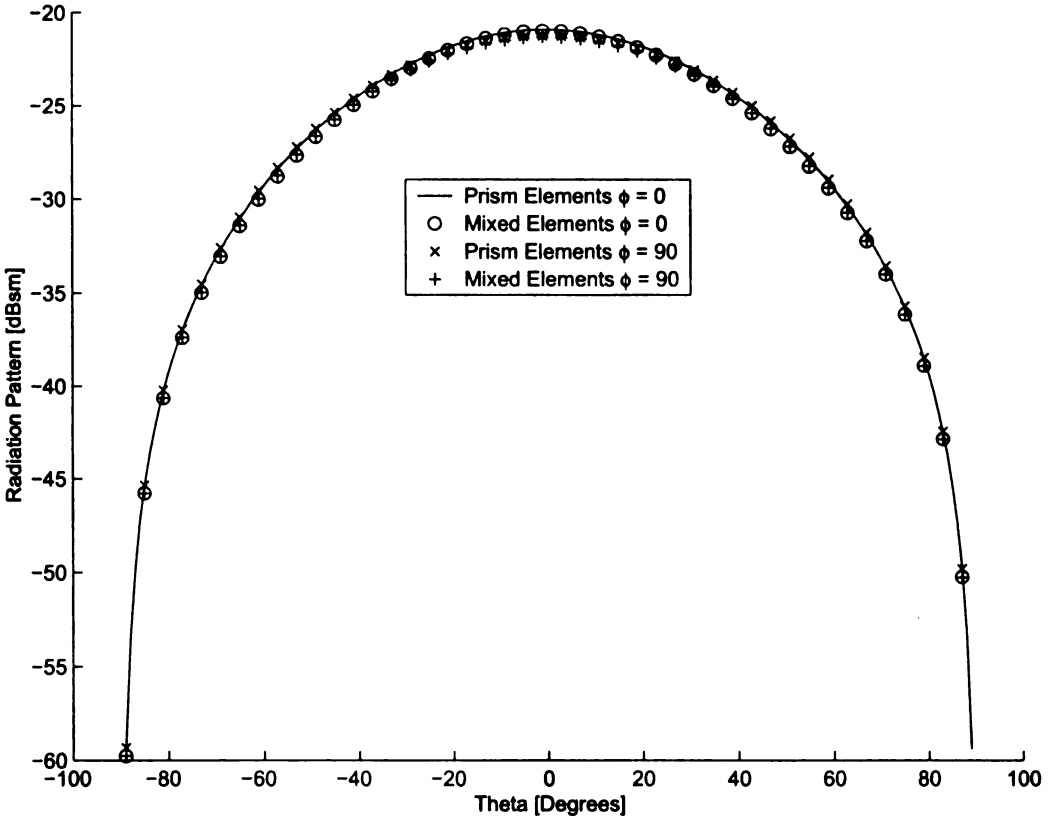

Figure 6.3. Mode 1 radiation pattern (σ_ϕ) of a 4-arm 1-turn spiral antenna.

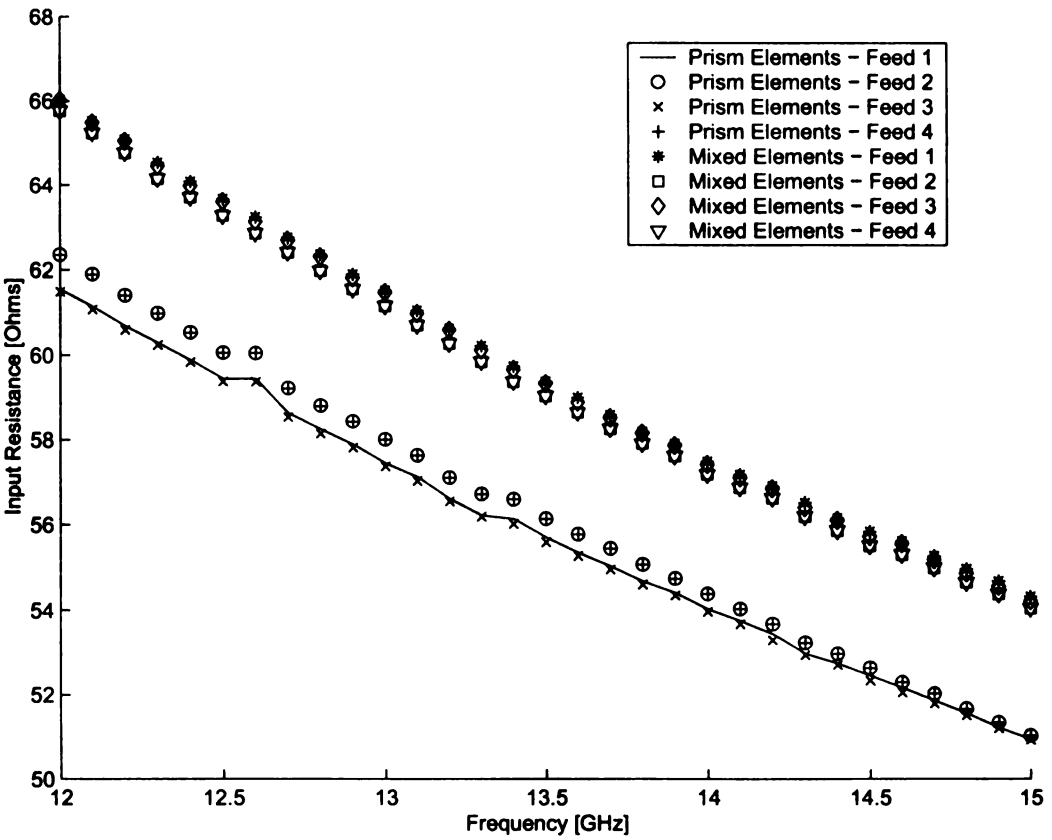

Figure 6.4. Mode 1 input resistance of a 4-arm 1-turn spiral antenna.

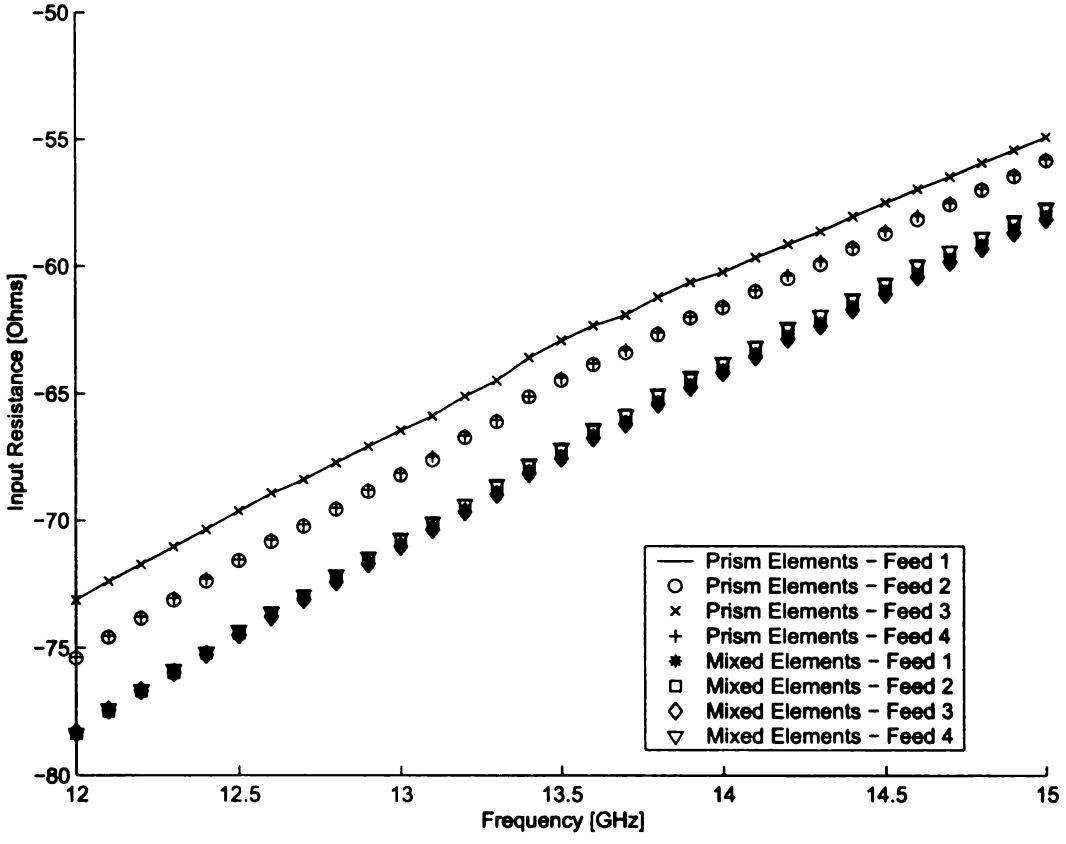

Figure 6.5. Mode 1 input reactance of a 4-arm 1-turn spiral antenna.

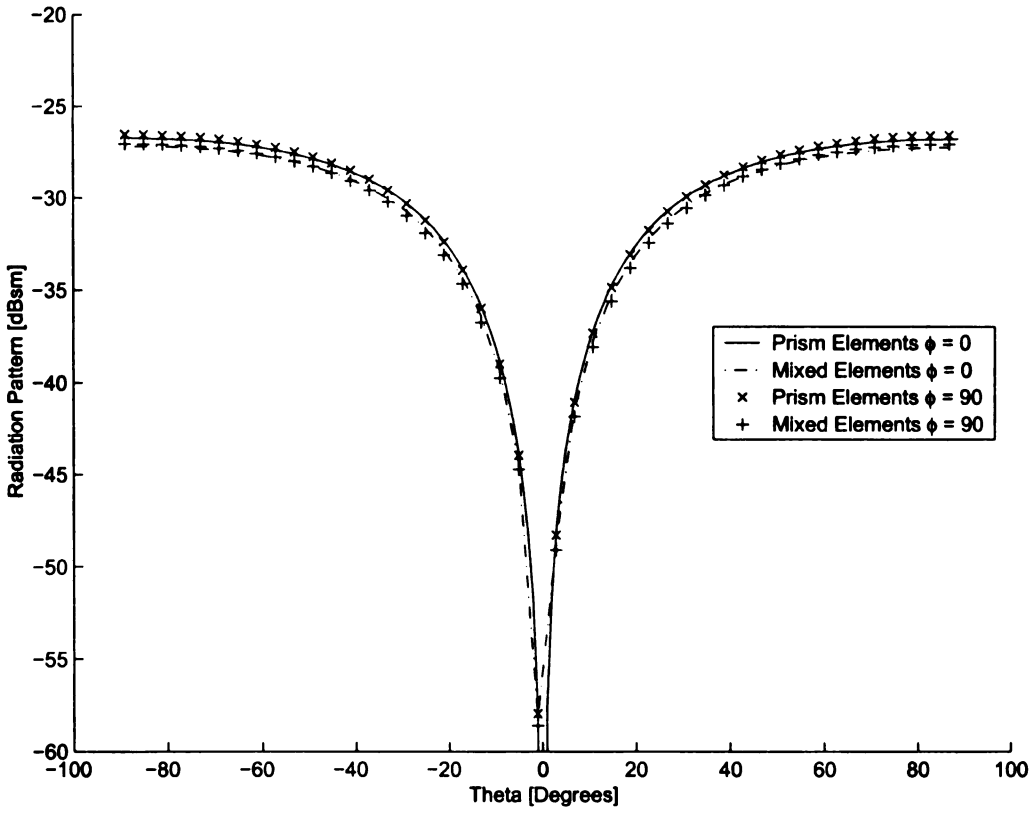

Figure 6.6. Mode 2 radiation pattern (σ_θ) of a 4-arm 1-turn spiral antenna.

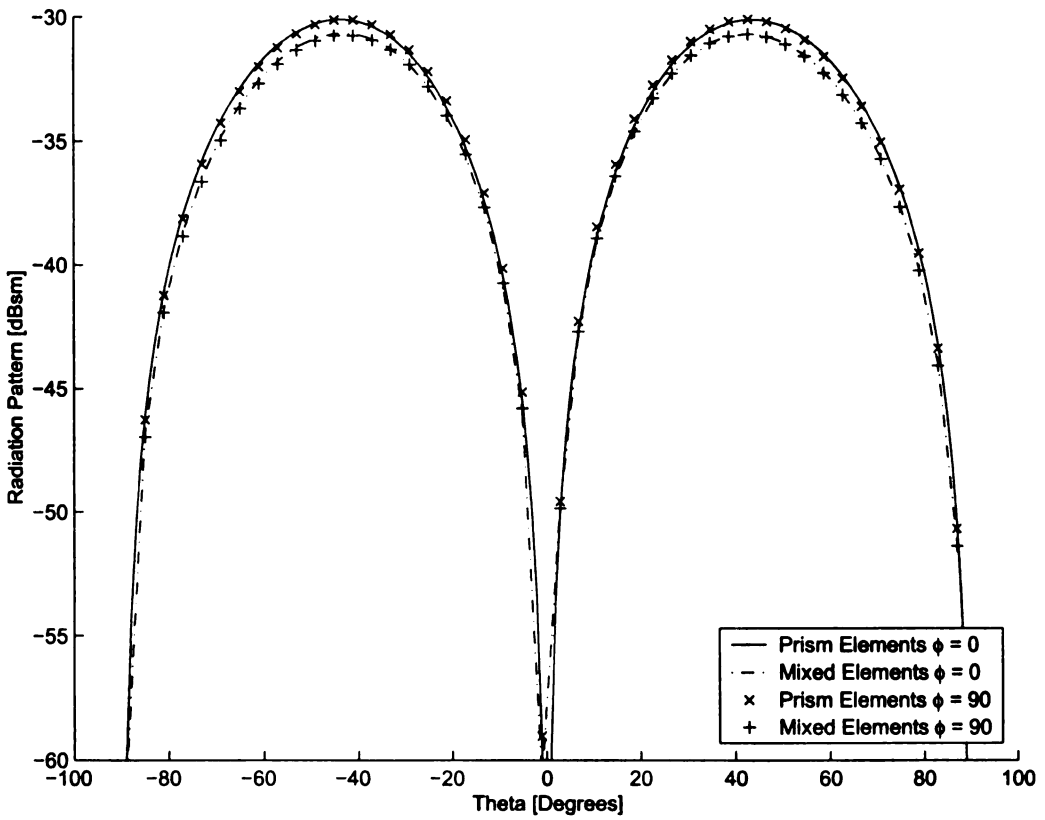

Figure 6.7. Mode 2 radiation pattern (σ_ϕ) of a 4-arm 1-turn spiral antenna.

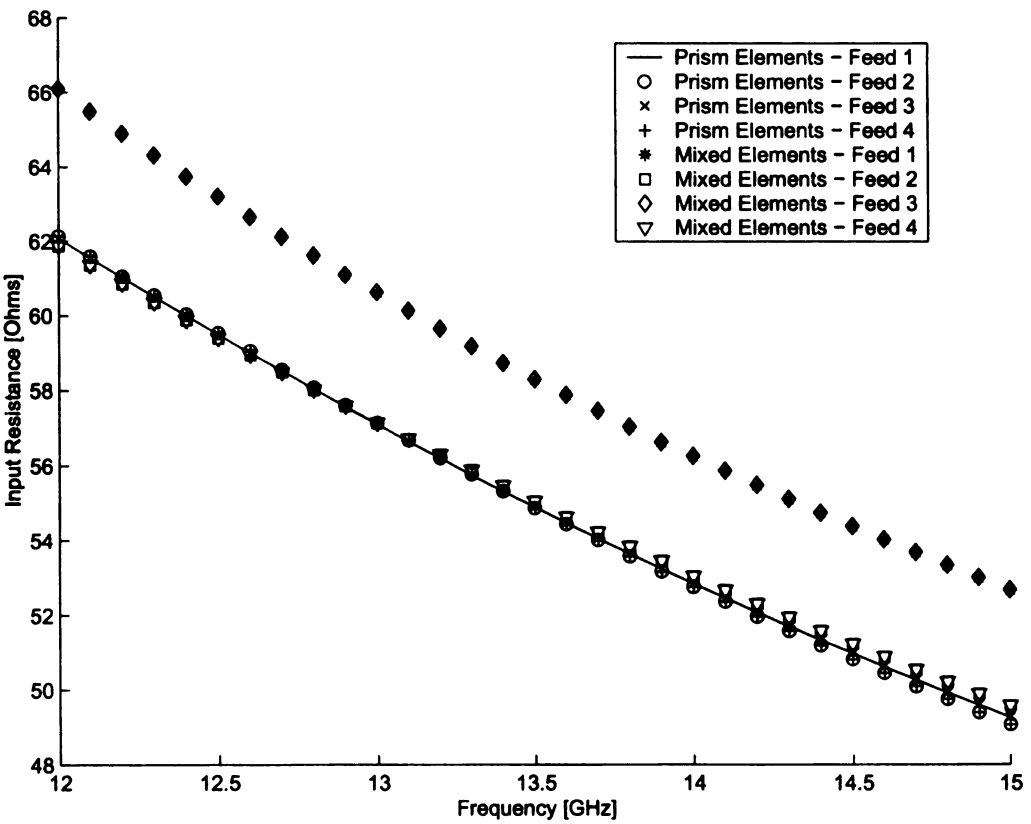

Figure 6.8. Mode 2 input resistance of a 4-arm 1-turn spiral antenna.

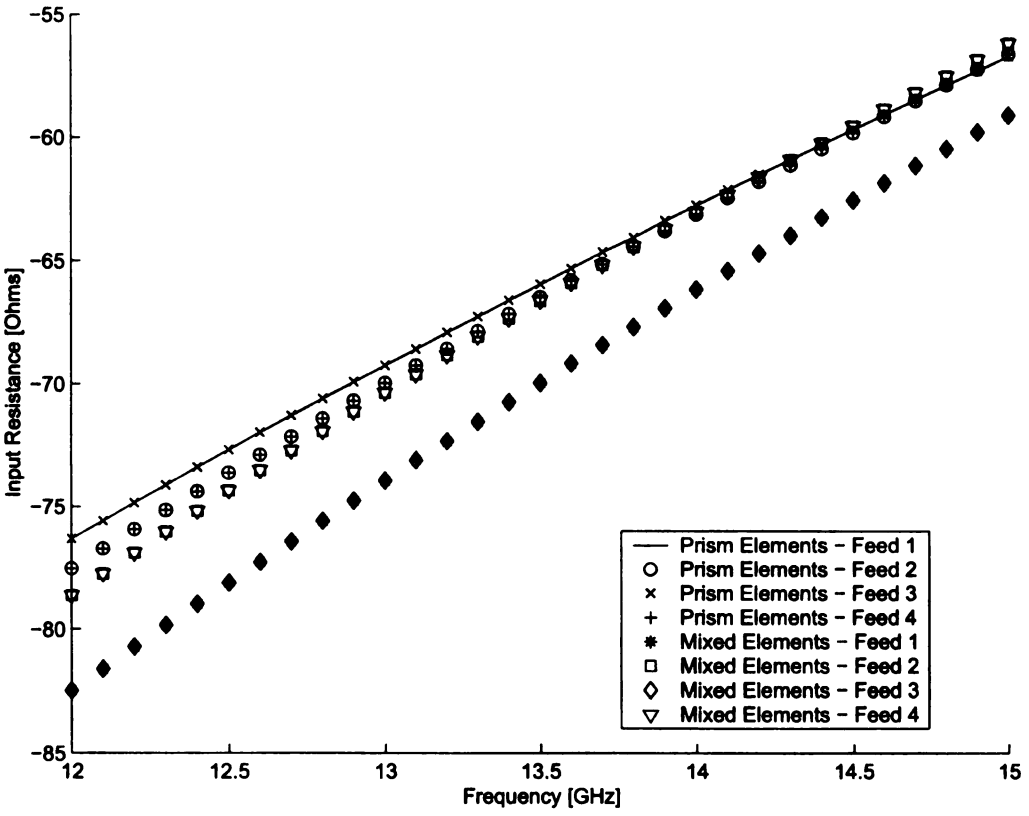

Figure 6.9. Mode 2 input reactance of a 4-arm 1-turn spiral antenna.

Figure 6.10. Four arm, two turn spiral antenna.

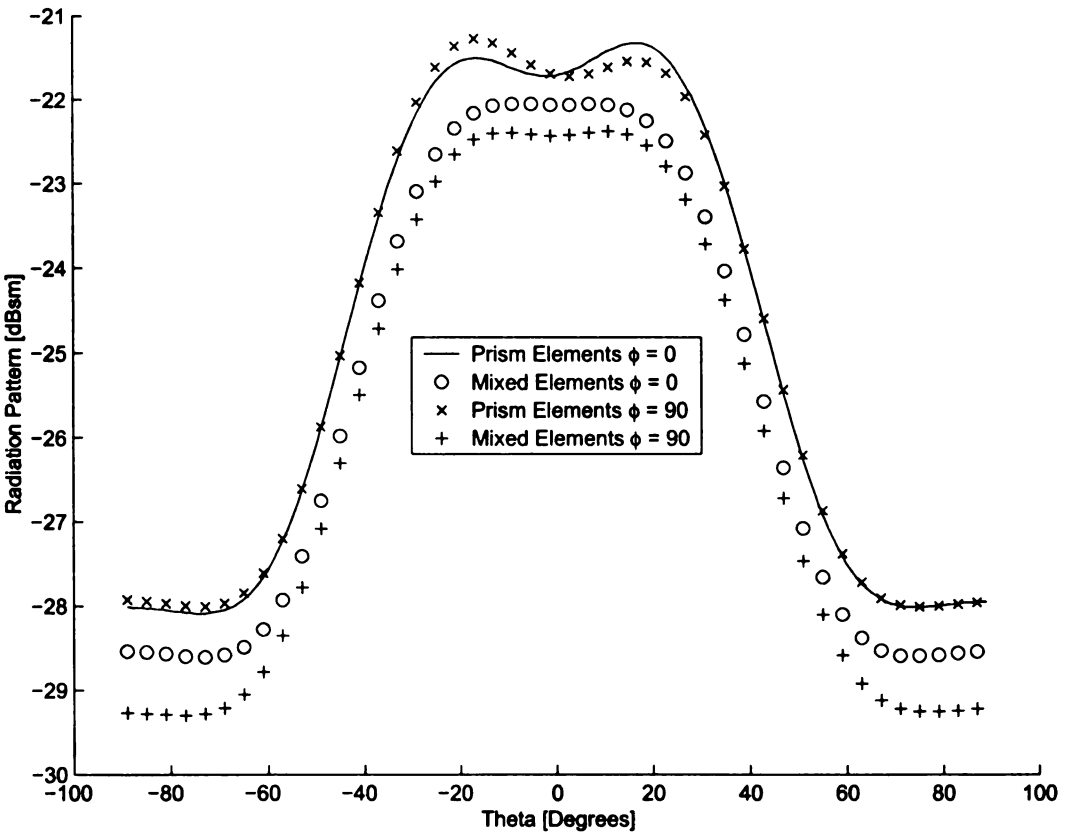

Figure 6.11. Mode 1 radiation pattern (σ_θ) of a 4-arm 2-turn spiral antenna.

