The Pennsylvania State University

The Graduate School

Department of Electrical Engineering

OPTICAL FREQUENCY SELECTIVE SURFACE DESIGN USING A GPU ACCELERATED FINITE ELEMENT BOUNDARY INTEGRAL METHOD

A Dissertation in

Electrical Engineering

by

Jason A. Ashbach

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

May 2016

The dissertation of Jason Ashbach was reviewed and approved* by the following:

Douglas H. Werner
Professor of Electrical Engineering
Dissertation Advisor
Chair of Committee

Timothy Kane
Professor of Electrical Engineering

George Kesidis
Professor of Electrical Engineering

Lyle N. Long
Distinguished Professor of Aerospace Engineering

Kultegin Aydin
Professor of Electrical Engineering
Head of the Department of Electrical Engineering

*Signatures are on file in the Graduate School

ABSTRACT

Periodic metallodielectric frequency selective surface (FSS) designs have historically seen widespread use in the microwave and radio frequency spectra. By scaling the dimensions of an FSS unit cell for use in a nano-fabrication process, these concepts have recently been adapted for use in optical applications as well. While early optical designs have been limited to well-understood geometries or optimized pixelated screens, nano-fabrication, lithographic and interconnect technology has progressed to a point where it is possible to fabricate metallic screens of arbitrary geometries featuring curvilinear or even three-dimensional characteristics that are only tens of nanometers wide.

In order to design an FSS featuring such characteristics, it is important to have a robust numerical solver that features triangular elements in purely two-dimensional geometries and prismatic or tetrahedral elements in three-dimensional geometries. In this dissertation, a periodic finite element method code has been developed which features prismatic elements whose top and bottom boundaries are truncated by numerical integration of the boundary integral as opposed to an approximate representation found in a perfectly matched layer. However, since no exact solution exists for the calculation of triangular elements in a boundary integral, this process can be time consuming. To address this, these calculations were optimized for parallelization such that they may be done on a graphics processor, which provides a large increase in computational speed.

Additionally, a simple geometrical representation using a Bézier surface is presented which provides generality with few variables. With a fast numerical solver coupled with a low-variable geometric representation, a heuristic optimization algorithm has been used to develop several optical designs such as an absorber, a circular polarization filter, a transparent conductive surface and an enhanced, optical modulator.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Douglas H. Werner, for his guidance and patience throughout my graduate program. I would also like to thank Dr. Keith Lysiak for his support through the Applied Research Lab and Professors Timothy Kane, Lyle Long and George Kesidis for serving on my committee and providing their insights on this work. Finally, I would like to thank Danielle Kiger for her love and support she has given me throughout the years and my mother for raising me and keeping me motivated to continue my education.

Chapter 1
Introduction and Technical Approach

Introduction

This chapter introduces the topic of the dissertation, which is the development of a finite element boundary integral method that has been accelerated using massively parallel compute devices such as a graphics-processing unit. The technical approach and original contributions are outlined.

Statement of the Problem

Periodic metallodielectric structures have recently been investigated by several groups for use in a variety of novel applications. These include optical applications of the Frequency Selective Surface (FSS) technology that was originally developed for microwave applications such as absorption filters [1, 2] to investigations into development of Negative Index Materials in an effort to achieve the perfect lens proposed by Pendry [3, 4]. As material lithographic technologies have progressed, the ability to produce periodic metallic screens with increasingly complicated geometries with nanoscale variations has begun to become a more viable approach to optical filter design. Furthermore, in an effort to move beyond standard element models, which have analytic solutions available, it becomes necessary to develop an effective numerical approach to this sort of problem. It is also necessary to develop an effective model with which one can achieve a desired response. A method is proposed here using a GPU accelerated hybrid

periodic finite-element boundary-integral (FEBI) method with prismatic meshes and Bézier surfaces to optimize quickly and effectively frequency selective surfaces.

Technical Approach

The research presented here is based on the FSS technology originally developed for microwave applications. FSS are planar devices comprised of a doubly periodic metallic screen, which is printed on a substrate such as the example FSS shown in Figure 1-1. These devices operate as spatial filters for electromagnetic radiation. While early designs focused on the radio frequency spectrum, the techniques used can be scaled down to micron or nanometer dimensions for operation at IR and optical frequencies. Our lab, for instance, has synthesized, fabricated and measured FSS filters at a variety of frequency regions using analytic and numerical approaches. These approaches have typically been done using a fractal technique [5] or a two-dimensional periodic numerical method such as the method of moments approach or a brick-element FEBI approach [6]. The advantage of using a numerical approach allows us to synthesize filters using a derivative-free stochastic optimization process such as the well-known genetic algorithm (GA) or more recent covariance matrix adaptation evolution strategy (CMA-ES) which allow for more flexibility in the synthesis of designs.