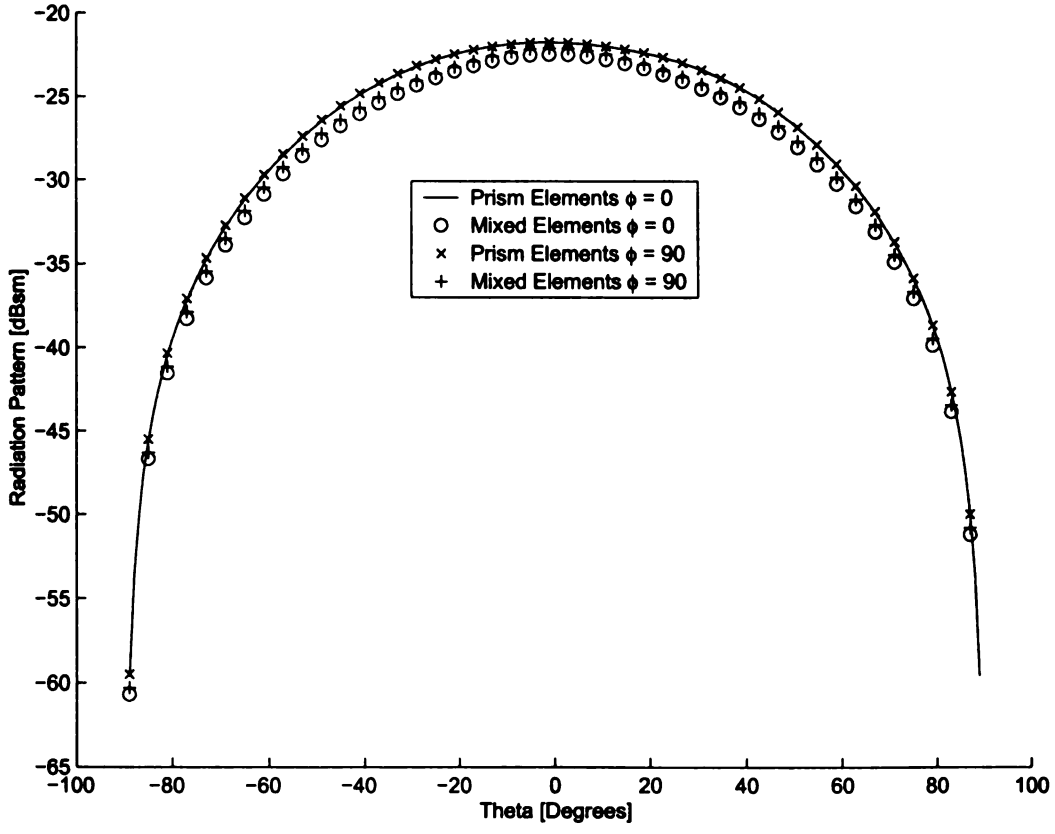

Figure 6.12. Mode 1 radiation pattern (σ_ϕ) of a 4-arm 2-turn spiral antenna.

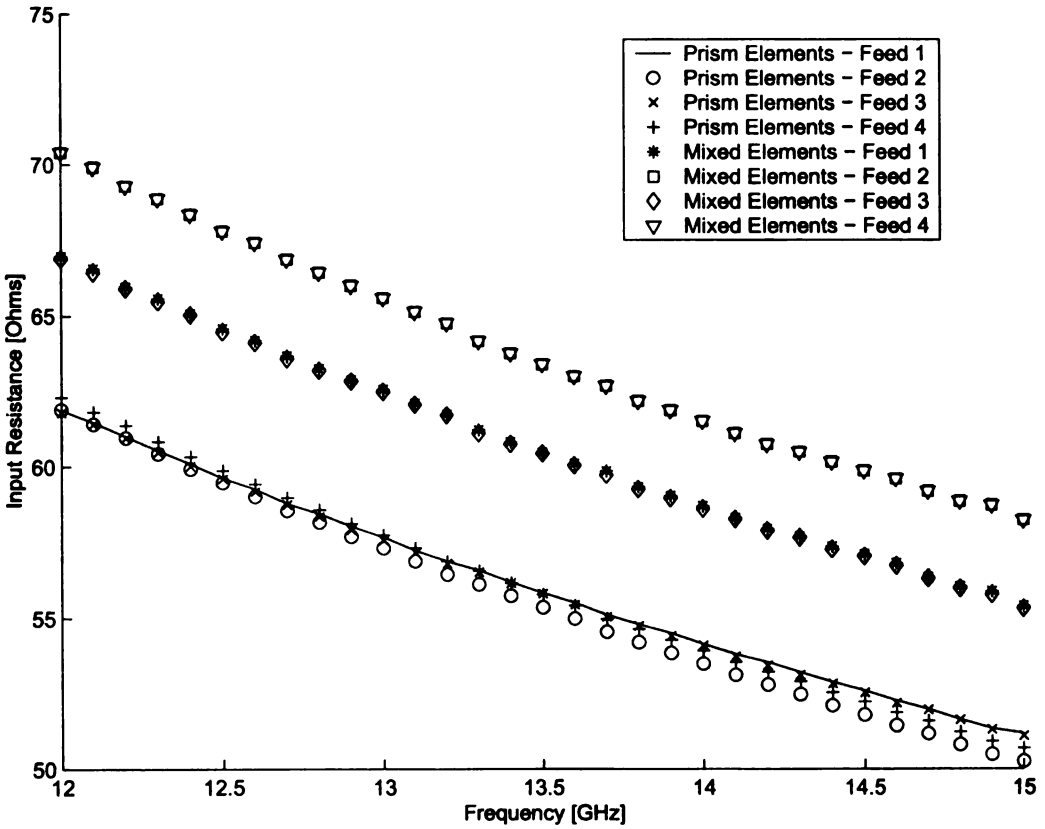

Figure 6.13. Mode 1 input resistance of a 4-arm 2-turn spiral antenna.

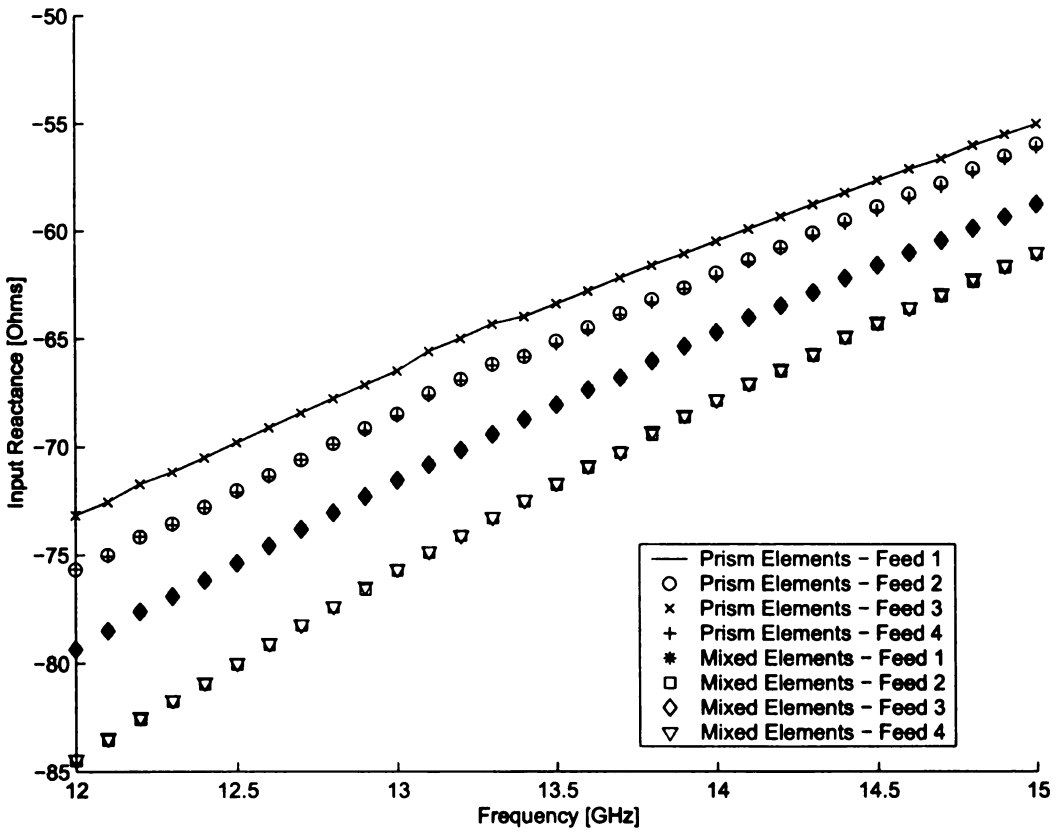

Figure 6.14. Mode 1 input reactance of a 4-arm 2-turn spiral antenna.

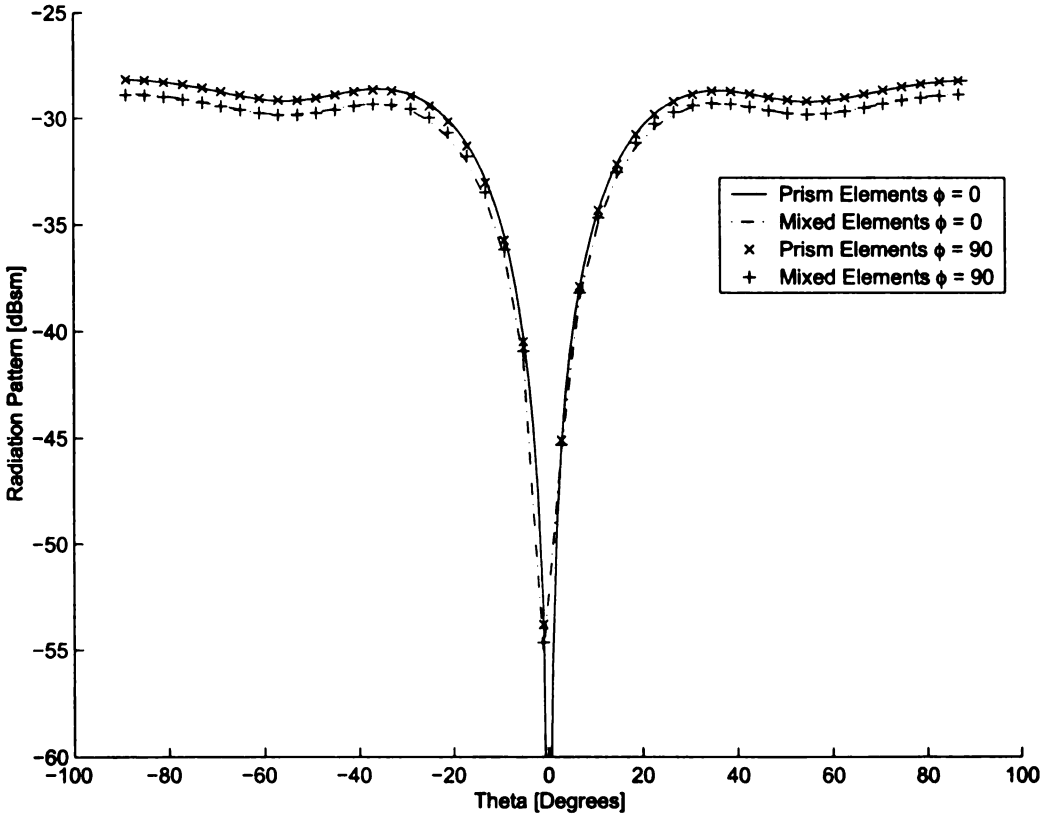

Figure 6.15. Mode 2 radiation pattern (σ_θ) of a 4-arm 2-turn spiral antenna.

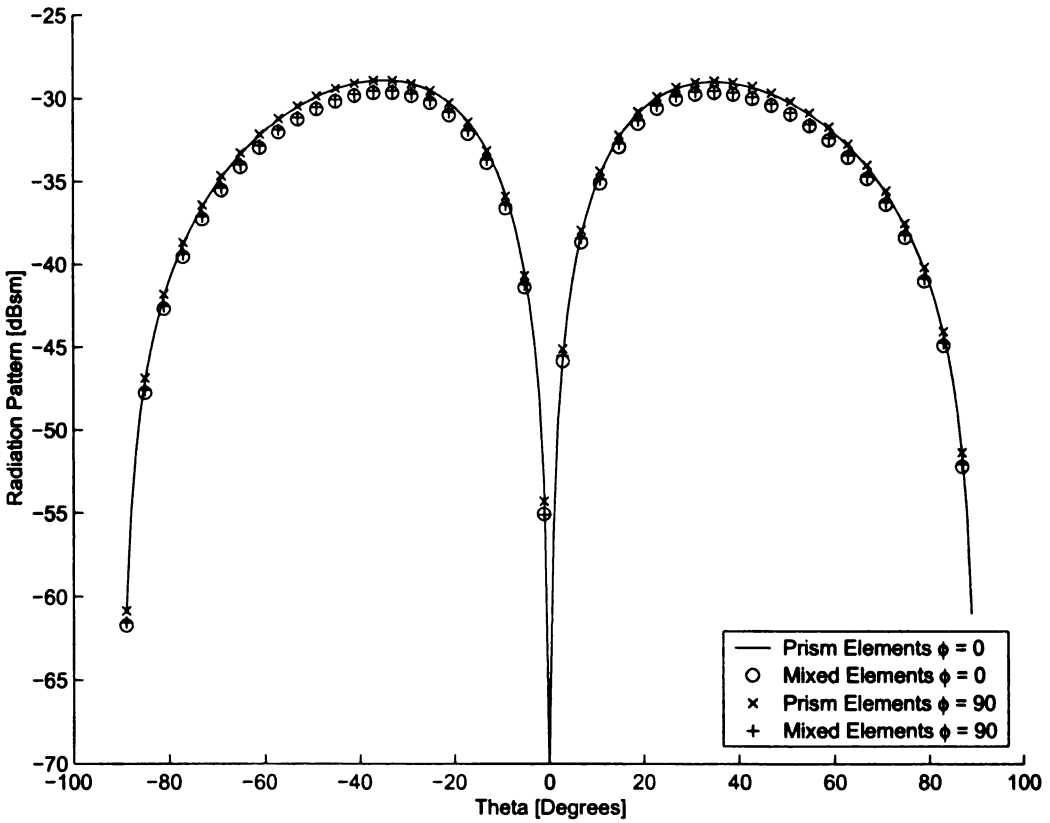

Figure 6.16. Mode 2 radiation pattern (σ_ϕ) of a 4-arm 2-turn spiral antenna.

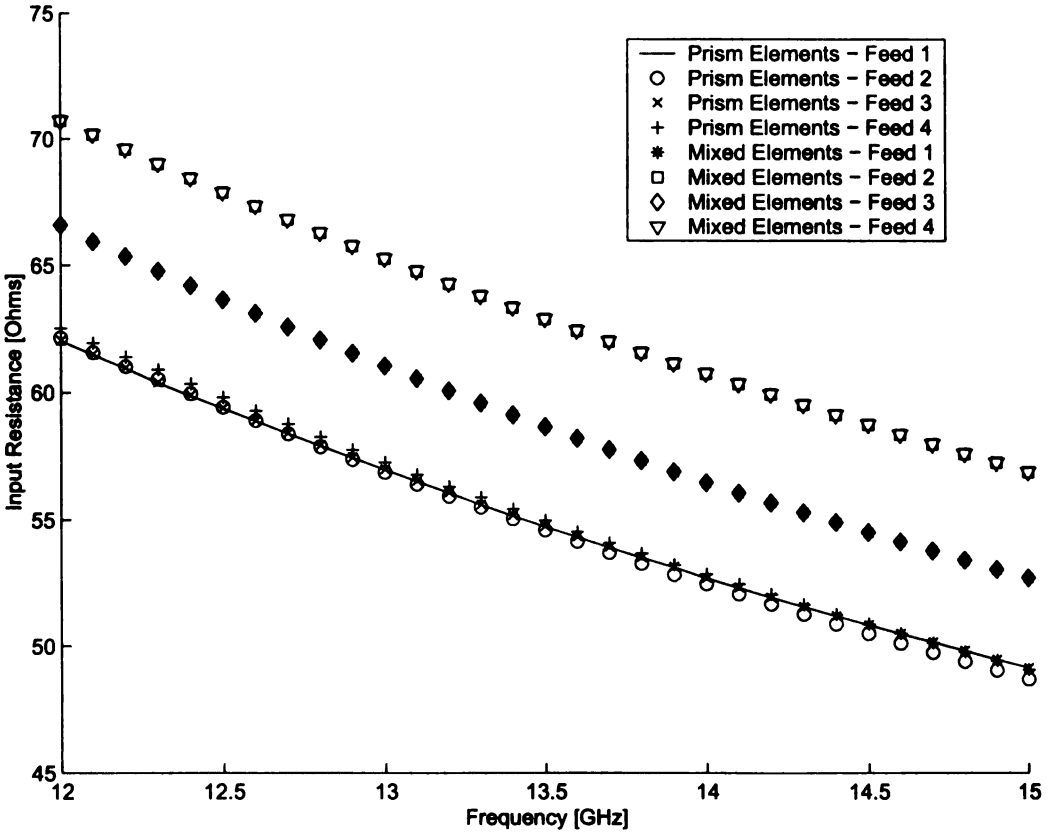

Figure 6.17. Mode 2 input resistance of a 4-arm 2-turn spiral antenna.

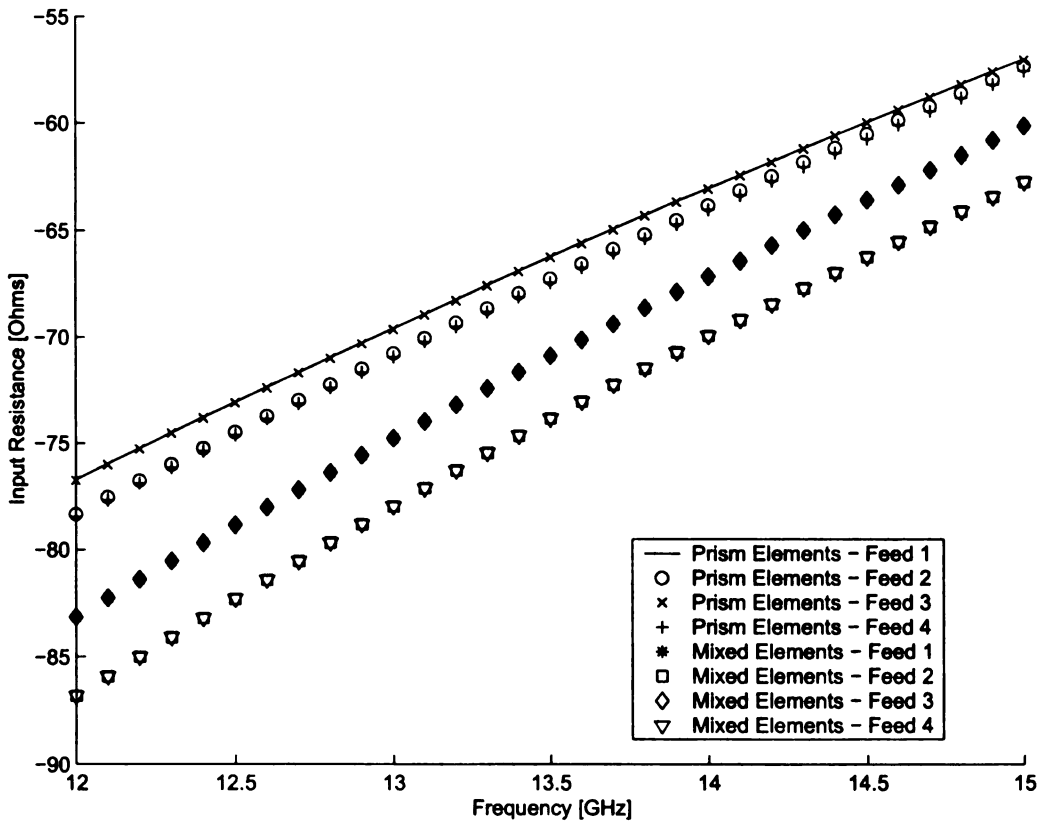

Figure 6.18. Mode 2 input reactance of a 4-arm 2-turn spiral antenna.

Figure 6.19. Four arm, three turn spiral antenna.

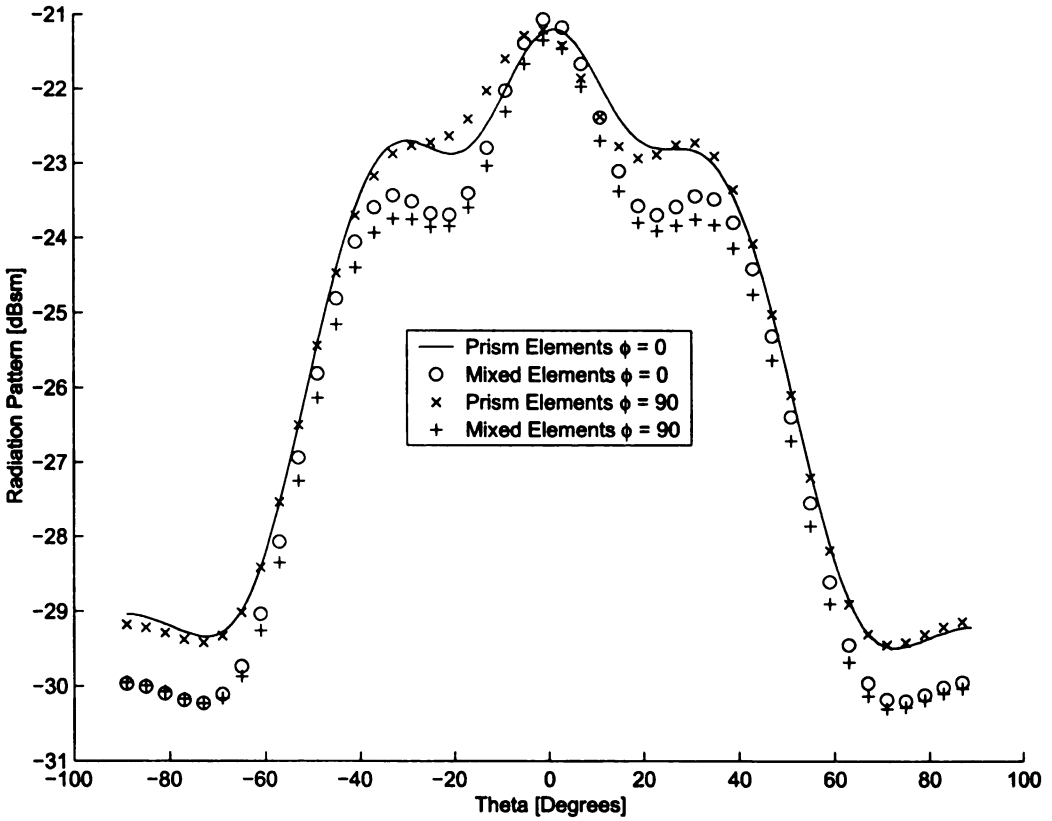

Figure 6.20. Mode 1 radiation pattern (σ_θ) of a 4-arm 3-turn spiral antenna.

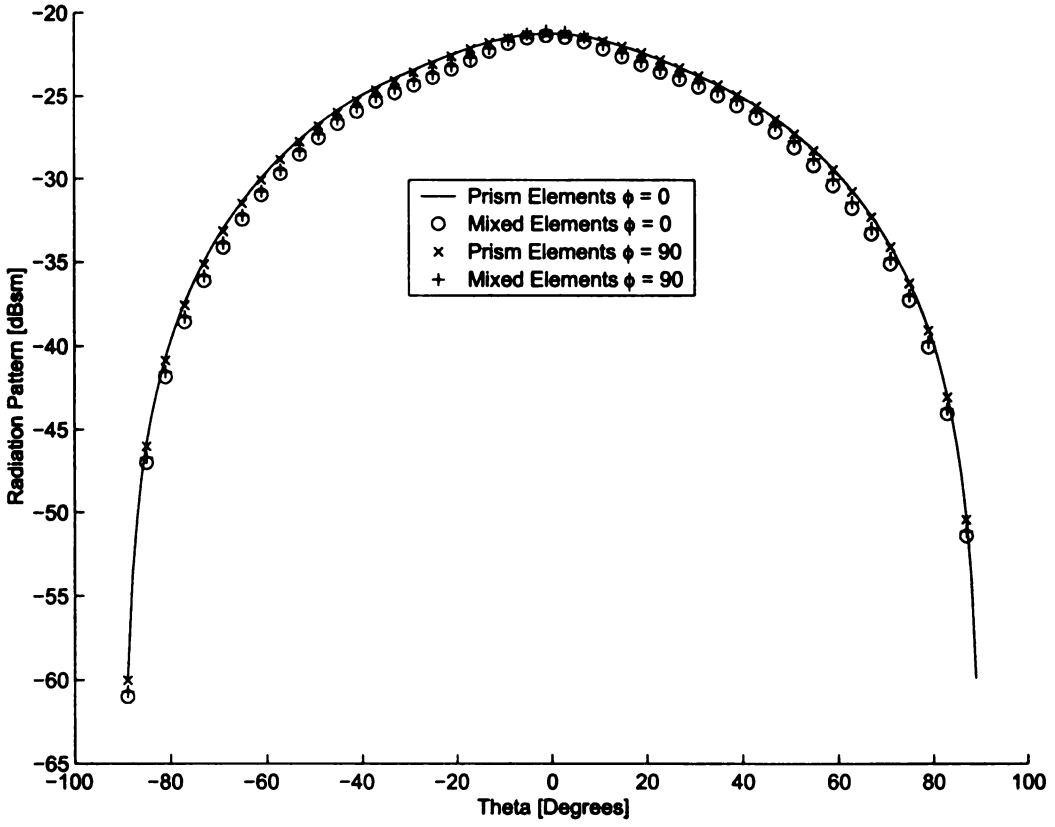

Figure 6.21. Mode 1 radiation pattern (σ_ϕ) of a 4-arm 3-turn spiral antenna.

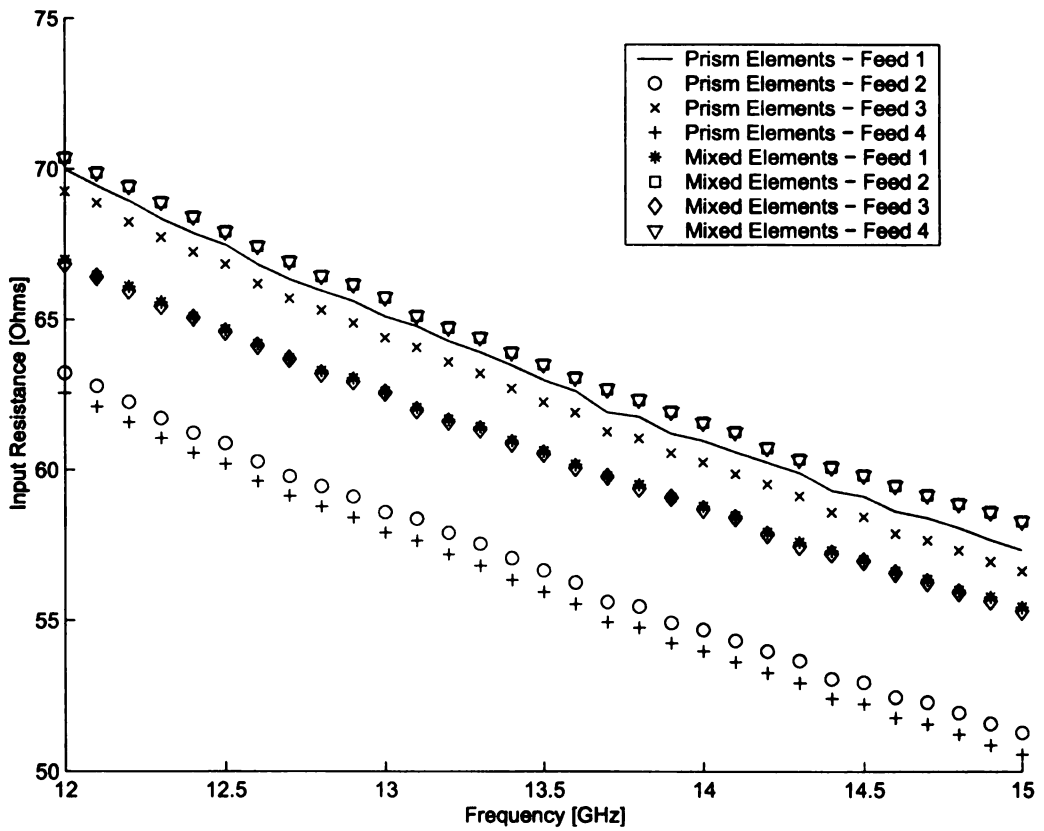

Figure 6.22. Mode 1 input resistance of a 4-arm 3-turn spiral antenna.

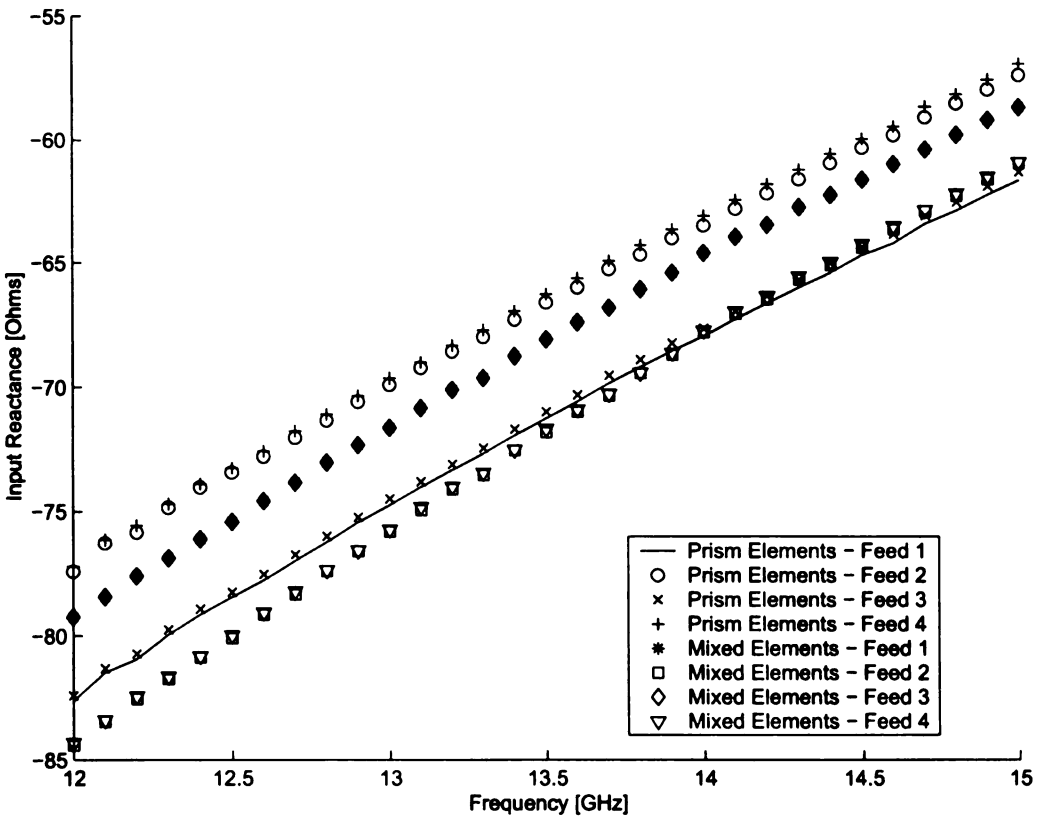

Figure 6.23. Mode 1 input reactance of a 4-arm 3-turn spiral antenna.

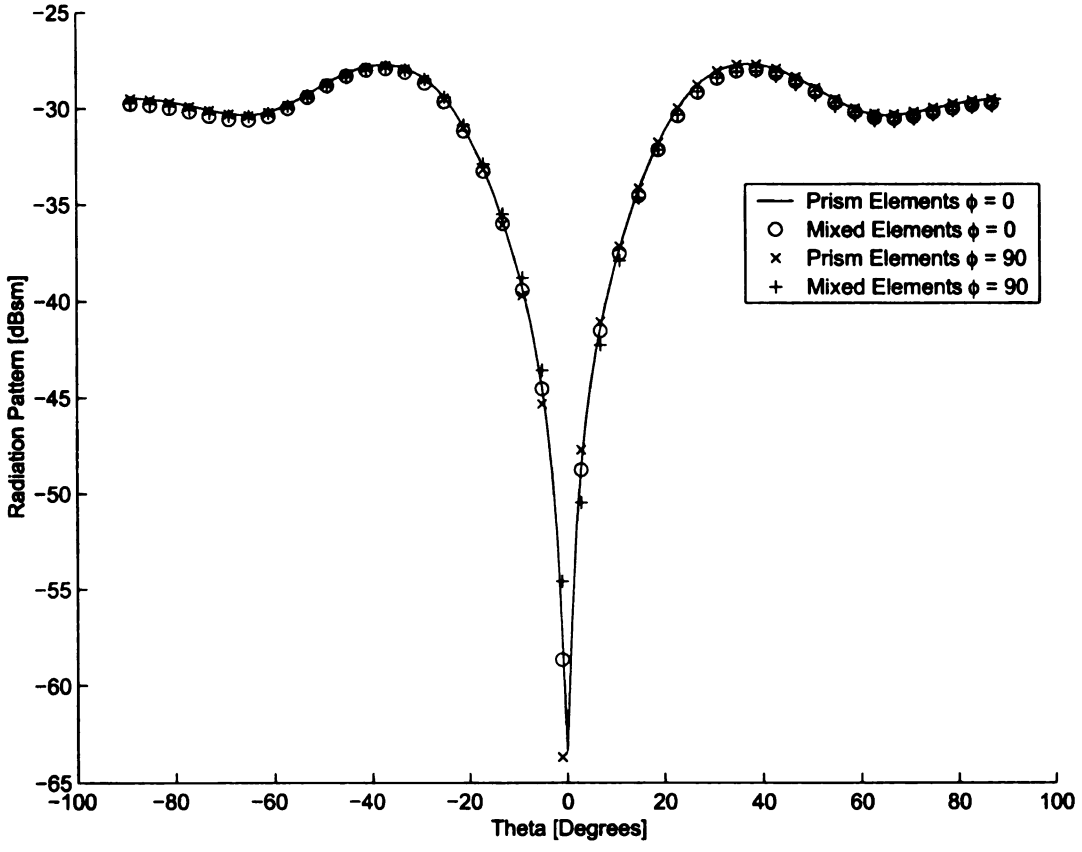

Figure 6.24. Mode 2 radiation pattern (σ_θ) of a 4-arm 3-turn spiral antenna.

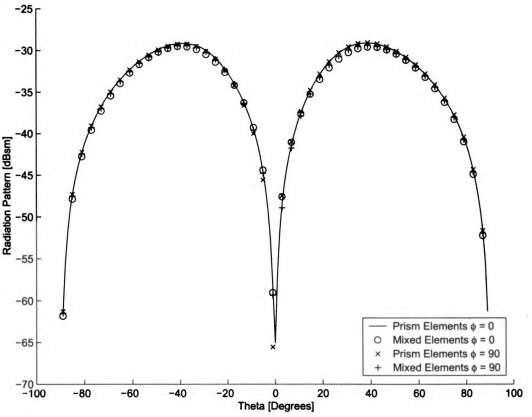

Figure 6.25. Mode 2 radiation pattern (σ_ϕ) of a 4-arm 3-turn spiral antenna.

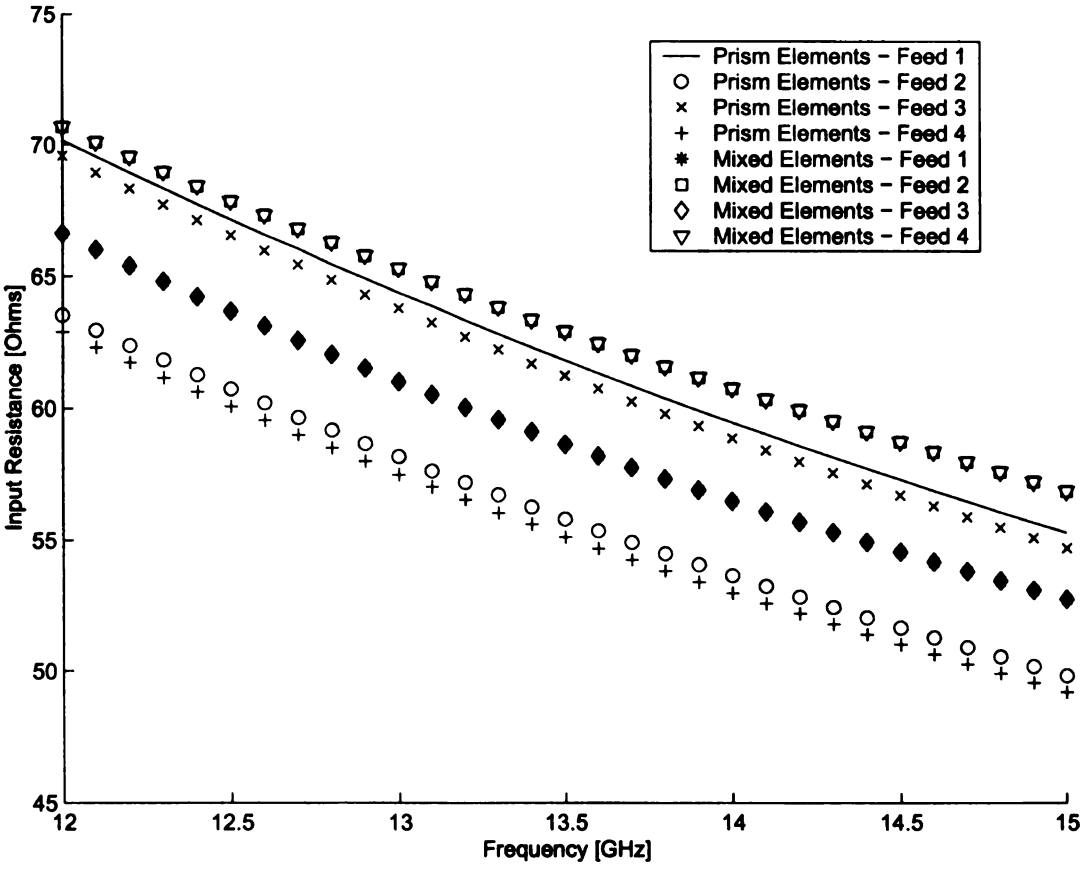

Figure 6.26. Mode 2 input resistance of a 4-arm 3-turn spiral antenna.

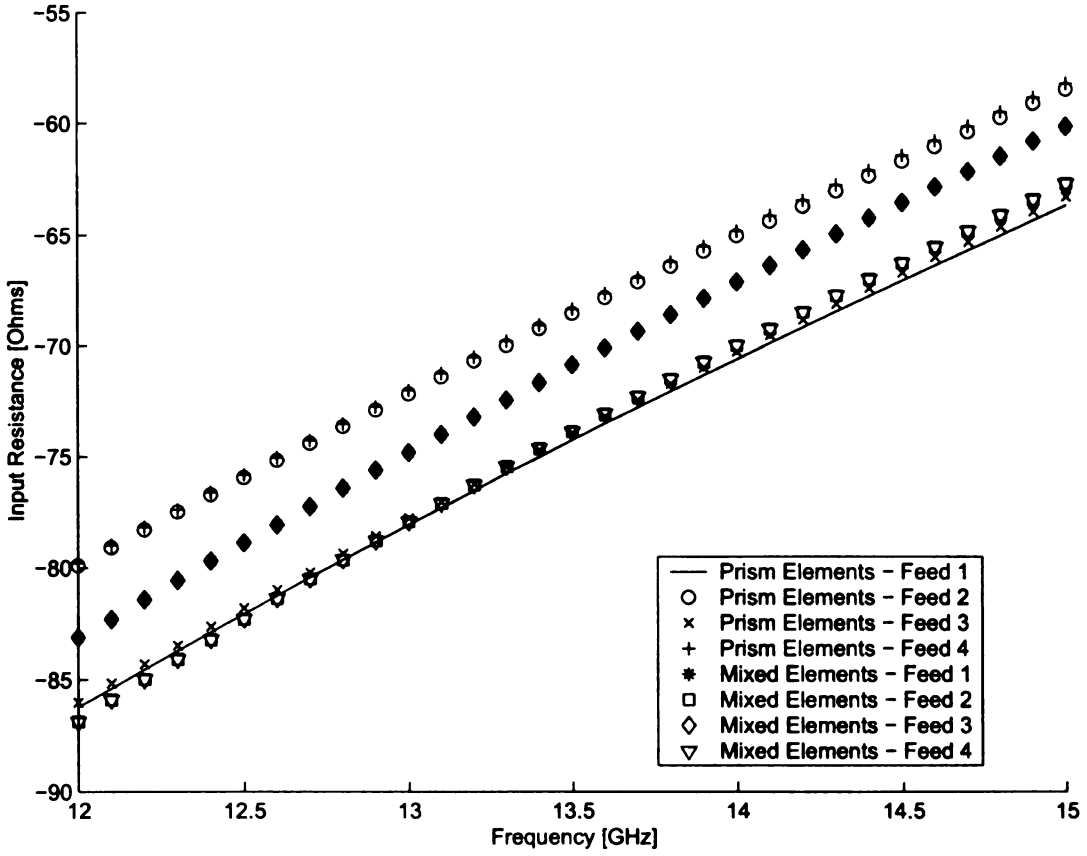

Figure 6.27. Mode 2 input reactance of a 4-arm 3-turn spiral antenna.

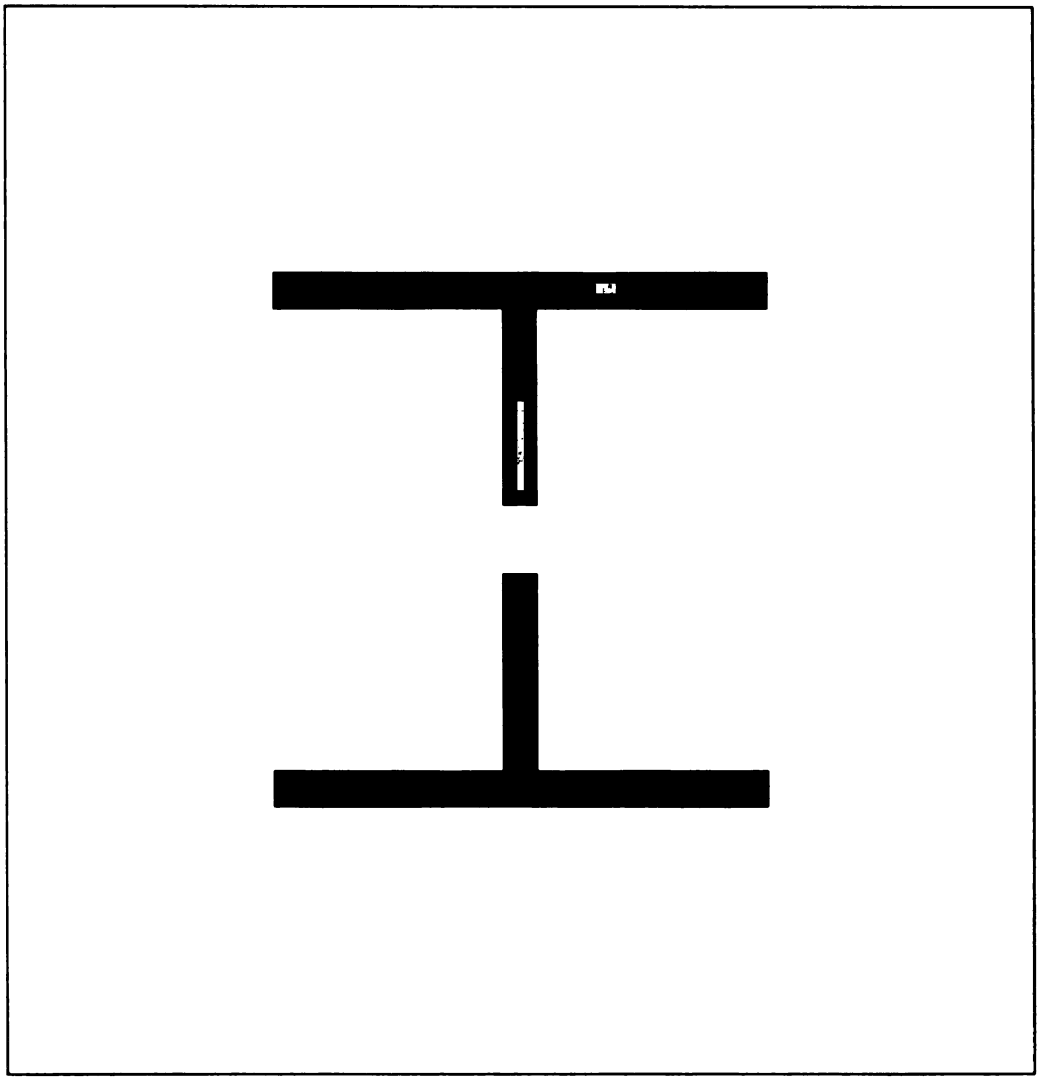

Figure 6.28. I-Dipole antenna array containing one dipole.