These stochastic optimization techniques are well suited for generalized FSS designs using a robust curvilinear geometry approach, which can be used with the finite element boundary integral method. Recent advances in semiconductor have opened the doors to a variety of lithography techniques, which can mass-produce fine patterns with unusual geometries making a study of generic curvilinear geometries more attractive than simpler pixelated designs.

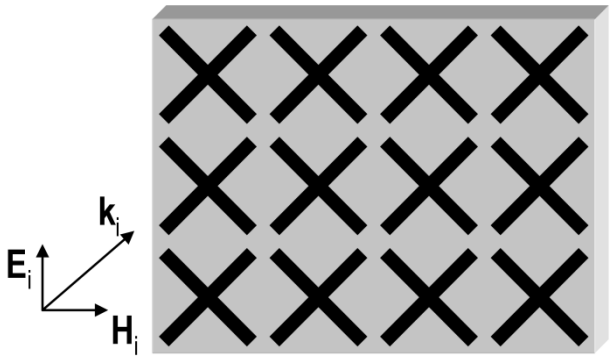


Figure 1-1: Diagram of a frequency selective surface composed of a periodic metallic screen that has been printed on a dielectric substrate.

The curvilinear geometry approach is achieved using a Bézier surface, which is a method to achieve arbitrary surface patterns using a limited number of variables. Using this technique, several example designs were optimized which might see practical application in future optical devices.

Original Contributions

The research I have performed during my Ph.D. studies at the Pennsylvania State University have led to several practical applications for optical FSS design.

- The development of a GPU optimized finite element boundary integral solver
- The use of Bézier surfaces coupled with a stochastic optimization routine for use in designing optical frequency selective surfaces and other two-dimensional metallic screens.
- The development of a wide-angle circularly polarized filter, which can be designed to transmit frequencies within the narrow CO2 absorption band while blocking nearby interferences or broadly turned for use in a communications system.

- The design of a filter that confines optical energy into a semiconductor substrate, which could potentially see applications in solar technologies.
- The design of an optical transparent metallic interconnect which is continuous periodically.
- The use of a metallic FSS screen, which enhances the variability in absorption for a silicon germanium electro-optic modulation system.

Overview

This dissertation introduces the use of GPU-based compute device to enhance the calculation speed of the boundary integral impedance matrix for use in a finite element solver. Prior to this, a finite element method, which is truncated by a triangular boundary integral, could up to several hours to calculate individual frequency points. Because of this, robust triangular finite element codes have had a tendency to use an approximate boundary condition such as the perfectly matched layer. These types of boundary conditions require the use of large meshes and have a large impact on the overall number of unknowns. This has an effect on the overall amount of RAM required to solve such a system causing many applications to require extremely powerful and expensive workstations to begin to approach such problems.

Chapter 2 introduces the finite element method and lays the mathematical foundation for the software that has been written.

Chapter 3 details the software that has been written and the design approaches that were done in the optical system design.

Chapter 4 details the Bézier surface as a practical tool for designing optical frequency selective surfaces using optical absorbers as an example.

In chapter 5, an approach for designing oblique-angle circular polarization filter is introduced. Using this polarization filter, two designs are presented which are optimized for applications near the CO2 absorption band and in the fibre-optics telecommunications band.

In chapter 6, a transparent metallic screen is shown. This filter was optimized under the restriction that there be limited polarization dependence and a continuous path between unit cells such that current can flow.

Chapter 7 discusses the state of the art for electro-absorption modulators and provides a suggestion for an FSS design that couples well with the change in absorption in such a device enhancing the absorption when the applied field is such that the dielectric absorption is higher.

The final chapter provides a summary of this work and provides suggestions for future work.

Chapter 2
Mathematical Foundations for the Periodic Finite Element Boundary Integral for Electromagnetic Problems

Background

In the 19th century, James Clerk Maxwell's insight unified four separate theorems concerning electric and magnetic fields into a set of equations that form the basis of electrodynamics, optics and circuit theory. These equations describe the behavior of the electric and magnetic fields, $\boldsymbol{E}$ and $\boldsymbol{H}$ respectively, with the commonly used differential form reproduced below in equations 2-1 through 2-4 with current density $\boldsymbol{J}$ and magnetic current density $\boldsymbol{M}$, and material properties consisting of the charge density, ρ, magnetic charge density, ρ_m, the permittivity, ϵ, and permeability, μ.