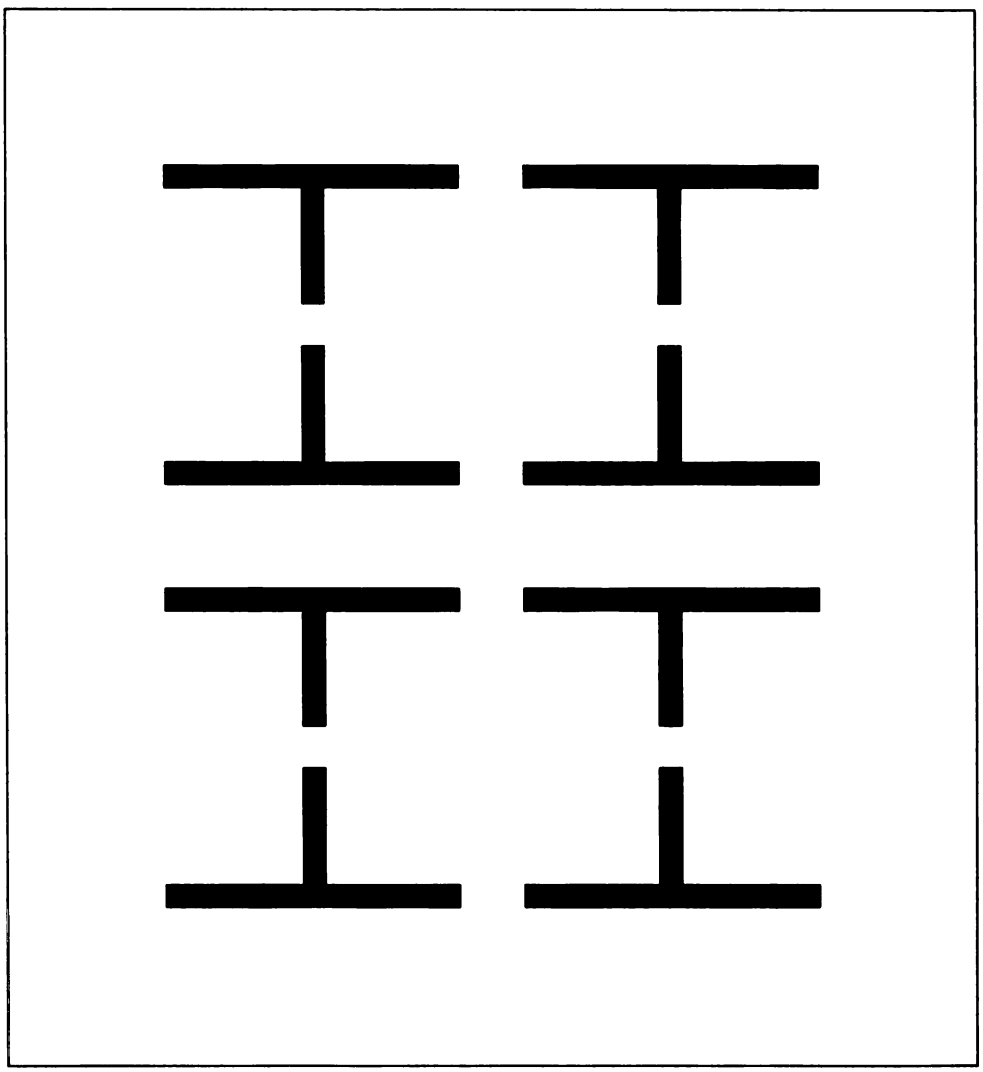

Figure 6.29. I-Dipole antenna array containing four dipoles.

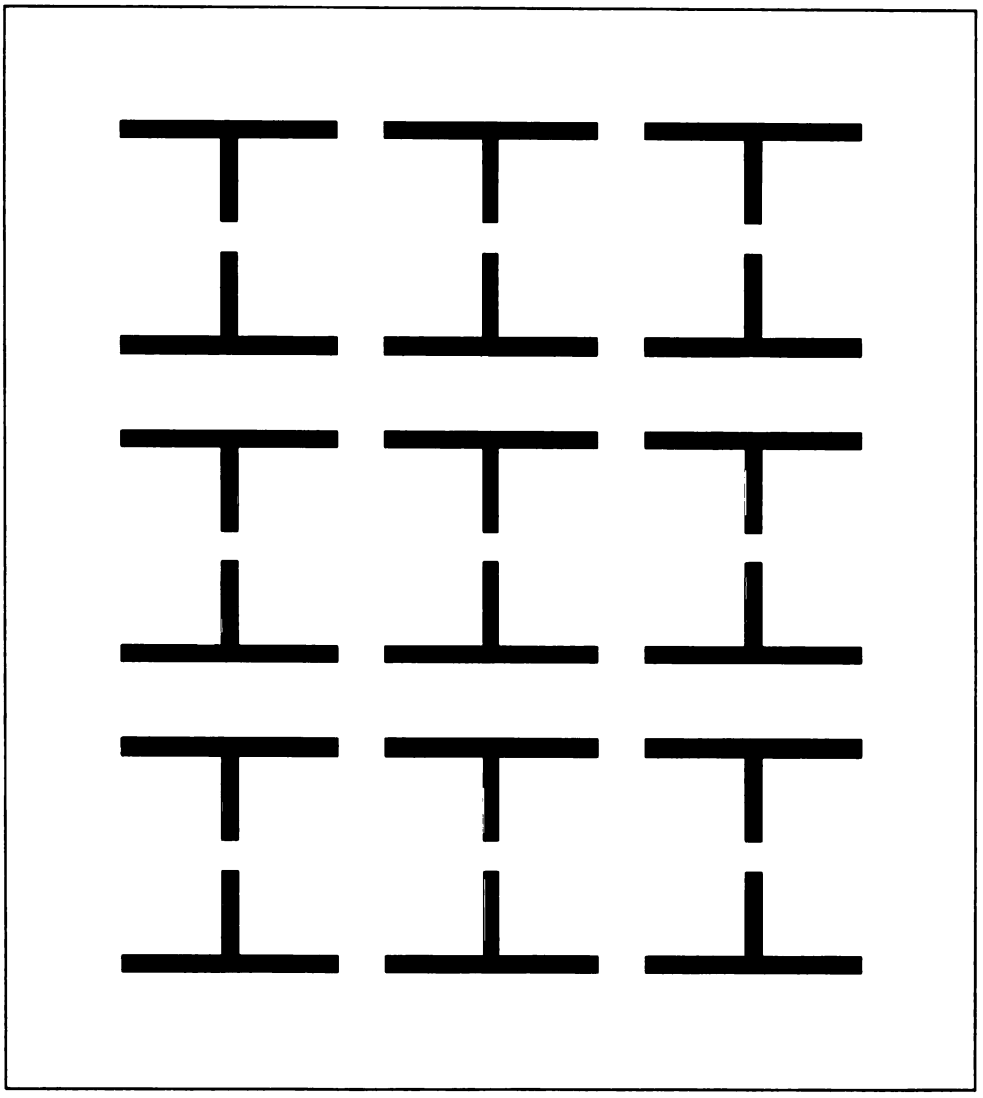

Figure 6.30. I-Dipole antenna array containing nine dipoles.

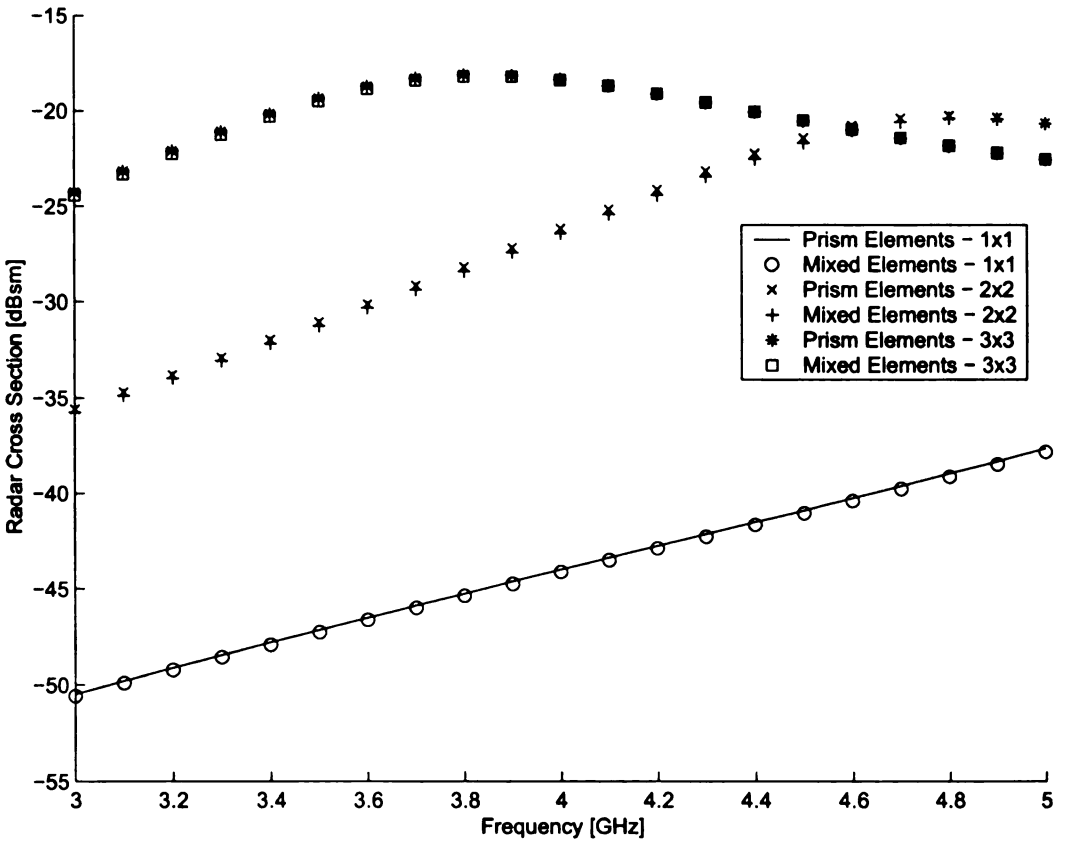

Figure 6.31. Backscatter radar cross section (σ_ϕ) of various i-dipole arrays.

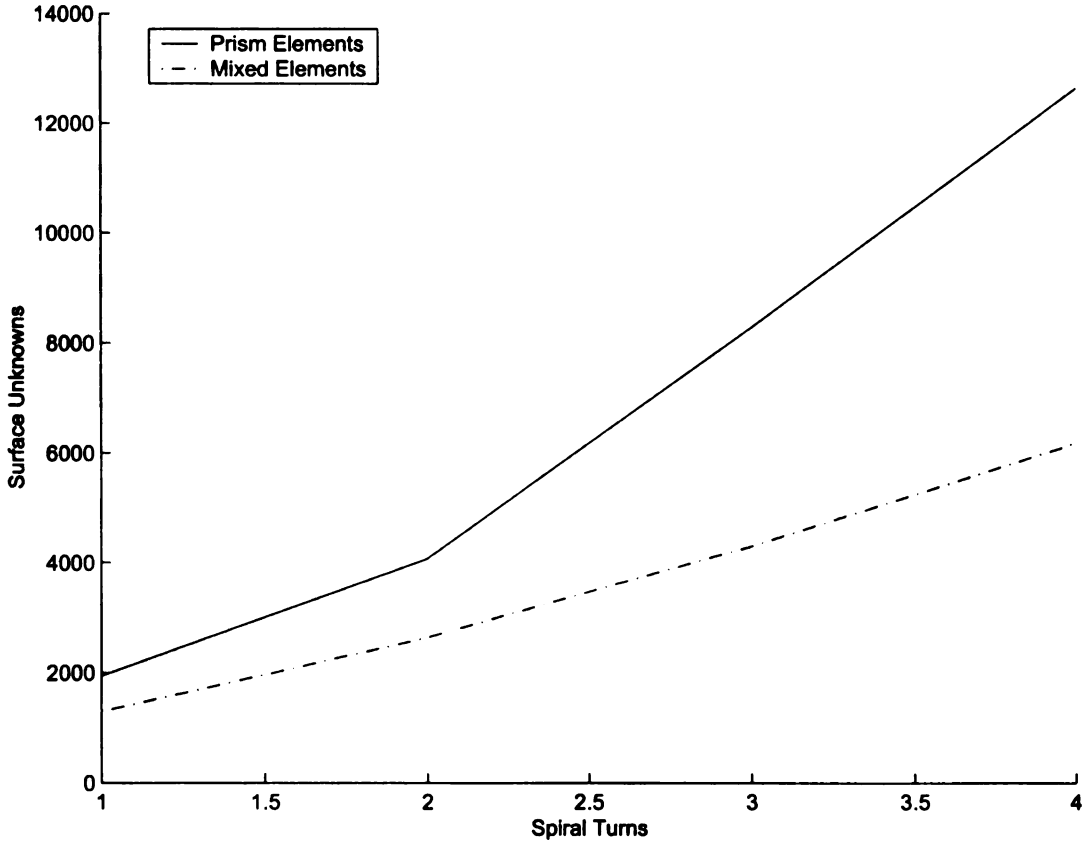

Figure 6.32. Number of surface unknowns in a spiral antenna versus the number of turns.

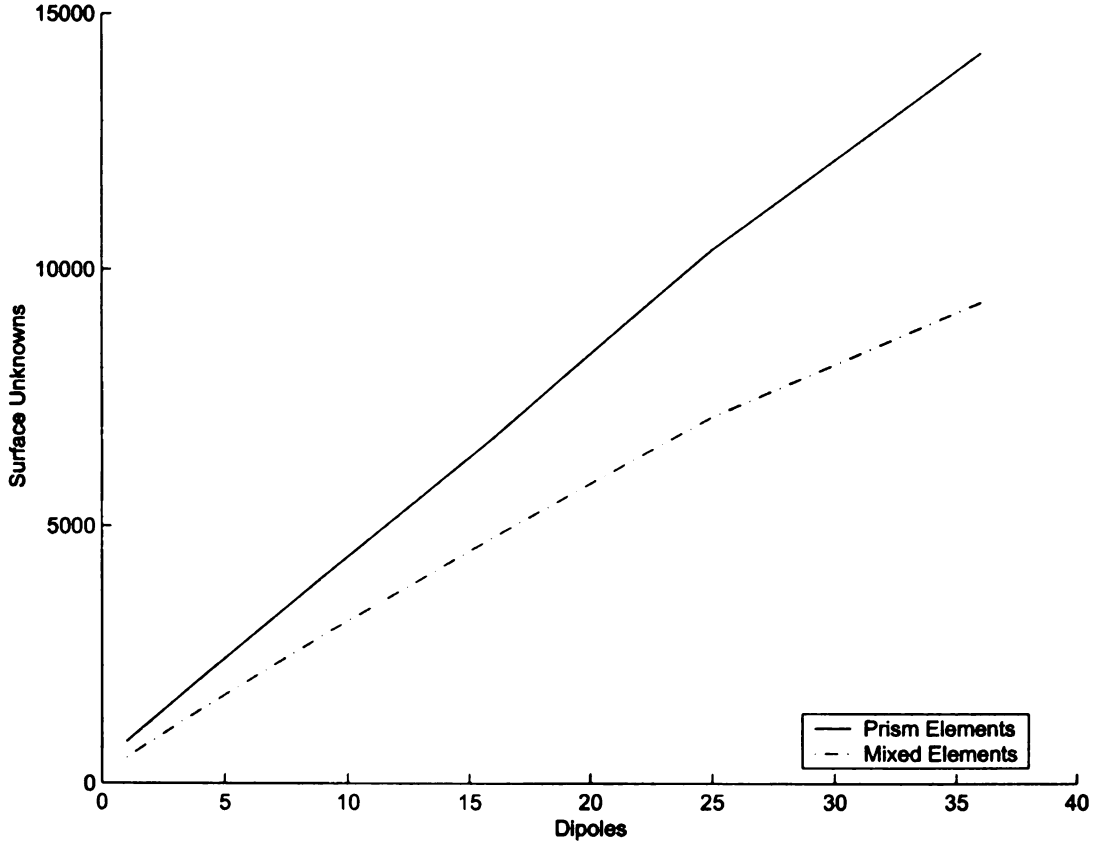

Figure 6.33. Number of surface unknowns in an i-dipole antenna array versus the number of dipoles.

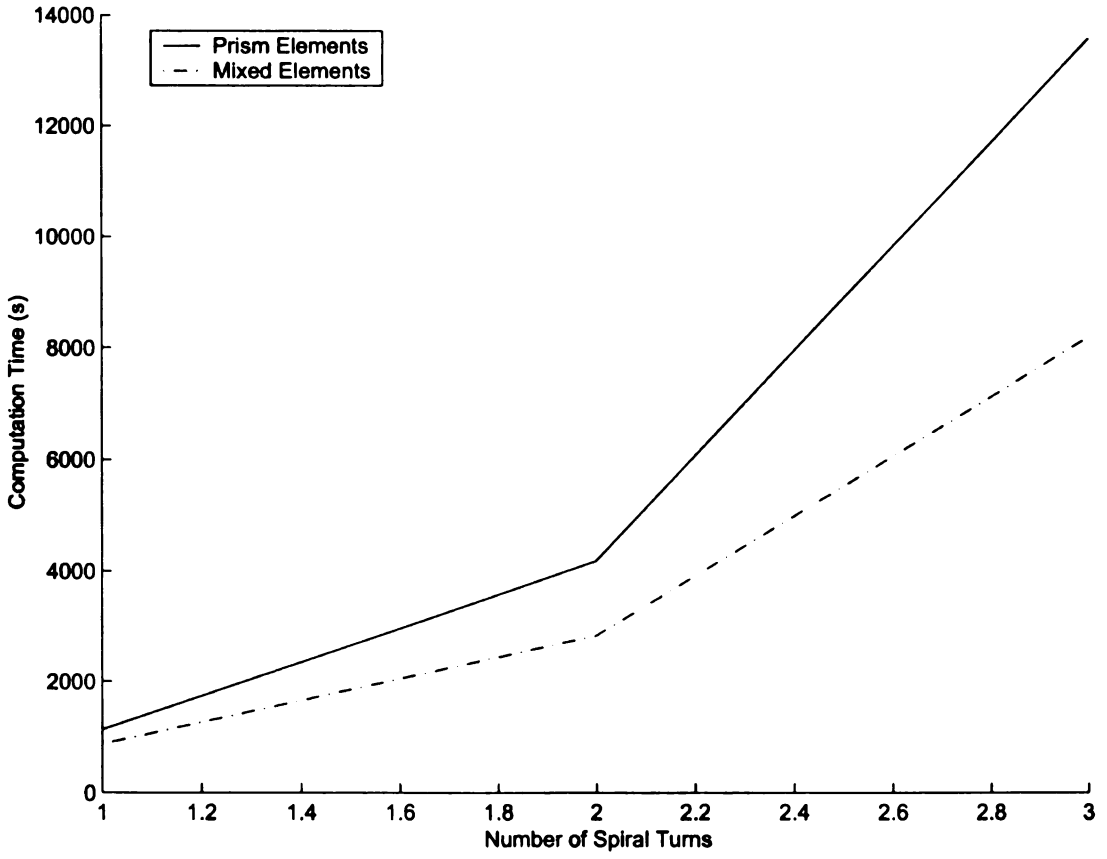

Figure 6.34. Pattern computation time for a spiral antenna versus the number of turns in the spiral.

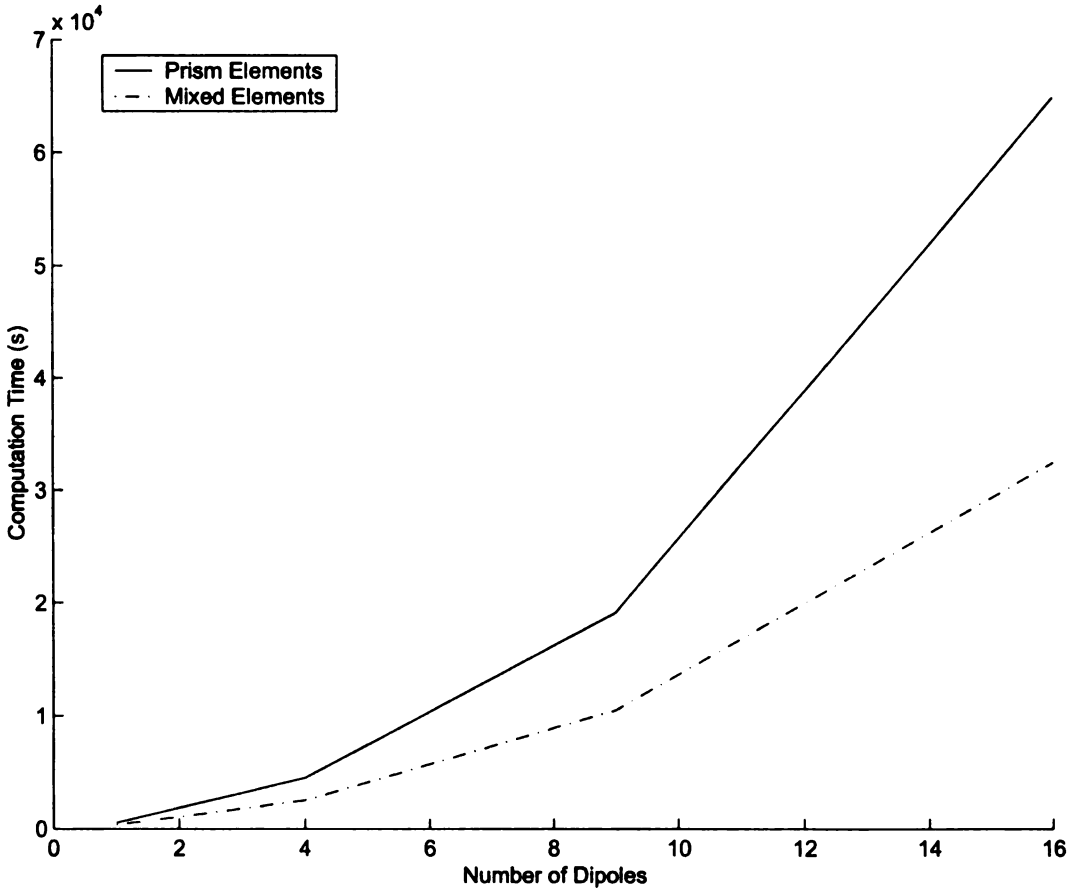

Figure 6.35. Pattern computation time for a i-dipole array antenna versus the number of dipoles.

CHAPTER 7

SOFTWARE DESIGN

This chapter presents an overview of the software design used to implement the computer program for this dissertation. Computer science techniques that are not often used in electromagnetics research based software programs were implemented to increase the efficiency and elegance of the program.

Throughout this chapter code fragments will be written in `typewriter` font to separate them from the rest of the text.

7.1 C++ vs. FORTRAN

The FORTRAN programming language has a long history in scientific and mathematical based software. At the time when computer technology advanced to a level where personal computing became readily available, and numerical simulation became feasible, FORTRAN was the only language to support complex variables as a native type. Since a high percentage of scientific problems deal with complex numbers, the choice to use FORTRAN for scientific programming was an easy one. Since that time computer technology has advanced in leaps and bounds, making it possible to model things with computers that the first computer modelers could only have dreamed about. Along with the evolution of computers, programming languages have grown from being simple, low-level mechanisms, to being a way to express software design ideas and algorithms in a very high-level and sophisticated manner. New software paradigms have been invented that allow a programmer to express ideas in software in a more abstract, yet intuitive, manner. The idea of object oriented programming (OOP), as opposed to procedure oriented programming, was introduced as a way to make computer programs more closely model real life situations. Unfortunately, FORTRAN was not designed with these types of high-level abstractions in mind. In

turn, FORTRAN programs written today look very much like programs written thirty years ago. FORTRAN has begun to address some of these issues with the standardization of FORTRAN 90 and 95 which allow the programmer to expresses concepts in a more abstract way than before, but limitations still exist. Despite this, writing scientific programs in FORTRAN 90/95 is a valid option for someone who doesn't feel like learning a whole new language but would like to express their software design ideas in a more high-level way. For more information about using FORTRAN 90/95, a good reference is [31]. To overcome the limitations of FORTRAN, however, a different programming language must be used. Although many such languages exist, one of the most popular is the C++ programming language.

The C++ programming language was developed by Dr. Bjarne Stroustrup at AT&T Bell laboratories in the early 1980's. It has since become one of the most popular programming languages in use today. This chapter does not present the details associated with the language itself, but rather focuses on using C++ to implement software programs. For the reader not familiar with C++, appendix A presents some of the basic ideas associated with the language and points the reader to numerous references that go into more detail. The C++ programming language contains various constructs that allow the programmer to make software more robust and extensible and, with it's standardization in 1998, C++ became an attractive language for scientific programming. Through the use of the C++ standard library it is easy to use sophisticated computer algorithms with a minimal amount of code. Data structures such as complex numbers, linked lists, dynamic arrays, and binary trees are all present in the library along with implementations of various fast sorting and searching algorithms.

7.1.1 Dynamic Memory Allocation

Programs, specifically scientific programs, are often created to model a general class of problems and a set of parameters is available to the user to create a more specific

problem. The memory needed by the program can vary widely based on the specific problem that is being solved. Traditionally, these types of programs have used statically allocated memory. This means that all variables, most importantly arrays, used in the program must be declared while the program is being developed. This can lead to various problems. The programmer must be able to determine the maximum possible size of these arrays without having any knowledge of the specific problem being solved, only a knowledge of the general class of the problems being modeled. If the programmer chooses these limits to be too small the program runs the risk of corrupting other parts of the program data and will probably end up crashing. If the programmer chooses an array maximum that is so large as to encompass any problem they can imagine, the amount of memory wasted for a normal program run can be significant. If the person using the program has access to the source code they can tailor the array sizes to their specific problem, but this is error prone and inconvenient. It makes the user have to worry about things that are really the programmer's responsibility.

The answer to these problems is dynamic memory allocation. Dynamic memory allocation allows the programmer to wait until runtime to determine the size of the arrays the program uses. This means the program will only use the exact amount of memory it needs to solve the problem; no more, no less. This also frees the user from having to worry about changing the source code to fit their current problem, allowing them to focus on more important things. C++ has direct support for dynamic memory allocation through the use of the `new` and `delete` operators.

FORTRAN 90/95 added support of dynamic memory allocation to the language. For this reason, dynamic memory allocation is not, by itself, a reason to choose C++ over FORTRAN. The important thing to note is that any modern program, no matter what programming language it is written in, should use dynamic memory allocation for portions of the program data that can change in size from one run of the program

to the next.

The advantages of dynamic memory allocation can be illustrated with a simple example. Consider a interactive program written to compute the average of a student's test scores. In C++, dynamic memory allocation can be used to make sure that the data array used to hold the test scores is only as big as is required without being any bigger. Consider the following program

```
#include <iostream>

int main()
{
    int number_of_scores = 0;

    std::cout << "Enter the number of scores\n";
    std::cin >> number_of_scores;

    double * student_scores = new double[number_of_scores];

    for (int i = 0; i < number_of_scores; i++)
    {
        std::cout << "Enter score:\n";
        std::cin >> student_scores[i];
    }

    double total_score = 0.0;
    for (int i = 0; i < number_of_scores; i++)
    {
        total_score += student_scores[i];
    }

    double avg_score = total_score / number_of_scores;
    std::cout << "The average score is " << avg_score << "\n";

    delete [] student_scores;

    return 0;
```

```
}
```

There are certainly flaws with this program, the most important being that there is no error checking performed, but this would only serve to detract from the important concept of the program. The main point of the program is illustrated by the dynamic memory allocation of the student score array. Notice that the memory is allocated after the user inputs the number of scores and must be deallocated, or deleted, at the end of the program. Forgetting the `delete` statement would introduce a memory leak to the program since the memory used by the student score array would never be returned to the operating system. This is one of the main criticisms of C++ dynamic memory allocation. Other programming languages use garbage collection schemes where the runtime determines whether a variable is still in use and reclaims the memory if it is not. These garbage collection schemes remove the need to manually delete dynamically allocated memory but introduce notoriously large overhead into a programming language. As a result, garbage collection was not included in the design of C++. Instead, the C++ standard template library containers provide a much better way to implement dynamic arrays of objects that relieve the programmer of the internal details of dynamic memory allocation and deallocation.

7.1.2 The Standard Template Library Containers

The discussion on dynamic memory allocation shows that it is an important part of any modern software program designed to work with variable size data. The program shown in the previous section shows that dynamic memory allocation is easy to use. The program shown is simplistic to be certain but it illustrates the point. This program does, however, heavily rely on the fact that the user know how many scores they need to enter before they can use the program. If the number of scores is large the user will find it inconvenient to have to determine the number of scores before they run the program. The program should be allowed to accept any number of

student scores without any previous knowledge of the total number of scores. An even more realistic scenario is that the student scores would be read from an external file. While the file could contain a number specifying the number of scores to read, this could be error prone if this file is updated manually by the user.

A better solution would be to allow the user to enter scores until they are finished, or simply specify the file from which the scores should be read. These scores could be added to some type of dynamic array which grows as needed. This type of dynamic array is not very difficult to implement from scratch for a C++ programmer with a grasp of classes and dynamic memory allocation. Templates could even be used to provide the array with the ability to hold any type of data, not just floating point data. Fortunately, writing a dynamic array from scratch is not necessary. The C++ standard template library (STL) provides this type of functionality, among other things, in it's container classes. Container classes allow the programmer to store and manipulate any type of data in a multitude of different ways. The most popular container is that of `std::vector`. Consider how the above student score program would be written using vectors.

```
#include <iostream>
#include <vector> // for std::vector

int main()
{
    std::vector<double> student_scores;

    while (1)
    {
        double score = 0.0;
        std::cout << "Enter score (-1 to exit):\n";
        std::cin >> score;

        if (score < 0.0)
            break;
```

```
        student_scores.push_back(score);
    }

    double total_score = 0.0;
    for (size_t i = 0; i < student_scores.size(); i++)
    {
        total_score += student_scores[i];
    }

    double avg_score = total_score / student_scores.size();
    std::cout << "The average score is " << avg_score << "\n";

    return 0;
}
```

It can be seen from the above code that the number of scores does not need to be known or stored since `std::vector` does all the necessary memory allocation and deallocation automatically. Also since `std::vector` is a template, any type of data can be stored in it, even user defined types. For example, a user defined type describing a student could be developed that held the student's name, ID number, grade point average, etc... A `std::vector` of students could then be just as easily created and manipulated as the one containing only test scores. In contrast to C++, FORTRAN 90/95 does not define this concept of a generic, dynamic array and does not even allow the programmer to define one.

7.1.3 Data Structures

The ability to group data and functionality together into a single program unit allows programmers to express real world ideas much more easily in software. Data structures play an important role in this concept. Both C++ and FORTRAN 90/95 allow the programmer to create data structures. C++ defines classes and structs while FORTRAN 90/95 defines modules. As far as C++ is concerned, classes and

structs are interchangeable so the only the concept of classes will be used in this discussion. The concepts of C++ classes and FORTRAN 90/95 modules both provide a similar mechanism that allows the programmer to group together data and the functions that operate on that data. However, there are some distinct differences. While FORTRAN 90/95 modules allow the programmer to define functions in a module, these functions are not really bound to the data declared in the module. They are simply stand-alone functions. Declaring them in a module simply allows the functions to be separately compiled and re-used in other programs. In C++, functions declared in classes are bound to the data declared in the class. These functions do not have meaning outside the context of the class in which they are declared. In the broadest sense, FORTRAN 90/95 modules simply represent an aggregation of data and functions, while C++ classes represent a wholly new data type. For this reason, FORTRAN 90/95 does not support the idea of object oriented programming, while C++ does.

7.2 Efficiency

One of the main goals of any scientific program is to make it as efficient as possible. Scientific software is usually developed for problems that are far too large to solve by hand and can even take a significant amount of time to solve with a computer. There is always a tradeoff between how general a program is and how fast it runs. The more computation done by hand before the program is developed, the faster the program will compute the results. On the other hand, computing parts of the program by hand usually leads to assumptions about the problem being solved and this limits the class of problems to which the program can provide a solution. This tradeoff is prevalent no matter what programming language is chosen to develop the software. In this case, scientists must be more creative in how they formulate the problem to be solved and the techniques they use to attack the problem. This dissertation provided a

mixed element formulation to improve the efficiency of the underlying program. This technique could be implemented in any number of programming languages with the same results. There are other parts of the program that can be made more efficient simply by choosing one language versus another.

There are numerous fundamental operations present in many real life programs regardless of what problem they are designed to solve, of which searching is one of the most common. Linear searching is the easiest algorithm to implement but it also the slowest. Hashing is one of the most complex algorithms but, when implemented properly, can be extremely fast. Hashing can be thought of as placing data into a file drawer. Using hashing the program can jump to the approximate location of the data instantly. For example, if the file drawer contained 1000 items a linear search could take use as many as 1000 comparisons to find the correct item. Hashing allows as few as one comparison, although in practice it's more than one but much less than 1000. The FORTRAN programming language does not have any direct support for searching algorithms other than linear. The C++ standard template library, on the other hand, has support for many different algorithms, including hashing.

7.2.1 Edge Creation

During mesh creation the edges of all the elements must be created. It is not possible, however, to simply loop through the elements and create all the edges on that element. Elements can share edges with other elements, so in the process of creating edges one must check if an element edge already exists as part of another element. Imagine a mesh object that holds a vector of edges. An edge is defined by its two global nodes and its global number as shown in this struct,

```
struct Edge
{
    size_t number;
    size_t node1;
    size_t node2;
```

```
};
```

The mesh object could have the following structure

```
#include <vector>

class Mesh
{
public:
    // ... Other mesh functions

    void AddEdge(const Edge& edge)
    {
        m_edge_vector.push_back(edge);
    }

    size_t FindEdge(size_t node1, size_t node2);

private:
    // ... Other mesh data

    std::vector<Edge> m_edge_vector;
};
```

The mesh object could then define the **FindEdge** function to determine if an edge already exists between two nodes in the mesh. Using linear searching this function can be defined as

```
size_t Mesh::FindEdge(size_t node1, size_t node2)
{
    for (size_t i = 0; i < m_edge_vector.size(); i++)
    {
        const Edge& edge = m_edge_vector[i];
        if ( (edge.node1 == node1 && edge.node2 == node2) ||
             (edge.node1 == node2 && edge.node2 == node1) )
```

```
        {
            return edge.number;
        }
    }
    return 0;
}
```

While this function is simple and correct, the major drawback is that as more edges are created this function will take longer to run. For problems with a large number of edges this can slow down edge creation considerably. Using a hashing algorithm can speed up this function. In FORTRAN, this functionality would have to be created from scratch, and implementing a hashing algorithm is not trivial. In C++, this functionality is already included in the way of the class **std::hash_map**. It should be noted, however, that **std::hash_map** is not officially part of the C++ standard so implementations of it can vary widely from platform to platform. To overcome this, the **std::hash_map** from STLPort can be used. To use **std::hash_map** two additional classes must be implemented. These classes are very simple and their only purpose is to implement a single function each. These types of classes are called function objects, or functors. Consider the new declaration of the **Mesh** class

```
#include <vector>
#include <hash_map>

class Mesh
{
public:
    // ... Other mesh functions

    void AddEdge(const Edge& edge)
    {
        m_edge_vector.push_back(edge);
        m_edge_map.insert(EdgePair(edge,m_edge_vector.size()));
    }
```

```
    size_t FindEdge(size_t node1, size_t node2);

private:
    struct EdgeHash
    {
        // Returns the hash key for an edge
        size_t operator()(const Edge& edge) const
        { return (edge.node1 + edge.node2); }
    };

    struct EdgeCompare
    {
        // Determines whether edge1 is the same as edge2
        bool operator()(const Edge& edge1, const Edge& edge2) const
        {
            return ( (edge1.node1 == edge2.node1 &&
                      edge1.node2 == edge2.node2) ||
                     (edge1.node1 == edge2.node2 &&
                      edge1.node2 == edge2.node1) );
        }
    };

    // Typing shortcuts
    typedef std::pair<Edge,size_t> EdgePair;
    typedef std::hash_map<Edge,size_t,EdgeHash,EdgeCompare> EdgeMap;
    typedef EdgeMap::const_iterator EdgeMapItr;

    // ... Other mesh data

    std::vector<Edge> m_edge_vector;
    EdgeMap m_edge_map;
};
```

It is seen from the above code that the functors for `std::hash_map` are declared as internal structures to the `Mesh` class. This means users of the `Mesh class` are

shielded from their implementation since they are not relevant outside of the class. The **typedef** statements simply introduce typing shortcuts and are very common when using the STL. The **AddEdge** function has also changed. Now it needs to add the edge to the edge map as well as the edge vector. The **FindEdge** function must also change.

```
size_t Mesh::FindEdge(size_t node1, size_t node2)
{
    Edge edge;
    edge.node1 = node1;
    edge.node2 = node2;

    EdgeMapItr edgeItr = m_edge_map.find(edge);
    if (edgeItr != m_edge_map.end())
        return edgeItr->second;

    return 0;
}
```

In this case, the edge map does the actual search for the edge using a hashing algorithm. This function may look a bit more confusing than the original linear search function but it is significantly faster. To illustrate the efficiency of the hashing algorithm Figure 7.1 shows the comparison of mesh creation time versus number of edges using linear searching and hashing.

7.2.2 Preassembly

One of the primary advantages of finite element methods are the sparse matrices it creates. This allows specialized iterative solution techniques that are much more efficient than direct inversion techniques. Due to the sparseness of the matrices there are only a small number of non-zero values in the matrix itself. Therefore, storing the entire matrix, including all the zero entries, is wasteful in terms of memory and computational effort. This becomes even more prevalent when modeling three

dimensional problems. To avoid this unnecessary memory and computational use techniques can be used to allow the storage and manipulation of only the non-zero matrix entries.

One popular technique is the compressed sparse row (CSR) technique. The CSR technique stores the non-zero matrix entries in a vector format and introduces one or more auxiliary data vectors that hold information about where each non-zero entry is located in the actual matrix. Using this method, the memory taken up by even a large three dimensional problem is minimal and the matrix vector multiply required at each step of an iterative solver is very fast, O(N) where N is the number of unknowns. Creating this CSR vector is accomplished by computing each non-zero entry and placing it into this vector while keeping track of where the entry is located in the full matrix. Similar to edge creation discussed in the previous section, this cannot simply be done by looping through all the elements and adding all non-zero entries to the vector. Two or more entries may reside in the same place in the full matrix. Therefore, during the computation of the non-zero entries it must be determined whether an entry already exists in the place where the current entry resides. In order to speed up this process during the actual run of the program, many finite element programs implement a "preassembly" process that predetermines the location of all non-zero matrix entries before actually computing the entry. Similar to edge creation, hashing can also speed up this process immensely as opposed to linear searching. In addition the use of data structures can make the implementation of a CSR technique straightforward. Consider the following structure

```
struct Entry
{
    size_t row;
    size_t col;
    size_t index;
    std::complex<double> value;
};
```

The **row** and **col** members of this structure hold the row and column numbers of the entry in the full matrix while the **index** holds the position of the entry in the compressed vector. A vector of these could be stored and this structure could also be used in a hashing implementation. The hashing implementation is very similar to the implementation shown for edge creation and the speed increase is substantial. Figure 7.2 shows the time it takes to preassemble the matrix versus the number of unknowns present.

One other thing to point out is that the matrix vector multiply for iterative solvers becomes very simple

```
void MatVecMult(const std::vector< std::complex<double> >& vect,
                std::vector<std::complex<double> >& prod)
{
    // Assume fe_vector is a vector of Entry's
    // computed in a different part of the program
    for (size_t i = 0; i < fe_vector.size(); i++)
    {
        const Entry& entry = fe_vector[i];
        prod[entry.row] += entry.value * vect[entry.col];
    }
}
```

Using the STL concept of iterators, which are not discussed in this dissertation, this matrix vector multiply loop can be computed very quickly, even for large problems.

7.3 Object Oriented Programming Design

As seen from the above discussion, C++ is a viable alternative to FORTRAN for scientific programming and it is not hard to write a C++ program much like a FORTRAN program using a procedure oriented approach. While this is possible, it does not take advantage of one of the most powerful features of C++, the ability to write object oriented software.

Object oriented programming (OOP) is a software design paradigm that considers objects to be the central elements of a software program. This is an intuitive idea since the world is made up of objects. Developing software to deal with cars is easier if the programming language allows a car to be described as a concrete idea with a strictly defined interface and functionality. OOP also provides a way to encapsulate implementation details so that they are shielded from users (i.e. other programmers) who don't need to know about them. OOP is not a new idea, it has been around since the mid 1960's. Programming languages like Simula and BCPL had constructs that allowed an object oriented approach and, in fact, C++ was modeled after some of these concepts. For a complete story on the design of C++ see [36].

The program written to validate the mixed element formulation developed for this dissertation was implemented using an object oriented design approach. Finite element programming is about the manipulation of finite elements which can be readily modeled as objects in a software program. Hence, object oriented programming techniques lend themselves well to finite element programming.

7.3.1 Mesh Creation

The first step in any finite element solution is to create the mesh. A finite element mesh is composed of nodes and edges, which are then grouped together to form elements. Nodes and edges have a global representation in terms of the entire mesh as well as a local representation within the confines of a finite element. A finite element acts as a map from the local to global representation. In effect, nodes and edges have both a local number and a global number. For example, a node may have a local number of 2 within a finite element but may actually be node 34 in the mesh.

Different types of elements have different numbers of nodes and edges and they are all created differently. Chapters 3 and 4 describe prism and hexahedral elements and show their corresponding representations. The way basis functions for the edges are defined on the element depends on how the local edges are created. For this

reason, one cannot simply change the way in which the local edges are numbered without changing the way the basis functions are defined for that element. The basis functions for an element also determine how the element's matrix entry is computed. Therefore, there is a tight coupling between how an element is created and how its matrix entry is computed.