$$\nabla \cdot \boldsymbol{E} = \frac{\rho}{\epsilon} \quad (2\text{-}1)$$

$$\nabla \cdot \boldsymbol{H} = \frac{\rho_m}{\mu} \quad (2\text{-}2)$$

$$\nabla \times \boldsymbol{E} = -\boldsymbol{M} - \mu \frac{\partial \boldsymbol{H}}{\partial t} \quad (2\text{-}3)$$

$$\nabla \times \boldsymbol{H} = \boldsymbol{J} + \epsilon \frac{\partial \boldsymbol{E}}{\partial t} \quad (2\text{-}4)$$

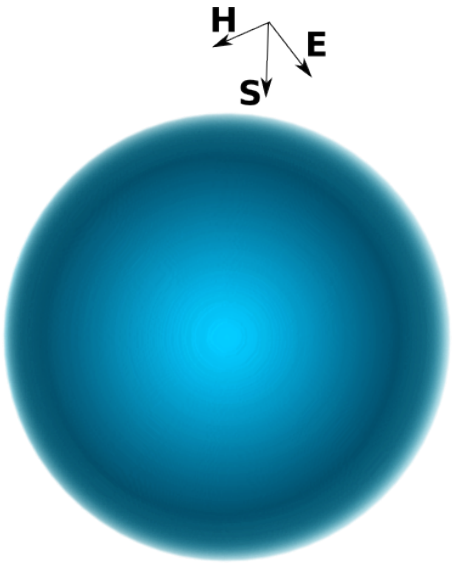


*Figure 2-1: Electromagnetic interaction with sphere can be solved using scattering theory. The vector incident upon the sphere describes an electric field, **E**, and a magnetic field, **H**, travelling in the **S** direction.*

Figure 2-2: An FSS with complicated geometry must be solved using a numerical method.

With these equations, one can determine the field vectors for electromagnetic waves traveling through media. When a wave impinges on a boundary, a discontinuity occurs in the material parameters. With simple media or simple boundaries such as in Figure 2-1, these interactions can oftentimes be solved analytically with some effort. However, in the case of a generalized frequency selective surface such as below in Figure 2-2, which contains multiple

materials with some generic geometry, solving for the field interactions becomes in many cases impractical. In this case, it becomes desirable to design a method that can be used to estimate the field response to such a material numerically. Fortunately, this is possible by reformulating Maxwell's equations into a set of integral equations and solving these numerically. This dissertation will address this by deriving a formulation for the hybrid finite-element boundary-integral method accelerated through massively parallel calculation using general-purpose graphics processing unit (GPGPU) programming.

The Time-Harmonic Form of Maxwell's Equations

In many cases, the solutions to Maxwell's equations are sinusoidal in time. This includes every problem that will be considered herein. Time-harmonic wave such as these have a generalized form that is represented by $e^{j\omega t}$. Using this form, one may insert $e^{j\omega t}$ insert Maxwell's equations as written in 2-1 through 2-4. Using basic derivative rules, one can replace $\frac{\partial}{\partial t}$ with $j\omega$ which simplifies Maxwell's equations such that only the spatial components need to be considered. Thus, one formulate two time-harmonic wave equations for fields $\boldsymbol{E}$ and $\boldsymbol{H}$ as in equations 2-5 and 2-6.

$$\nabla^2 \boldsymbol{E} = \nabla \times \boldsymbol{M} + j\omega\mu \boldsymbol{J} + \frac{1}{\epsilon}\nabla\rho + j\omega\mu\sigma \boldsymbol{E} - \omega^2 \mu\epsilon \boldsymbol{E} \quad (2\text{-}5)$$

$$\nabla^2 \boldsymbol{H} = \nabla \times \boldsymbol{J} + \sigma \boldsymbol{M} + j\omega\sigma \boldsymbol{M} + \frac{1}{\mu}\nabla\rho_m + j\omega\mu\sigma \boldsymbol{H} - \omega^2 \mu\epsilon \boldsymbol{H} \quad (2\text{-}6)$$

Boundary Conditions

Maxwell's equations can be used to solve for the field vectors in continuous media. However, most problems of interest are not confined to simple continuous media. When an

electromagnetic system is considered which is not confined to continuous media, Maxwell's equations cannot be solved without being provided the boundary conditions, which is problem-dependent. For these, one must consider the integral form of Maxwell's equations, written below in equations 2-7 through 2-10. These equations describe the field relations and material properties over an extended region of space rather than instantaneously at a specific point in space as in the differential formulation with Q_e and Q_m being the total electric and magnetic charges respectively.