Mesh creation and matrix entry computation are very often separated in a finite element software program, sometimes being implemented in completely different files. Since element creation and matrix entry computation are tightly linked, a change in one area almost always necessitates a change in the other. This tight coupling between different parts of the program can be the source of numerous hard to find errors, especially for someone who is not the original author of the code. The problem occurs because the part of the program that creates the mesh and the part of the program that computes the matrix entries are too tightly coupled with the details of the elements they are using. It would be beneficial to hide these implementation details from parts of the program that don't require them. For example, instead of hard coding the local nodes used for the local edge of an element in the mesh creation routines, the mesh creation routines should ask the element for this information. This type of encapsulation can be performed using objects. To illustrate, imagine a object representing a finite element modeled by the following class.

```
#include <vector>

class Element
{
public:
    Element(size_t nodes, size_t edges)
    {
        m_nodes.resize(nodes);
        m_edges.resize(edges);
    }
```

```
    virtual ~Element()
    { }

    size_t GetNumberOfNodes() const
    { return m_nodes.size(); }

    size_t GetNumberOfEdges() const
    { return m_edges.size(); }

    size_t GetNode(size_t index) const
    { return m_nodes[index - 1]; }

    size_t GetEdge(size_t index) const
    { return m_edges[index - 1]; }

    void SetNode(size_t index, size_t node)
    { m_nodes[index - 1] = node; }

    void SetEdge(size_t index, size_t edge)
    { m_edges[index - 1] = edge; }

    virtual void GetEdgeNodes(size_t edge,
                              size_t& node1,
                              size_t& node2) const = 0;

private:
    std::vector<size_t> m_nodes;  // local to global node map
    std::vector<size_t> m_edges;  // local to global edge map
};
```

This code is taken nearly verbatim from the program used for this dissertation. The **GetEdgeNodes** functions is declared as a pure virtual function which means that any class that derives from this class *must* implement this function. This makes **Element** an abstract base class (ABC) since objects of type element cannot be directly created. The idea is to derive a class from **Element** and implement the **GetEdgeNodes** functionality in that class. For example, to implement a class representing prism elements

and a class representing hexahedral elements the following code is used.

```
#include "Element.h" // Assuming Element is defined here

class Prism : public Element
{
public:
    Prism()
        : Element(6, 9)
    {
    }

    void GetEdgeNodes(size_t edge,
                      size_t& node1,
                      size_t& node2) const
    {
        // Return the two nodes needed for the local edge
    }
};

class Hexahedral : public Element
{
public:
    Hexahedral()
        : Element(8, 12)
    {
    }

    void GetEdgeNodes(size_t edge,
                      size_t& node1,
                      size_t& node2) const
    {
        // Return the two nodes needed for the local edge
    }
};
```

The implementation of the **GetEdgeNodes** function for prisms and hexahedrals is not explicitly shown since there are many ways to implement this function. The

implementation must tell the caller which two local nodes correspond to a certain local edge. Designing an element class in this fashion frees mesh creation routines from having to know the details of how a element edges are created.

7.3.2 Matrix Entry Calculation

The discussion in the previous section shows how mesh creation routines can be isolated from the details of finite element representation. At this point it is only natural to wonder if matrix entry calculations could be treated in a similar manner. This is indeed possible and leads to an elegant design that is both robust and extensible. Adding one additional function to the `Element` class, matrix entry calculation can be nicely encapsulated in the element objects.

```
#include <vector>
#include <complex>

class Element
{
public:
    // ...
    // Same as before
    // ....

    virtual std::complex<double> GetFeEntry(size_t test_edge
                                            size_t source_edge)
                                            const = 0;

private:
    std::vector<size_t> m_nodes;  // local to global node map
    std::vector<size_t> m_edges;  // local to global edge map
};
```

With the addition of the `GetFeEntry` function the element is now responsible for its own edge creation and matrix entry computation. Other parts of the program do not need to know anything about the internal details of the element. The fact

that `GetFeEntry` is a virtual function provides the ability to implement polymorphic behavior for objects inherited from `Element`. Consider how easily the FE matrix entries could now be computed for all the elements in the mesh

```
#include <vector>
#include <complex>
#include "Element.h"

// Assume that the elements are created and that
// both prism and hex elements exist in the elements
// array.
void BuildFEMatrix(std::vector<Element*>& elements)
{
    for (size_t i = 0; i < elements.size(); i++)
    {
        Element * element = elements[i];
        size_t edges = element->GetNumberOfEdges();
        for (size_t j = 1; j <= edges; j++)
        {
            for (size_t k = 1; k <= edges; k++)
            {
                std::complex<double> entry
                     = element->GetFeEntry(j,k);

                // Add this entry into the FE matrix
            }
        }
    }
}
```

The use of a solid object oriented design makes this function very small yet extremely robust. Imagine the situation where a new element was introduced into the mesh. Assuming this element was derived from class Element and implemented the needed functionality the code the `BuildFEMatrix` would remain *unchanged.* This concept of "old code calling new code" is a cornerstone of object oriented design.

7.3.3 Basis Function Integration

While the encapsulation of element edge creation and element matrix entry computation discussed in the previous sections is a powerful technique, most times these alone are not enough. Oftentimes the need arises to integrate a basis function over the extent of the finite element. At the time of the original design of the program, however, the programmer may not know all possible integrands that could eventually be used in a basis function integral. Once again polymorphism can be used to provide a simple yet elegant solution. Consider the abstract base class defined by

```
#include <complex>

class Integrand
{
public:
    virtual std::complex<double> Eval(double x,
                                      double y,
                                      double z) = 0;
};
```

This class does absolutely nothing. It has no data members and does not define any functionality, it simply declares one pure virtual function. It's importance is seen when another function is added to the `Element` class.

```
#include <vector>
#include <complex>

class Element
{
public:
    // ...
    // Same as before
    // ....

    virtual std::complex<double> Integrate(size_t edge,
```

```
                                        Integrand& ignd)
                                        const = 0;

private:
    std::vector<size_t> m_nodes;  // local to global node map
    std::vector<size_t> m_edges;  // local to global edge map
};
```

Notice the last argument in the new **Integrate** function. It is a reference to an **Integrand** object. Through polymorphism, any object derived from **Integrand** could be used in this argument. A programmer can therefore define any integrand they wish and use it in this function. In addition, since **Integrate** is a virtual function it is also available to implement polymorphic behavior. In effect, polymorphism is used both in the call to the function itself, and in one of the function's arguments. Designs such as this lead to very extensible and reusable code.

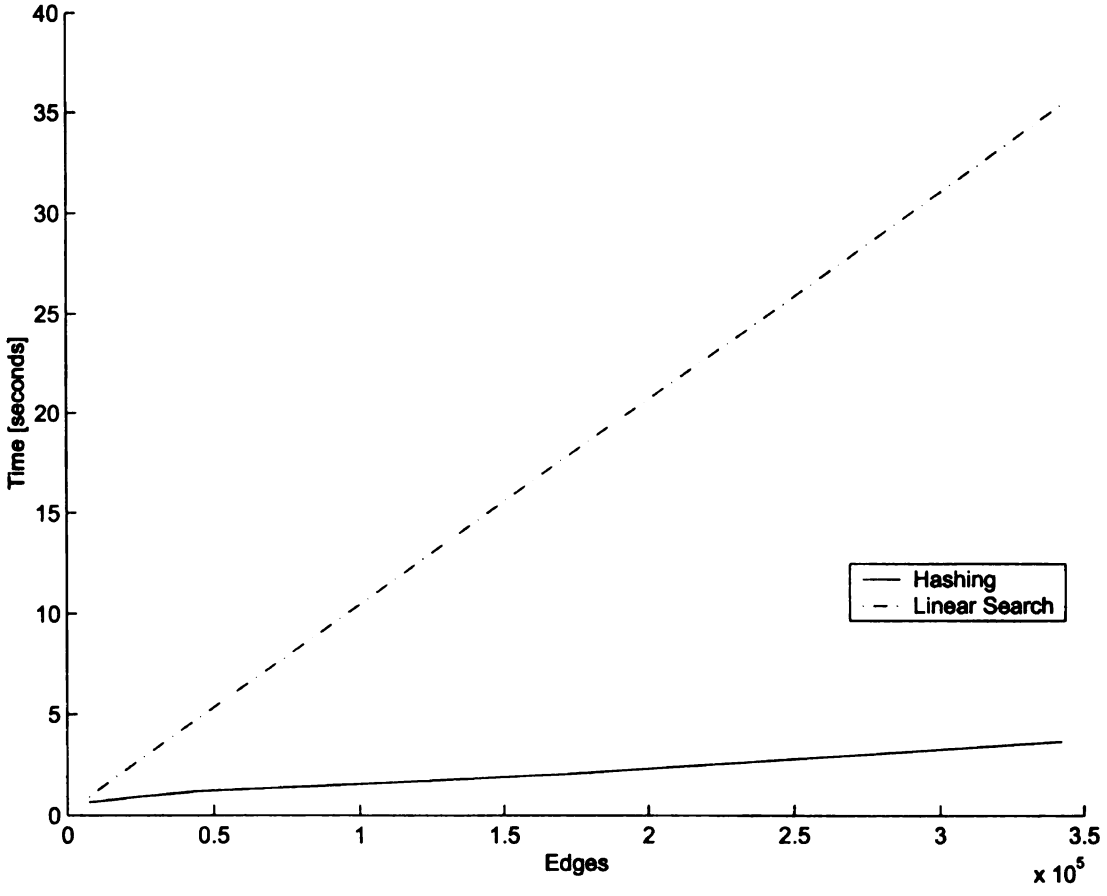

Figure 7.1. Comparison of edge creation using hashing and linear searching algorithms

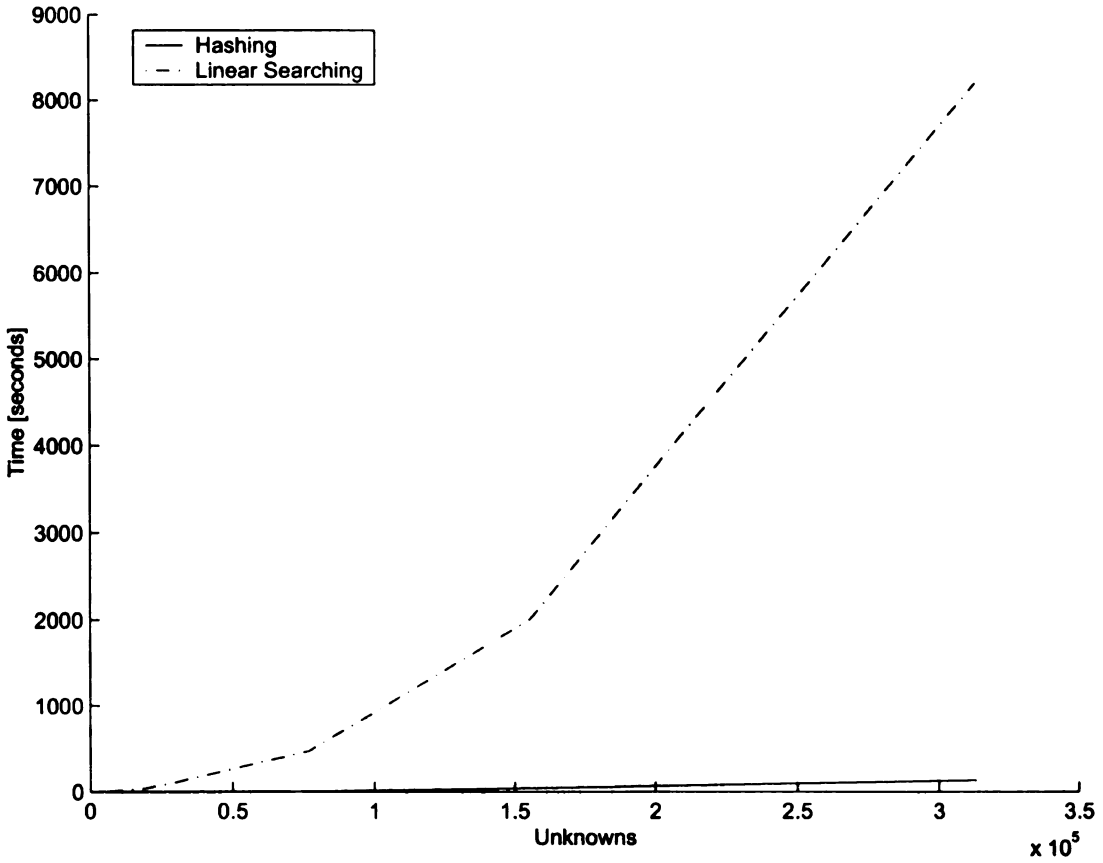

Figure 7.2. Comparison of preassembly using hashing and linear searching algorithms

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

A mixed element formulation was developed to model three-dimensional planar cavity backed aperture antennas. This formulation was shown to be equivalent to a formulation using only prism elements but was more efficient in terms of memory and computational demand. The main memory and computational savings were due to the fact that the mixed element formulation produced fewer surface unknowns than the prism element formulation for the same geometry.

The prism element formulation was presented first, followed by the hexahedral formulation. The two formulations were then combined and simulations were shown for a few simple test cases. These results were shown to be in good agreement with the results computed for a single element formulation.

Next, a couple of challenging complex geometries were studied using the mixed element formulation. These geometries were described as geometrically constrained as opposed to electrically constrained. The mixed element formulation was shown to be more efficient for these geometries and scaled better as the geometrical dimensions were increased. Slight discrepancies between the mixed element and prism element formulations were seen in the modeling of impedance and pattern simulations for these complex geometries. These discrepancies were caused by dissimilar mesh densities surrounding the feed points causing different input powers to be produced at the feeds. These discrepancies were not seen in radar cross section simulations as RCS measurements do not include a feed model.

Finally the software design of the program was discussed. It was shown that the use of object oriented programming along with the C++ programming language can make software more robust and extensible as opposed to a procedural approach using

a language such as FORTRAN. Also, well known computer algorithms were applied to certain parts of the program that resulted in much more efficient implementations.

The mixed element formulation provides a good foundation on which to develop tools to model highly complex geometries in a more efficient manner than using a single element. Future work on this idea might be to include the ability to use tetrahedral elements which allow any arbitrary complex three-dimensional surface to be modelled. Future directions for the idea might be to have a mixture of elements in depth rather than across the surface.

APPENDICES

APPENDIX A

OVERVIEW OF THE C++ PROGRAMMING LANGUAGE

This appendix presents a brief overview of the C++ programming language but by no means does it aim to discuss all the details associated with the C++ language. Rather, this overview is presented with the goal of providing the reader with the necessary background needed to understand the software design considerations discussed in chapter 7. For the reader interested in a more thorough treatment, there are literally hundreds of books available on C++. The most popular reference texts in terms of introductory C++ are [32]-[35]. References will be given to books dealing with more specific topics as those topics are presented.

Source code fragments shown in this appendix will occur in `typewriter` font to distinguish them from the rest of the text.

A.1 History

The C++ programming language was originally designed as an extension to the C programming language by Bjarne Stroustrup at AT&T Bell Labs in the mid 1980's. The decision to make C++ backward compatible with C is cause for great debate in the C++ community. On one hand, compatibility with C allowed C++ to spread rapidly through the programming community since there was little a programmer needed to learn to start coding in C++. On the other hand, compatibility with C hampered users of C++ due to the same reasons that made it easy to learn. For many people C++ was just a "better C" and the most powerful features of C++ were never used.

The reality is that C++ is a completely different language than C in all ways, excluding syntax. To write effective C++ programs a different design paradigm must be used. The flexibility of C++ is both it's strength and weakness. In his book on

the design and evolution of C++ [36], Dr. Stroustrup describes how one of his main goals when designing C++ was to not force the user to have to do things in a certain way. This allows people to express the same idea in many different ways but at the same time it can also cause wide variations in the way a program is written. It also allows previous users of C to write C++ programs much like they used to write C programs, thereby eliminating the features that make C++ so powerful.

This appendix does not assume any knowledge of C++ although familiarity with programming in general is assumed. It will not discuss things such as compiling, linking, and running programs as it is assumed the reader is familiar with these ideas.

Readers who are familiar with procedural languages such as C and FORTRAN may find the concepts of user defined data types and templates a bit confusing at first; however, once these ideas understood it becomes easier to develop software programs that consist of a set of small, separate, well-defined pieces. Programming in this fashion makes software much easier to understand and maintain.

A.2 Hello World

One of the most famous ways to introduce a user to a new language was shown by Brian Kernighan and Dennis Ritchie in their book "The C Programming Language", and is called "Hello World" program. This program simply outputs the phrase "Hello World" to the screen. In C++, objects are viewed as the central piece of software in a program, so the way to create the "Hello World" program in C++ would be to create an object representing the world and let it say speak for itself. In C++ terms that would be expressed by the following program.

```
#include <iostream>

class World
{
public:
```

```
    World()  { std::cout << "Hello\n"; }
    ~World() { std::cout << "Good-bye\n"; }

};

// Create the world
World theWorld;

int main()
{
    return 0;
}
```

Readers unfamiliar with C++ will most likely be confused by this program. To the C programmer it looks as though this program does nothing and to the FORTRAN programmer the syntax is probably so foreign that it's impossible to know where to start. The program above is shown to illustrate that the way in which a C++ program is written can look quite different than a program written in C or FORTRAN. Successfully compiling and running this program results in the words "Hello" and "Good-bye" to be printed to the screen. The goal of this appendix to teach the non C++ programmer how and why this program works, so for the reader who is thoroughly confounded by the above program, be patient and read on, it will all become clear. But before this program can be understood the basic syntax of the C++ language must be introduced and the concept of a user defined data type must be presented.

A.3 Basic Syntax

Programming languages are defined by their syntax. A language defines a set of constructs, usually in the form of keywords, that allow compilers to translate the high level language into machine code that a computer can execute. This section presents the different constructs that C++ defines. Only basic code concepts associated with

the C++ language will be discussed here. Readers familiar with C should be able to skim this section quickly as there is nothing new to be presented with the exception of references which are not available in the C language (at least not yet). Readers familiar with other languages may use this section to quickly learn the basic variable types and expressions that C++ defines.

A.3.1 Types and Expressions

All programming languages predefine a set of types and expressions that can be used to write software programs. Types allow programmers to model a variety of numeric and non-numeric data. Expressions define operations that work on these predefined types. C++ defines the following types:

- `bool` - Boolean Type. Takes the value **`true`** or **`false`**.
- `int` - Represents an integer type.
- `char` - Represents a character.
- `float` - Represents a single precision floating point variable.
- `double` - Represents a double precision floating point variable.

Integral types (`int` and `char`) may be prefixed with the word **`signed`** or **`unsigned`** which determines whether the variable can hold negative numbers. The default for integers is to have signed values while the default for **`char`** is platform dependent. In addition the word **`long`** may also be prefixed in front of all types except **`char`** and **`float`** effectively doubling the machine size of the type.

There are a large number of expressions and operations defined by the C++ language. For this appendix only a certain subset of these need to be examined. For a complete listing of all expressions and operators defined by the C++ language see [32]. The most common operations are:

+, -, *, /, =, ==, !, %, &, |, &&, ||, <<, >>

The first four represent the standard binary mathematical operations of addition, subtraction, multiplication and division. The = operator is used for assignment. The == operator is used for equality testing, ! is the negation operator, % is the modulus operator, & is a bitwise AND operation, | is a bitwise OR operation, && is a logical AND operation, || is a logical OR operation, << is the left shift operator, and >> is the right shift operator. The binary operators such as + and - also define assignment overloads so the operator += is defined as an operator that adds and assigns to the variable it is working on. For example

```
int i = 5; // Declare a variable i and set its value to 5
i += 4;    // i now equals 9, same as i = i + 4
```

The statements in the above code fragment end the way all statements in C++ do, with a semicolon. This means that statements can extend over more than a single line without needing any special characters and, conversely, more than one statement can be contained on a single line.

The portion of code located after the // characters is called a comment. The // characters tell the compiler that anything from that point forward until the end of the line should be ignored. Comments are a useful way to let other programmers know the intent of a statement in a program if it is not clear from the context of the statement. In this case the comments are superfluous and are only included to introduce their syntax. Normally these type of comments should be avoided as they only serve to clutter up the code without presenting any relevant information.

Other operators are defined in an expected way such as (<, >) for less than and greater than and so on.

A.3.2 Functions

Programming languages almost always give the programmer a way to group a set of related commands and operations into a functional unit. These functional units have

a variety of names such as procedures, modules, subroutines, and interfaces. In C++ they are called functions. In C++ there is more than one type of function. Functions can be classified as free functions or member functions. Free functions are functions that are available to any translation unit in which their declaration is visible. Member functions are functions that are bound to classes and structs. Only free functions are important at this point in the discussion. Member functions will be introduced later when the concept of a user defined type is introduced. In C++, a function (free or member) must be declared before it can be called. To declare a function the programmer must tell the compiler what type the function input arguments will be and the type of it's return value, if any. For example

```
// Declare a function f taking an integer and returning nothing
//
// void is a C++ keyword that acts as a placeholder for
// something that does not have a type.
void f(int i);

// Declare a function g taking nothing and returning an integer
//
// It is not necessary to include the void keyword for a
// function that takes no input arguments.
int g();
```

It is seen that these function declarations end in a semicolon just like any other C++ statement. Note that declaring a function only specifies it's interface, not the function itself. The function must also be defined. The function `f` can be defined by

```
// Defines the function f
void f(int i)
{
    i += 5;
} // Now there is no semicolon
```

The curly braces ({,}) are used to define the scope of the function. In this example, the result is the number 5 being added to the variable `i`. When called, the function starts executing commands directly after the left brace and continues until it hits the right brace at which point the program returns to the point from which the function was invoked. The scope of the function determines how variables are treated. If a variable is declared within the function braces it is only visible to that function and it is called a local variable.

One important note about functions is that they may be declared multiple times in a program, but they can only be defined once.

A.3.3 Arrays

Many times a programmer needs to create more than one of the same type of variable but needs to be able to access these separate variables as a single entity. In this regard programming languages allow the ability to create arrays of variables. In C++ arrays of variables are easy to create. For example, consider the following

```
// Declare an array of 100 integers
int ia[100];

// Declare an array of 50 doubles
double da[50];
```

As can be seen above, C++ uses a slightly different syntax to declare an array by using the square brackets `[]` as opposed to the usual parentheses. Accessing the elements of the array are done using the same operators, however it is important to note that in C++ the first element of an array starts at zero and continues to n-1 where n is the number of elements in the array. This means that `ia[0]` is the first element of the ia array and `ia[99]` is the last element. This is different from many other programming languages where indexing starts at one. This indexing trips up a lot of programmers when they first start using C++, particularly scientific programmers with a background in FORTRAN, so it is important to keep this in mind.

A.3.4 Pointers and References

Pointers are the whipping boys of C++, everybody loves to hate them. They've received this reputation because they can lead to subtle errors in programs that are hard to diagnose. They also introduce some syntactical hurdles to the language that many people find confusing and unnecessary. When used properly, however, pointers can be very useful so understanding how they work is an important aspect of C++ programming.

Simply put, pointers are addresses. In other words they point to an address of a variable rather than being a variable themselves[1] In C++ pointers are used for a multitude of reasons, one of them is to provide the ability to return more than one argument from a function. To explain this, a slight digression about how function calls behave in C++ is required.

Function arguments in C++ are "passed by value", which is just a fancy term that means a copy of the argument is passed to a function. In other words, when an variable is passed to a function only a copy of the variable is given to the function. For example, consider the code

```
void f(int i)
{
    i += 5;
}

void g()
{
    int i = 4;
    f(i);  // i is still 4
}
```

Even though the function `f` adds 5 to the value of its argument it only receives a

[1] In actuality pointers are also variables in the respect that they take up memory in the computer, although the value of the pointer itself is rarely useful. Instead it is what the pointer points to that is important.

copy of the `i` variable from the function **g**. Therefore the `i` variable *in the function* **g** remains unchanged.

In order to have the function `f` change the actual value of `i`, a pointer to `i` must be passed to the function `f`. Passing a pointer to i to the function `f` simply tells `f` where to find the `i` variable. Once `f` knows where to find the `i` variable declared in the function **g**, it can change its value. Here's how that would be accomplished in C++ terms.

```
void f(int * pi)
{
    *pi += 5;
}

void g()
{
    int i = 4;
    f(&i);  // i is now 9
}
```

Syntactically there is a bit of a difference between passing a normal variable and passing a pointer to a variable. First, the declaration of `f` has changed. There is now a (*) between the variable type and the name of the variable. Here the asterisk is not acting as the multiplication operator but rather it states that the argument for the function `f` holds an *pointer* to a variable, and the name of the pointer is `pi` (pi = pointer-to-i, not the irrational number π). Inside the function `f` is a strange looking statement. In this case the (*) operator in front of `pi` takes on yet another meaning, different from either it has had before. It is used to *dereference* the pointer. Dereferencing a pointer means retrieving the value in the memory location indicated by the pointer value. Since `pi` is a pointer to `int`, the process of dereferencing the pointer will result in a normal `int` variable. In this case it means the the pointer is used to retrieve the actual value referenced by the pointer, here the variable `i` from

the `g` function, then five is added to this value.

The way the variable `i` in `g` is passed to `f` also has to change. In order to pass a pointer to `i` the (&) operator must be used. In this context the (&) operator says that the address of `i` is passed to the function `f`, hence it is called the "address-of" operator. This is in contrast to the meaning of the (&) operator described above as the bitwise AND operator.

C++ is famous (or infamous) for having operators mean different things in different contexts. In this example alone the (*) operator has two separate meanings, in addition to its use as the multiplication operator, and the (&) operator has a different meaning from the bitwise AND operator. In fact, the (&) operator has yet another meaning that will be presented when references are introduced. These multiple duty operators can be confusing to the programmer just starting out in C++ and are often misunderstood and misused. Learning how to use these operators in the correct way is one of the main keys to learning how to use pointers.

To get back to why pointers were originally introduced, as a way to return multiple values from a function, consider the following code where a function is needed to convert a point in polar coordinates to a point in Cartesian coordinates.

```
#include <cmath>  // For sin and cos

void PolToCart(double r, double phi, double * px, double * py)
{
    *px = r * cos(phi);
    *py = r * sin(phi);
}

void g()
{
    double r = 2.0, phi = 0.707;
    double x = 0, y = 0;
    PolToCart(r, phi, &x, &y);
}
```

In real C++ code this type of conversion would probably be wrapped in an object but for the sake of this discussion on pointers the above code should suffice. Imagine the case where pointers could not be used. Two separate functions would have to be created and called, one to compute the x-coordinate and one to compute the y-coordinate. Using pointers these two functions can be combined into a single, more intuitive, function.

In C, pointers were the only way to provide a way to return more than one variable from a function. C++ introduced another way to pass the actual value of a variable to a function, called references. In short, a reference is just an another name for an *existing* variable. While null pointers can exist (e.g. `int * pi = 0`), there is no such thing as a null reference[2]

Through the use of references, C++ allows a programmer to pass a variable to a function "by reference". This means the actual value of the the variable is passed to the function rather than a copy of the variable. The way to declare a reference is very similar to the way a pointer is declared. Instead of using the (*) operator the (&) operator is used. For example to declare a reference called `i` to a variable called `j` the following code is used.

```
int f()
{
    int j = 4;
    int& i = j; // Declare i to be a reference to j
}
```

An important thing to remember about references is that when the value of a reference is changed what it refers to also changes. Consider the following code.

```
int f()
{
```

[2] There are ways to fake null references but why anyone would want to do such a thing is beyond comprehension.

```
    int j = 4;
    int& i = j; // Declare i to be a reference to j
    i += 5; // i = 9 and so does j
}
```

Because `i` is just another name for `j`, any operation performed on `i` is also performed on `j`. In this case the fact that `j` also changes is easy to see but in cases where a reference is declared a significant distance away from where it is changed it is easy to forget that `i` is actually a reference and not just a normal variable.

The way to pass a variable to a function by reference is no different than passing the variable by value as far as the caller is concerned. Only the definition of the function itself must change. Consider how the **PolToCart** function would be re-written using references.

```
#include <cmath>  // For sin and cos

void PolToCart(double r, double phi, double & x, double & y)
{
    x = r * cos(phi);
    y = r * sin(phi);
}

void g()
{
    double r = 2.0, phi = 0.707;
    double x = 0, y = 0;
    PolToCart(r, phi, x, y);
}
```

Here the call to `PolToCart` in function `g` looks the same as it would look if `x` and `y` were passed by value. The difference lies in how `x` and `y` are declared in the function **PolToCart**. The (&) operator in the definitions of `x` and `y` determine that `x` and `y` are passed by reference instead of by value like `r` and `phi`.

Some programmers dislike using references because the call to `PolToCart` looks exactly the same whether `x` and `y` are passed by value or passed by reference. In contrast, the call to the `PolToCart` function that uses pointers specifies more clearly that the actual values `x` x and `y` may change during the course of the function call. To help alleviate these concerns, a function that requires references to arguments whose values will not change during the course of the function should declare them as `const` references. By simply putting a `const` in front of the variable type (e.g. `const double & x`) the caller of the function is assured that the value of `x` will not be changed during the course of the function. At this point the reader may see no difference between passing a variable by constant reference and passing a variable by value and for the case of built in types this is indeed the case. Passing types by constant reference becomes more important when user defined types are introduced.

For the case of the `PolToCart` function the decision to use pointers or references comes down to a matter of personal taste, while in other cases one may be preferable to the other.

A.3.5 Loops

Loops are involved in all but the simplest programs. Loops allow a set of statements to be repeated a set number of times in a straightforward manner. C++ offers three types of looping constructs, the `for` loop, the `while` loop, and the `do-while` loop. The `for` loop consists of three statements separated by semicolons

1. Where to start the loop,
2. Where to end the loop,
3. An operation that should be computed at each step of the loop.

For example, consider the code to fill an array with a set of numbers.

```
int ia[100];
```

```
for (int i = 0; i < 100; i++)
{
    ia[i] = i;
}
```

The first statement of the `for` loop states where the loop should start, in this case at `i = 0`. The second statement is checked at the end of each iteration of the loop. If the statement is true the loop continues, if it is false the loop ends. In this case the loop should continue while `i` $<$ 100. The third statement provides an operation that should be performed at the end of each iteration of the loop. In this case that means the variable `i` should be incremented by one at the end of each iteration. The operator (++) is simply shorthand that means one should be added to the corresponding variable.

The loop structures **`while`** and **`do-while`** are very similar to each other. They both continue executing as long as a certain condition is met. There is one major difference between the two loop structures. In a **`do-while`** loop the statements inside the loop are executed at least one time, while in the case of a **`while`** loop the statements inside the loop may not get executed at all. Consider the following code

```
int i = 100;
while (i < 100) // Only execute the loop statements if i < 100
{
    // Statements to execute
}

do
{
    // Execute these statements at least once
} while (i < 100); // Continue only if i < 100
```

In the first case the statements inside **`while`** loop will never be executed because the value of `i` is not less than 100. In the second case the statements inside the **`do-while`**

loop will be executed at least once since the termination condition is not tested until the loop has executed for the first time.

A.3.6 Conditionals

Making decisions is an important part of program design. A software program must be able to respond effectively to different types of input in the correct way. Programming languages therefore give the programmer conditional statements. C++ defines two types of conditional statements, the **if-elseif-else** statements, and the **switch-case** statements. Both of these types are effective ways to make decisions during the course of a program and each has its advantages and disadvantages. The most used type is the **if-elseif-else** statement. This statement makes it easy to provide a way to correctly respond to various program states. The most general way to use this type of conditional statement is shown in the following code

```
#include <iostream>

void f()
{
    int number;
    std::cout << "Please enter a number between 0 and 2: ";
    std::cin >> number;

    if (number == 0)
    {
        std::cout << "You entered 0\n";
    }
    else if (number == 1)
    {
        std::cout << "You entered 1\n";
    }
    else if (number == 2)
    {
        std::cout << "You entered 2\n";
    }
```

```
    else
    {
        std::cout << "You entered an invalid number\n";
    }
}
```

This simple function prompts the user to input a number and using an **if-elseif-else** lets the user know what number they entered. Admittedly this program is a bit convoluted to say the least, but the objective is to show an example of an **if-elseif-else** conditional statement. The **std::cin** object is declared in the **iostream** file and is used to input data from the user.

The other type of conditional statement in C++ is the **switch-case** statement. This statement is useful for times when there are a large number of possible values to choose from rather than using a long list of **if-elseif** statements. The function shown above can be written using a **switch-case** statement.

```
#include <iostream>

void f()
{
    int number;
    std::cout << "Please enter a number between 0 and 2: ";
    std::cin >> number;

    switch (number)
    {
    case 0:
        std::cout << "You entered 0\n";
        break;
    case 1:
        std::cout << "You entered 1\n";
        break;
    case 2:
        std::cout << "You entered 2\n";
        break;
```

```
        default:
            std::cout << "You entered an invalid number\n";
            break;
        }
    }
```

In this example the user is again prompted for a number but the decision making is done by a **switch-case** statement. A couple things need to be pointed out however. The **break** statements are used by a **switch-case** statement to break out of the statement. If these were not included each **case** statement would be evaluated as true and the program would print out the statements associated with *every* **case** label as well as the **default** label. The **default** label is there in the situation that none of the **case** labels are matched. This gives the programmer the ability to define some sort of default processing for this situation.

A.3.7 Dynamic Memory Allocation

There are two ways a programmer can declare a variable in C++, statically and dynamically. These two different variable declarations determine where in memory the variable will be held. There are two types of memory a computer program can use, one is referred to as the stack, while the other is referred to as the heap. Variables declared on the stack become part of the programs internal memory. Variables declared on the heap are allocated during the execution of the program.

Up to this point all variables in the code examples have been declared on the stack. In C++ the way to declare a variable on the heap is through the use of the **new** operator. Consider the following code which shows the difference between declaring a variable statically and declaring a variable dynamically

```
void f()
{
    int i;              // Allocated on the stack
    int * j = new int; // Allocated on the heap
```

```
    delete j;           // new must always be matched with
                        // delete
}
```

As seen in the example memory allocated with **new** is in the form of a pointer. Another important thing to note is the use of the **delete** operator. In C++ any memory allocated with **new** must be deallocated with **delete**. In this case if **delete** is not called on the variable j the memory j uses will not be returned to the operating system when the function f terminates. There is no way for the system to know whether this memory is still being used so it assumes that it is and this memory is effectively lost. This is called a memory leak. With personal computers containing 256MB - 1GB of memory these days the loss of an integer variable (usually 4 bytes) may not seem all that bad. Imagine, however, if the program this function resides in runs for weeks at a time and the function is called millions of times. The lost memory adds up and eventually the computer will run out of memory to give to the function and the program will crash.

In the example shown above there is really no difference between declaring the variable on the heap as opposed to the stack with the exception of the burdensome call to delete. Where dynamically memory allocation can be important is in terms of arrays.

Sometimes it is not known how big an array should be when the program is written. The age old solution to this problem is to statically allocate an array with a size large enough to hold the maximum anticipated value. This solution has two major drawbacks that stem from the fact that choosing this maximum value can be difficult. If the size chosen is too small the program runs the risk of trying to access elements of the array that do not exist (e.g. the 101st element of a 100 element array). This can cause any number of problems, from overwriting data that happens to be in a memory location contiguous to the array, to causing the program to crash all together. On

the other hand, if the size of the array is chosen to be way too big, massive amounts of memory will be wasted. The better solution to this problem is defer the sizing of the array until the actual size is known. Allocating and de-allocating an array dynamically in C++ requires a slightly different syntax than dynamically allocating a normal variable. Consider the following code used to dynamically allocate an array of 100 integers

```
void f()
{
    int * ia = new int[100];
    delete [] ia;
}
```

From the above example it is clear that to declare an array dynamically the **new** operator must be combined with the ([]) operators. In addition, deleting the array must be done by combining the **delete** operator with the ([]) operators. In this case dynamic memory allocation doesn't seem to have helped that much, the **ia** could just have easily been declared statically. There is one important difference, shown in the following code.

```
void f()
{
    int array_size = 100;

    // *Error* - array_size must be constant
    int ia1[array_size];

    // OK - array_size can be a variable
    int * ia2 = new int[array_size];

    delete [] ia2;
}
```

Even though it is easy to see that the **array_size** variable has a size of 100 and therefore the size of **ia1** should be 100, the compiler cannot make this distinction.

The size of a statically allocated array must be a constant expression. Prefixing the `const` keyword to the `array_size` declaration (e.g. `const int array_size = 100;`) will allow this code to compile, although this is really no different than putting a 100 inside the brackets of the `ia1` declaration.

In this case the `array_size` variable is a fixed size but one can easily imagine that `array_size` could be a variable input by the user or read from some input file and in that sense it would be truly variable. In that case dynamic memory allocation is the *only* way to correctly size the array.

A.4 User Defined Types

The most fundamental concept in C++ is that of an object. An object can represent a physical entity of some kind such as a car or an animal, but it can also represent a more abstract idea such as a vehicle or a species. C++ gives the programmer the ability to model objects in a way that is extremely intuitive by allowing them to be introduced as a new type to the language. These new types can be used in a very similar manner to the types predefined by the language. These user defined types provide the basis for object oriented programming.

A.4.1 Classes

A class is the way in which C++ allows programmers to define a new type for the language. This type can be anything the programmer desires but is usually in the form of some countable object. The concept of creating a class can be introduced by recalling the "Hello World" program introduced at the beginning of this appendix.

```
#include <iostream>

class World
{
public:
    World()  { std::cout << "Hello\n"; }
```

```
    ~World() { std::cout << "Good-bye\n"; }

};

// Create the world
World theWorld;

int main()
{
    return 0;
}
```

This program can be broken up into four distinct sections:

- The include declaration.
- The definition of the world class.
- The declaration of the world object.
- The main function.

Starting at the bottom the main function serves as the entry point for the program and is called by the operating system when a C++ executable is invoked. This function must be present in all C++ programs and is a relic kept for compatibility with C. From the looks of it main seemingly does nothing except return the value 0. This return value simply states that the program ran to completion successfully. Nothing else happens inside the main function so it would be reasonable to believe that the program itself does nothing. However, as stated earlier, running this program will cause the word "Hello" and "Good-Bye" to be printed to the screen. This behavior is due to the next section up.

The statement of `World theWorld` declares an object of type `World` with the name `theWorld`. This definition occurs outside any curly braces and is therefore

considered to be a global object. When a global object is declared it is actually initialized before the call to `main`. So in this case the world is created even before the program starts running.

Following the program from the bottom up the definition of the `World` type is found. The `class` keyword in front of `World` tells the compiler that `World` is a user defined type. The definition of the `World` class is delimited by curly braces as usual but there is a slight difference at the end of a class as opposed to the end of a function or loop. The right brace is followed by a semicolon, making the declaration of a class a C++ statement. This may seem a bit strange and a little redundant (couldn't the compiler figure this out from the `class` keyword?) but this is the way objects are declared in C++. Within the definition for the `World` class there are two functions, one named `World` and the other named `~World` . These functions are called member functions which means they are only defined for objects of type `World`. Member functions will be discussed in more detail a bit later. Just above these member functions is the `public` keyword followed by a colon. In a class the `public` keyword means that any member functions or member variables (classes can also hold data) below are considered to be public and any user of the class may access them. There are two other keywords (protected and private) that specify access to the class but these are not important for the current discussion.

The first member function named `World` is a special function called a constructor. A constructor has the same name of it's corresponding class and may or may not take arguments. In this case the `World` constructor takes no arguments so it is called a default constructor. A constructor is used to initialize an object but is never actually called directly. The declaration of the object automatically calls the constructor. An important thing to remember about constructors is that an object must be fully functional upon its construction. For this case the constructor doesn't do much but if an object needs to perform initialization operations, such as allocating memory or

opening a database connection, the constructor is the place to do it. Inside the `World` constructor is the following statement

```
std::cout << "Hello\n";
```

This statement may look a bit strange but all it says is that the `std::cout` object is sent the string `"Hello\n"`. The `std::cout` object is an object defined by the C++ standard library (not the language itself) that represents the output of the system, usually the computer screen. This is where the Hello comes from when the program is run. The "\n" at the end of "Hello" simply prints a new line which means the next string sent to the `std::cout` object will start printing on the next line.

The next member function named `~World` is also a special member function called a destructor. The destructor is called each time an object of a class is destroyed and it has the same name of the class except that there is a (~) in front of it. Unlike the constructor, the destructor cannot take any arguments. Destructors, like constructors, are also never called directly[3] they are automatically called by the system when an object is destroyed. In the case of the "Hello World" program the destructor for the object `theWorld` is actually called after the function `main` returns.

There are two other special functions associated with classes called the copy constructor and the assignment operator. A copy constructor is used to create an object from a copy of another object of the same type. The assignment operator is used to assign an object of one type to an object of the same type. While these functions are an important part of good object oriented design, a detailed explanation of these functions would be too time consuming to be worthwhile for the current discussion. The interested reader is encouraged to consult any introductory C++ book.

The last statement to look at is the `#include` statement. This line tells the compiler to find a file named `iostream` and place the contents of that file at the position

[3] Well almost never. There are certain cases where a destructor needs to be called directly but they are extremely rare.

of the `#include` statement. The `iostream` file is where `std::cout` is declared. The double colon operator (::) between `std` and `cout` is called the scope resolution operator. The scope resolution operator can have a couple different meanings depending on the context in which it is used. For this case it means that `cout` can be found in the namespace `std`. The `std` namespace holds most of the elements of the standard C++ library.

A namespace is simply a construct that allows groups of variables and functions to reside in a common area without polluting the global namespace. Namespaces are declared much like classes but there is no semicolon following the last curly brace. For example, the `std` namespace is declared

```
namespace std
{
    // Declare things for the std namespace
}
```

Namespaces helps avoid name collisions that happen when two programmers define a function with the same name for example. For example, if there are two programmers whose names are Fred and Joe and they both want to declare a function named `foo` that takes an integer argument they cannot both define the function at global scope like

```
// Fred's foo function
void foo(int i)
{
}

// Joe's foo function
void foo(int j) // *Error*: foo has already been defined
{
}
```

but they can define the same function in different namespaces.

```
namespace Fred
{
    // Fred's foo function
    void foo(int i)  // Now foo is actually Fred::foo
    {
    }
}

namespace Joe
{
    // Joe's foo function
    void foo(int j) // OK : Now foo is actually Joe::foo
    {
    }
}
```

So to summarize the steps executed in the "Hello World" program

1. All global objects are constructed. In this case our object **theWorld** prints "Hello".

2. The `main` function is called. In this case it does nothing.

3. All global objects are destroyed. In this case that means "Good-Bye" is written to the screen.

This example of the C++ "Hello World" program was modeled after an example given on the website http://www.relisoft.com. This website contains a lot of good information about C++ including an online book from which this example was taken. The author of this website has also published a book related to the material contained on the website [37].

The "Hello World" example above only used the constructor and destructor member functions. While these functions are important they are not the only kind of

member functions that can exist in a class. Other member functions are defined similarly to free functions except that they are confined to work with objects of the class in which they are defined. For example, the ability the count the number of people in the world could be added to the world class with the addition of a few member functions and a member variable.

```
#include <iostream>

class World
{
public:
    World()
        : m_number_of_people(0)
    {
        std::cout << "Hello\n";
    }

    ~World()
    {
        std::cout << "Good-bye\n";
    }

    int GetNumberOfPeople() const
    { return m_number_of_people; }

    void AddPerson()
    { m_number_of_people++; }

    void DeletePerson()
    { m_number_of_people--; }

private:
    int m_number_of_people;
};

// Create the world
```

```
World theWorld;

int main()
{
    // Find out how many people are in the world
    std::cout << "The world has " <<
        theWorld.GetNumberOfPeople() << " people\n";

    // Add a person to the world
    theWorld.AddPerson();
    std::cout << "The world has " <<
        theWorld.GetNumberOfPeople() << " person\n";

    // Delete a person from the world
    theWorld.DeletePerson();
    std::cout << "The world has " <<
        theWorld.GetNumberOfPeople() << " people\n";

    return 0;
}
```

There are a few more things to explain here. First three new member functions have been added to the `World` type, called `GetNumberOfPeople`, `AddPerson`, and `DeletePerson`. These functions do exactly what their names imply. The way to access these functions through an object of type `World` is to use the dot (.) operator. As is seen in the above example, the `GetNumberOfPeople` function is called by writing `theWorld.GetNumberOfPeople()`.

The number of people in the world is stored by the member variable `m_number_of_people` and is declared `private`. This means that only functions that are part of the class are allowed to directly manipulate the variable. In other words, only functions that are part of the `World` type may directly change the `m_number_of_people` variable. This type of indirection is very common in object oriented programming and is called "data hiding", although this is a bit of a mis-

nomer. The objective of this type of indirection is not to hide the internal data of the class but rather to protect is from being misused. Hence a better name for this might be "data protection". In any case, it shields the users of the class from the inner workings of the class itself.

It may seem strange to create this kind of indirect access to the `m_number_of_people` variable. Why couldn't the variable be declared `public` and the users of the class access it directly? Then, there would be no need for any of the member functions other than the constructor and destructor. While this might seem like a more efficient way to do things, it can cause problems. Imagine what would happen if the way `World` kept track of the number of people changed. Say that instead of the number of people being stored in a variable, an array of people was stored instead. When `GetNumberOfPeople` was called it would return the size of the array. Using the "data protection" technique the users of this class do not need to concern themselves with this change. In their view the class acts in the same way as before. If the users were allowed direct access to the member variables they would all have to rewrite their code to reflect the change in an implementation detail of the `World` type.