$$\oint_C \boldsymbol{E} \cdot d\boldsymbol{l} = -\iint_S \boldsymbol{M} \cdot d\boldsymbol{s} - \frac{\boldsymbol{\partial}}{\boldsymbol{\partial t}} \iint_S \mu \boldsymbol{H} \cdot d\boldsymbol{s} \qquad (2\text{-}7)$$

$$\oint_C \boldsymbol{H} \cdot d\boldsymbol{l} = \iint_S \boldsymbol{J} \cdot d\boldsymbol{s} + + \frac{\partial}{\partial t} \iint_S \epsilon \boldsymbol{E} \cdot d\boldsymbol{s} \qquad (2\text{-}8)$$

$$\oiint_S \epsilon \boldsymbol{E} \cdot d\boldsymbol{s} = Q_e \qquad (2\text{-}9)$$

$$\oiint_S \mu \boldsymbol{H} \cdot d\boldsymbol{s} = Q_m \qquad (2\text{-}10)$$

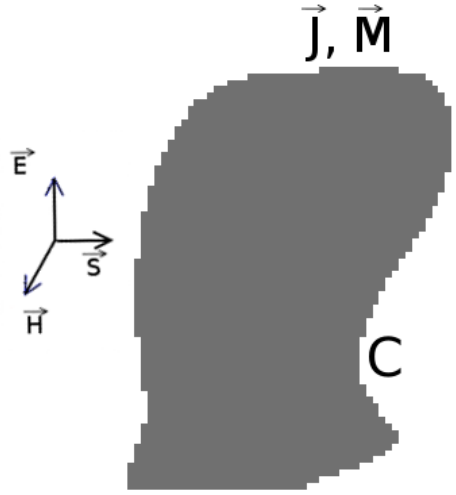


Figure 2-3: A completely arbitrary electromagnetic system.

Consider the geometry displayed in Figure 2-3. A wave that interacts with the boundary within the confines of the outlined contour, C, induces currents within the boundary governed by equations 2-5 and 2-6. Thus, for a finite conductive media as the height $\Delta y \to 0$ the surface integral vanishes and the integrals may be solved as

$$\hat{n}\times(\boldsymbol{E}_2 - \boldsymbol{E}_1) = -\boldsymbol{M} \quad (2\text{-}11)$$

$$\hat{n}\times(\boldsymbol{H}_2 - \boldsymbol{H}_1) = \boldsymbol{J} \quad (2\text{-}12)$$

The boundary conditions can be completed by determining the conditions along the interface. Thus, one may integrate 2-7 and 2-8 simply arriving at the following boundary conditions.

$$\hat{n}\cdot(\epsilon_2\boldsymbol{E}_2 - \epsilon_1\boldsymbol{E}_1) = \rho_e \quad (2\text{-}13)$$

$$\hat{n}\cdot(\mu_2\boldsymbol{H}_2 - \mu_1\boldsymbol{H}_1) = \rho_m \quad (2\text{-}14)$$

Here, ρ_e and ρ_m represent the electric and magnetic charge densities respectively. For a source-free media, which is not a perfect conductor, one may assume that no charges exist along the surface. In this case, the right hand side of equations 2-9 through 2-12 may be set to zero. Furthermore, physically magnetic charge and current cannot occur so for most cases one may simplify the boundary conditions by setting $\boldsymbol{M}$ and ρ_m to zero. From these equations, one can see that the normal components of the electric and magnetic fields are discontinuous across a boundary.

Oftentimes, when one is designing a numerical approximation of an electromagnetic system, the use of the exact boundary conditions, seen above, becomes impractical. For this reason, approximate boundary conditions, such as the Standard Impedance Boundary Condition (SIBC), have been developed. The SIBC is derived from the case of a plane wave travelling from one medium of infinite extent with impedance $Z_1 = \sqrt{\frac{\mu_1}{\epsilon_1}}$ into another medium of infinite extent

with impedance, $Z_2 = \sqrt{\frac{\mu_2}{\epsilon_2}}$. From the time-harmonic form of Maxwell's equations, it is known that the electric field and the magnetic fields can be related through the surface currents as

$$\boldsymbol{J_s} = \hat{n}\times(\boldsymbol{H}_2 - \boldsymbol{H}_1) \tag{2-15}$$

$$\boldsymbol{H} = \frac{\hat{n}\times\boldsymbol{E}}{Z} \tag{2-16}$$

$$\hat{n}\times(\hat{n}\times\boldsymbol{E}) = -Z_1 Z_2 \hat{n}\times\boldsymbol{H} \tag{2-17}$$