The `const` keyword can have multiple definitions in the C++ language. Earlier it was seen that putting `const` in front of a variable in a function call stated that the variable would not change during the course of the function. In the definition of the `GetNumberOfPeople` the `const` means something different. It tells the compiler that the internal *state* of the `World` type will not change during the course of the `GetNumberOfPeople` function. In other words, the function `GetNumberOfPeople` will not change any of the internal variables of the `World` type. `AddPerson` and `DeletePerson` do not have `const` since they change the internal variable `m_number_of_people` thereby changing the state of the object. Only `GetNumberOfPeople` does not change the internal state of the object so it is declared

const.

One more thing to notice about the class is that the constructor has changed. The line containing :m_number_of_people(0) family is called a constructor initialization list. Initialization lists are used to initialize member variables without the need for assignment. The constructor could also have been written

```
World()
{
    m_number_of_people = 0;
    std::cout << "Hello\n";
}
```

and in this case the two constructors are equivalent. The main difference is that the initialization list is completed even before the constructor of the object is called. For reasons of efficiency the initialization list is usually faster, especially when user defined types are declared as member variables in other user defined types. There is another case where an initialization list must be used. If a type defines any **const** member variables they can only be initialized in the initialization list. Member variables that are **const** are useful for member data that will not be changed during the lifetime of an object, hence they are read-only variables. For example, if an type was created to model a piece of inventory for some sort of warehousing program the item might have some unique id number that identifies it. Since this id number is not going to change it should be declared **const**. Consider the following type

```
class Item
{
public:
    Item(const int id)
        : m_id(id)
    {
    }
```

```
private:
    const int m_id;
}
```

Not a very useful type for sure but the fact that the member variable `m_id` is `const` means that it must be initialized in the initialization list. The following will not work

```
class Item
{
public:
    Item(const int id)
    {
        m_id = id; // *Error* : m_id is const
    }

private:
    const int m_id;
}
```

and trying to do this will cause the compiler to complain.

A.4.2 Structs

Structs, defined by the keyword `struct`, are similar to classes in that they represent a grouping of related data and functions. Structs were originally introduced in the C programming language and were kept in C++ for compatibility. As far as C++ is concerned there is essentially no difference between classes and structs. Therefore anything that can be said about classes can be said about structs. There is only on minor difference between structs and classes. The default access level in structs is `public` while the default access level in classes is `private`. This makes the following code equivalent

```
struct S
{
    // Default access is public
    int d1;
```

```
    int d2;
}

class C
{
// Default access is private, public must be specified
public:
    int d1;
    int d2;
}
```

In C++ structs are instantiated in the same way as classes and can have functions, constructors, destructors and anything else that a class can contain.

A.4.3 Templates

There are times when the internal type of a class will not be known at the time the class is designed. For example, suppose a class representing a dynamic array is being designed. A dynamic array is one that grows in size as needed and resolves the problem of having to guess at a "maximum value" of an array during the design of a program. A basic interface for a dynamic array class is easily defined

```
class DynArray
{
public:
    DynArray();

    int GetNumberOfItems() const;
    void AddItem(double item);
    void DeleteItem(int index);

private:
    int m_size;
    double * m_items;
};
```

The details of how the class might be implemented are not important for this discussion. Rather it is noted that only items of the type **double** can be held in this dynamic array. If the user wanted to have an dynamic array of integers, possibly even a dynamic array of a user defined type, this class is not suitable. Sure another similar class could be written for all the built-in types where the only change would be the type of data the class holds, but this is a lot of unnecessary work and a maintenance nightmare. Each time a function was added to any one of the classes it would have to be reproduced in all the other classes. On top of that, it would still not solve the problem of creating a dynamic array of a user defined type. The reality is that there is no way to know beforehand what type the user may want to have in a dynamic array.

Enter templates. Templates allow a programmer to defer the type of internal data in a class like **DynArray** until the user specifies it. Templates are created using the **template** keyword. To create a template for the **DynArray** class the following syntax is used

```
template <class T>
class DynArray
{
public:
    DynArray();

    int GetNumberOfItems() const;
    void AddItem(T item);
    void DeleteItem(int index);

private:
    int m_size;
    T * m_items;
};
```

Notice that the only change in the class interface is the addition of the line **template <class T>** and the change of the world **double** to the letter **T** for the type of the member variable **m_items** and in the function **AddItem**. The first line declares this class to be a template that is *parameterized* by the type **T**. This means that the type of **T** will not be known until a user of this class specifies it. A user of this class would have to specify the type **DynArray** is to hold when declaring the array.

```
void f()
{
    DynArray<int> da;
    da.AddItem(1);
    da.AddItem(2);
    da.DeleteItem(0);
}
```

When the compiler encounters the statement **DynArray<int>** it produces a entire copy of the **DynArray** class using **int** instead of **T** in the class definition. This code is then inserted into the program as if the **DynArray** class was declared to use integer data. If the user specifies another **DynArray** with the type **double** such as **DynArray<double>** another copy of the **DynArray** template is created but this time **double** is used in place of **T** in the class definition.

There are more things that can be done with templates but implementing "containers of T" is one of the most common uses. The standard template library, to be discussed later, makes heavy use of templates, hence its name.

A.5 Object Oriented Programming with C++

The classes (and structs) defined in the previous section constitute the cornerstone of object oriented design in C++. The techniques described in this section make use of the class concept to provide a way to create truly object oriented software.

A.5.1 Composition

Composition is a technique of embedding one user defined type inside another user defined type and is a very common technique in C++ programming. The concept of composition defines the "has-a" relationship, so in effect one type "has-a" object of another type. For example a rectangle can be defined by two points, its upper left point and its lower right point. In C++ this can be defined like

```
class Point
{
public:
    Point(int x = 0, int y = 0)
        : m_x(x), m_y(y)
    {
    }

    void Set(int x, int y)
    { m_x = x; m_y = y; }

    void Get(int& x, int& y)
    { x = m_x; y = m_y; }

private:
    int m_x;
    int m_y;
}

class Rectangle
{
public:
    Rectangle(Point upper_left, Point lower_right)
        : m_upper_left(upper_left), m_lower_right(lower_right)
    {
    }

private:
    Point m_upper_left;
```

```
    Point m_lower_right;
}
```

Here there are two objects of type `Point` embedded directly in the `Rectangle` type. The constructor for `Point` introduces a new concept, the concept of default arguments. In order to embed the `Point` objects in the `Rectangle` class the `Point` object must be able to be constructed without any arguments. The `Point` constructor takes two arguments but these arguments have suitable default values so these values can be specified in the constructor. For any arguments not specified by the user the default argument will be used.

A.5.2 Inheritance

Inheritance is the ability to inherit functionality in a new user defined type from an existing user defined type. Inheritance defines an "is-a" relationship. It is said that when one type inherits from another type the new type "is-a" object of the existing type. An example of inheritance is defining a new type called `Dog` from a type called `Animal`. Since a dog "is-a" animal it can inherit all the functionality of an animal and define some new functionality of it's own. In code this is stated as

```
class Animal
{
public:
    // Various animal functions
};

class Dog : public Animal
{
public:
    // Includes all animal functions plus any
    // that dog defines
};
```

The colon operator in the declaration of `Dog` is used to define inheritance and in this case it is specified that Dog will inherit publicly from Animal. There is also protected

and private inheritance but for the sake of space they will not be discussed here. Public inheritance is the most prevalent and to understand the material in Chapter 7, a knowledge of protected and private inheritance is unnecessary.

A.5.3 Polymorphism

Inheritance by itself is a nice feature, it allows the programmer to group common characteristics of multiple types into a common base class, but this is not the most powerful way to use inheritance.

Polymorphism is the ability for one thing to act as many. For C++ this means that one type can act as multiple types and the way to achieve this in through inheritance. Through the use of a special function called a virtual function, a pointer (or reference) to a base class can actually be used to call functions in a derived class. This is an extremely powerful technique and is helpful in creating robust and extensible programs. Continuing with the example of the **Animal** and **Dog** classes, polymorphism can be introduced by the following code

```
#include <iostream>

class Animal
{
public:
    virtual void Run() = 0;
    // Various animal functions
};

class Dog : public Animal
{
public:
    void Run() { std::cout << "Dog::Run\n"; }
    // Includes all animal functions plus any
    // that dog defines
};
```

```
void MakeAnimalRun(Animal * animal)
{
    // Member functions are called with the (->) operator for
    // pointers rather than the (.) operator
    animal->Run(); // Calls Dog::Run()
}

int main()
{
    Dog dog;
    MakeAnimalRun(&dog);
    return 0;
}
```

Running this code would produce the phrase "Dog::Run" to be printed to the screen. This may seem strange because the type of pointer in **MakeAnimalRun** is **Animal** but the actual type that it points to is of type **Dog** so the **Dog::Run** function is called. This magic occurs because the **Run** member function is declared **virtual** in the **Animal** class (actually its declared pure virtual, but that will be explained shortly) which means the function call is not resolved until runtime. This means that the function **Run** is associated with the type of object a pointer points to rather than the type of the pointer. How this magic is achieved is beyond the scope of this discussion but the result is truly amazing. If another class called **Cat** was derived from **Animal** and the **MakeAnimalRun** function was passed a pointer whose actual type was **Cat** it would call the **Cat::Run** function. Cool huh?

A note about the `= 0` at the end of the declaration the **Animal** class. When a virtual function is defined without this syntax in a base class it means that any derived classes *may* redefine this function. In other words the function has a default implementation. When a virtual function is defined with the `= 0` syntax it is considered to be a "pure" virtual function. Pure virtual functions do not have to have a

default implementation (although they can) and when a class inherits from this class it *must* redefine this function. So in the example above if `Dog` had not redefined the `Run` function it would have caused the compiler to issue an error.

A.6 The Standard Template Library

The Standard Template Library, or STL, is a subset of the entire C++ standard library (which is not explicitly discussed in this appendix) that supplies the C++ program standard implementations of many of the most useful computer science techniques and algorithms. The section cannot even hope to give a complete view of the STL as entire books have been written about it. Some of the books are mentioned in the Remarks at the end of this appendix. Rather the goal of this section is to briefly overview some of the most widely used portions of the STL. The STL is broken up into two major components: containers and algorithms. Only containers will be discussed here as they are the only relevant parts for the software design discussion in chapter 7.

A.6.1 STL Containers

The STL containers provide a standard way for all C++ programmers to use various data structures that arise is almost all computer programs. There are two types of containers: sequence containers and associative containers. The sequence containers are `vector`, `string`, `list` and `deque`. These containers help the program create arrays or lists of objects that grow as needed without the programmer having to worry about any memory issues. The associative containers include `set`, `map`, `multiset`, and `multimap`. The associative containers allow the programmer to relate one type to another and provide fast lookup for items. A standard container that is not officially an STL container but is still widely used is `valarray`. This container provides a highly optimized container that can be used for numerical arrays. There is also one important non-standard associative container that is nonetheless widely available

called **hash_map**. The container provides the implementation of a hash table, but since it is not standardized implementations of it are platform dependent. Other containers exist in the STL but are not important for the current discussion.

The STL containers make heavy use of templates and as many as four template parameters arguments can be specified for a single container. Most of the time, however, the majority of the template arguments are not required and the STL provides suitable defaults.

The most prevalent of the two sequence containers are **vector** and **string**. The **vector** container allows the programmer to create a dynamic array of objects that grows as needed. Consider the following code

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);

    std::cout << "The vector has " << v.size() << "
    elements\n";

    return 0;
}
```

Running this code will produce the result that the **vector** has two elements but nowhere is there the need to allocate any memory. The **vector** does all the memory allocation itself and takes care of all de-allocation at the end of the program.

The **string** container works a bit differently from the rest of the containers in that it can be instantiated without the use of any template parameters. The following code creates and string and assigns it the word "Hello".

```
#include <iostream>
#include <string>

int main()
{
    std::string str;
    str = "Hello";
    return 0;
}
```

Although it may seem like the **string** container is not a template it actually is. The template for string is actually **basic_string** but a type definition is used to create the illusion of a non-templated string class however knowing this is not crucial to using the **string** class.

The rest of the containers work in a way very similar to the **vector** container, although each has a little different functionality so they are not interchangeable. Choosing which container to use for a specific problems takes experience.

A.7 Remarks

Trying to compress a detailed explanation of the C++ programming language and the C++ standard library into the appendix of a dissertation is like trying to fit an 800 pound gorilla into a size 2 dress. Something has to give. This appendix presented a mostly cursory treatment of some of the concepts of C++ while other important concepts were omitted altogether (e.g. copy constructors, class assignment operators, inline functions, static variables and functions, exceptions, and operator overloading among others). The hope is that the reader unfamiliar with C++ will have garnered enough information to follow the discussion of software design in Chapter 7. Interested readers are strongly encouraged to consult the many references given on introductory C++ for more complete information.

In addition to the introductory texts already mentioned there are a number of other books that are very popular reference texts for C++. Most of the following references have more to do using the language more effectively rather than describing the language itself. For more information on how to effectively use C++ the books by Scott Meyers [38], [39] are some of the best. A bit more problem oriented are the books by Herb Sutter [40], [41]. The definitive reference of the Standard Library is by Nicolai Josuttis [42] while books on effectively using the Standard Template Library include [43] and [44]. For C++ efficiency considerations [45] is the definitive (and only) reference. For more advanced techniques including some revolutionary template programming techniques see [46] and [47].

Newsgroups abound on the Internet and C++ has at least three different groups dedicated to different aspects of the language. The most popular is comp.lang.c++ where people ask questions about all sorts of language issues. The signal to noise ratio of posts in this group can be rather low and there are definitely some strange characters that hang out there. There is also a moderated version, aptly named comp.lang.c++.moderated, where a concerted effort is made to raise the signal to noise ratio of posts. Another group called comp.std.c++ is dedicated to the discussion of the standardization of C++. While this group can be a little technical there is the possibility of finding out some interesting tidbits from time to time.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Jianming Jin, *The Finite Element Method in Electromagnetics*, John Wiley & Sons Inc., New York, NY, 1993

[2] John L. Volakis, Arindam Chatterjee, Leo C. Kempel, *Finite Element Methods for Electromagnetics*, IEEE Press, New York, NY, 1998.

[3] X. Yuan, D.R. Lynch, K.Paulsen, "Importance of normal field continuity in inhomogeneous scattering calculations", *IEEE Trans. Microwave Theory Tech.*, Vol. 29, pp. 638-642, April 1991

[4] H. Whitney, *Geometric Integration Theory*, Princeton University Press, NJ, 1957

[5] Jian-Ming Jin, John L. Volakis "TE Scattering by an Inhomogeneously Filled Aperture in a Thick Conducting Plane", *IEEE Trans. on Ant. and Propag.* Vol. 38, No. 8, pp. 1280-1286, August 1990

[6] Tom Cwik, Cinzia Zuffada, Vahraz Jamnejad, "Modeling Three-Dimensional Scatterers Using a Coupled Finite Element-Integral Equation Formulation", *IEEE Trans. on Ant. and Propag.*, Vol. 44, No. 4, pp. 453-459, April 1996

[7] Jui-Ching Cheng, Nihad I. Dib, Linda P.B. Katehi, "Theoretical Modeling of Cavity-Backed Patch Antennas Using a Hybrid Technique", *IEEE Trans. on Ant. and Propag.*, Vol. 43, No. 9, pp. 1003-1013, September 1995.

[8] Ke-Li Wu, Chen Wu, John Litva, "Characterizing Microwave Planar Circuits Using the Coupled Finite Element-Boundary Element Method", *IEEE Trans. on Microwave Theory and Technique*, Vol. 40, No. 10, pp. 1963-1966 October 1992

[9] Jian-She Wang, Raj Mittra, "Finite Element Analysis of MMIC Structures and Electronic Packages Using Absorbing Boundary Conditions", *IEEE Trans. on Microwave Theory and Technique*, Vol. 42, No. 3, pp. 441-449, March 1994

[10] A. Chatterjee, J.L. Volakis "Conformal Absorbing Boundary Conditions for the Vector Wave Equation", *Microwave and Optical Technology Letters*, Vol. 6, No. 16, pp. 886-889, 1993

[11] J.M. Jin, J.L. Volakis, V.V. Leipa, "Fictious Absorber for Truncating Finite Element Meshes in Scattering", *IEE Proceedings-H*, Vol. 139, No. 5, pp. 472-476, 1992

[12] Robert E. Collin, *Field Theory of Guided Waves*, IEEE Press New York, NY, 1991

[13] W.C. Chew, "Some Observations on the Spatial and Eigenfunction Representations of Dyadic Green's Function", *IEEE Trans. on Antennas and Propagation*, Vol. 37, pp. 1322-1327, 1989

[14] J. Vanbladel, "Some Remarks on Green's Dyadic for Infinite Space", *IEEE Trans. on Antennas and Propagation*, Vol. 9, pp. 563-566, 1961

[15] M.S. Viola and D.P. Nyquist, "An Observation on the Sommerfeld-Integral Representation of the Electric Dyadic Green's Function for Layered Media", *IEEE Trans. on Microwave Theory and Technique*, Vol. 36, pp. 1289-1292, 1988

[16] J.S. Bagby and D.P. Nyquist, "Dyadic Green's Function for Integrated Electronic and Optical Circuits", *IEEE Trans. on Microwave Theory and Technique*, Vol. 35, pp. 206-210, 1987

[17] Chen-To Tai, *Dyadic Green Functions in Electromagnetic Theory* IEEE Press, New York, NY, 1994

[18] Leo C. Kempel and John L. Volakis, *Radiation and Scattering from cylindrically conformal printed antennas*, Ph.D. Dissertation, The University of Michigan, Ann Arbor, MI, 1994

[19] Chei-Wei Wu, *Coupling between Cavity-Backed Antennas on an Elliptic Cylinder*, Ph.D. Dissertation, Michigan State University, East Lansing, MI, 2001

[20] Charles Macon, *Modeling the Radiation from Cavity-Backed Antennas on Prolate Spheroids using a Hybrid Finite Element-Boundary Integral Method*, Ph.D. Dissertation, Michigan State University, East Lansing, MI, 2001

[21] D.R. Wilton, S.M. Rao, A.W, Glisson, D.H. Schaubert, O.M. Al-Bundak, C.M. Butler. "Potential Integrals for Uniform and Linear Source Distributions on Polygonal and Polyhedral Domains", *IEEE Transactions on Antennas and Propagation*, Vol. 32, No. 3, March 1984

[22] C.J. Reddy, M.D. Deshpande, C.R. Cockrell, and F.B. Beck, "Radiation characteristics of cavity backed aperture antennas in finite ground plane using the hybrid FEM/MoM technique and geometrical theory of diffraction", *IEEE Transactions on Antennas and Propagation*, Vol. 40, No. 10, pp. 1327-1333, October 1996

[23] Roger F. Harrington, *Time Harmonic Electromagnetic Fields*, McGrawHill Book Company, New York, NY, 1961

[24] Tayfun Ozdemir, John L. Volakis, "Triangular Prisms for Edge-based Vector Finite Element Analysis of Conformal Antennas ", *IEEE Trans. on Ant. and Propag.*, Vol. 45, No. 5, pp. 788-797, May 1997

[25] S.M. Rao, D.R. Wilton, A.W. Glisson, "Electromagnetic Scattering by Surfaces of Arbitrary Shape", *IEEE Transactions on Antennas and Propagation*, Vol. 30, No. 3, May 1982

[26] O.C. Zienkiewicz, R.L. Taylor, *The Finite Element Method Volume I*, McGraw-Hill Book Company, Maidenhead, Berkshire, England, 1967

[27] William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling, *Numerical Recipes in C : The Art of Scientific Computing*, Cambridge University Press, 2nd edition 1993

[28] George Ernest Antilla, *Radiation and Scattering from Complex Three-Dimensional Geometries using a Curvilinear Hybrid Finite Element-Integral Equation Approach*, Ph.D. Dissertation, University of California, Los Angeles, CA, 1993

[29] Andrew F. Peterson, Scott L. Ray, Raj Mittra, *Computational Methods for Electomagnetics*, IEEE Press, New York, NY, 1998

[30] Robert G. Corzine, Joseph A. Mosko, *Four Arm Spiral Antennas*, Artect House Inc, Norwood, MA, 1990

[31] Stephen J. Chapman, *FORTRAN 90/95 for Scientists and Engineers*, WCB/McGraw-Hill, Boston, MA, 1998

[32] Bjarne Stroustrup, *The C++ Programming Language*, Third Edition, Addison-Wesley, Boston, MA, 1997

[33] Victor Shtren, *Core C++ : A Software Engineering Approach*, Prentice Hall, Upper Saddle River, NJ, 2000

[34] Andrew Koenig, Barbara E. Moo, *Accelerated C++ : Practical Programming by Example*, Addison-Wesley, Boston, MA, 2000

[35] Stanley B. Lippman, Jose Lajoie, Josee Lajoie, *C++ Primer*, Addison-Wesley, Boston, MA, 1998

[36] Bjarne Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, Boston, MA, 1994

[37] Bartosz Milewski, *C++ in Action: Industrial Strength Programming Techniques*, Addison-Wesley, Boston, MA, 2001

[38] Scott Meyers, *Effective C++*, Second Edition, Addison-Wesley, Boston, MA, 1998

[39] Scott Meyers, *More Effective C++*, Addison-Wesley, Boston, MA, 1997

[40] Herb Sutter, *Exceptional C++*, Addison-Wesley, Boston, MA, 2000

[41] Herb Sutter, *More Exceptional C++*, Addison-Wesley, Boston, MA, 2001

[42] Nicolai M. Josuttis, *The C++ Standard Library*, Addison-Wesley, Boston, MA, 1999

[43] Matthew H. Austern, *Generic Programming and the STL*, Addison-Wesley, Boston, MA, 1999

[44] Scott Meyers, *Effective STL*, Addison-Wesley, Boston, MA, 2001

[45] Don Bulka and David Mayhew, *Efficient C++*, Addison-Wesley, Boston, MA, 2000

[46] Andrei Alexandrescu, *Modern C++ Design : Generic Programming and Design Patterns Applied*, Addison-Wesley, Boston, MA, 2001

[47] David Vandevoorde and Nicolai M. Josuttis *C++ Templates: The Complete Guide*, Pearson Education, 2002