A similar process can be used to derive a boundary condition approximation for a thin finite conductive surface. Noting that the current density is related to the electric field through the conductivity of a material with thickness τ in between two materials with impedance *Z*, the boundary conditions can be derived as

$$\boldsymbol{J} = \sigma\boldsymbol{E} \tag{2-18}$$

$$\boldsymbol{J}_s = \tau\boldsymbol{J} \tag{2-19}$$

$$\boldsymbol{E} = \frac{\boldsymbol{J}_s}{\tau\sigma} = Z\frac{\boldsymbol{J}}{\sigma} \tag{2-20}$$

Through these, the fields above and below may be determined as

$$\hat{n}\times[\hat{n}\times(\boldsymbol{E}_1 + \boldsymbol{E}_2)] = -2Z\frac{\boldsymbol{J}_s}{\sigma} \tag{2-21}$$

$$\hat{n}\times(\boldsymbol{E}_2 - \boldsymbol{E}_1) = 0 \tag{2-22}$$

Surface Equivalence

For a numerical representation of an electromagnetic system to be valid, it must be understood that there is an equivalence between the system and the model of that system. To justify this, one must consider whether a few properties are valid within the context of Maxwell's

equations. That is, whether the solutions to Maxwell's equations are unique and linear for a particular electromagnetic system consisting of interactions between media and whether, given unique, linear solutions, one may consider two sources within a region to be equivalent given that the fields generated from them are equivalent.

Uniqueness

To begin, the uniqueness of the solutions to Maxwell's equations will be considered. Consider a generic, lossy with material properties ϵ and μ in which electric and magnetic sources, $\boldsymbol{J}_s$ and $\boldsymbol{M}_s$ respectively, are generating fields. If Maxwell's equations produce solutions that are not unique, one may consider two sets of fields generated: the first set called $\boldsymbol{E}_a$ and $\boldsymbol{H}_a$ as well as a second set called $\boldsymbol{E}_b$ and $\boldsymbol{H}_b$. All solutions must satisfy Maxwell's equations. Thus,

$$-\nabla\times\boldsymbol{E}_a = \boldsymbol{M} + j\omega\mu\boldsymbol{H}_a \qquad \nabla\times\boldsymbol{H}_a = \boldsymbol{J} + j\omega\epsilon\boldsymbol{E}_a \tag{2-23}$$

$$-\nabla\times\boldsymbol{E}_b = \boldsymbol{M} + j\omega\mu\boldsymbol{H}_b \qquad \nabla\times\boldsymbol{H}_b = \boldsymbol{J} + j\omega\epsilon\boldsymbol{E}_b \tag{2-24}$$

Subtracting (2-24) from (2-23), arrives at

$$-\nabla\times(\boldsymbol{E}_a - \boldsymbol{E}_b) = j\omega\mu(\boldsymbol{H}_a - \boldsymbol{H}_b) \qquad -\nabla\times(\boldsymbol{H}_a - \boldsymbol{H}_b) = (\sigma + j\omega\epsilon)(\boldsymbol{E}_a - \boldsymbol{E}_b) \tag{2-25}$$

which may be rewritten as

$$-\nabla\times\delta\boldsymbol{E} = j\omega\mu\delta\boldsymbol{H} = \delta\boldsymbol{M}_t \qquad -\nabla\times\delta\boldsymbol{H} = j\omega\epsilon\delta\boldsymbol{E} = \delta\boldsymbol{J}_t \tag{2-26}$$

where $\delta\boldsymbol{E}$ and $\delta\boldsymbol{H}$ are the differences in the electric and magnetic fields and $\delta\boldsymbol{M}_t$ and $\delta\boldsymbol{J}_t$ are the total differences in the magnetic and electric current densities. For the solutions to Maxwell's equations to be unique, these values must all be zero implying that $\boldsymbol{E}_a = \boldsymbol{E}_b$ and $\boldsymbol{H}_a = \boldsymbol{H}_b$.

Conservation of energy suggests that this is the case. This can be shown by using the electromagnetic conservation of energy equation.

$$\oiint_S (\delta \boldsymbol{E} \times \delta \boldsymbol{H}^*) \cdot d\boldsymbol{s} + \iiint_V [(\sigma + j\omega\epsilon)^* |\delta \boldsymbol{E}|^2 + (j\omega\mu)|\delta \boldsymbol{H}|^2] dv = 0 \qquad (2\text{-}27)$$

The surface integral in (2-27) can be shown to be zero through vector identities when the tangential components of $\boldsymbol{E}$ or $\boldsymbol{H}$ are specified over the boundary. For the volume integral to be equal to zero, it must be that $\delta\boldsymbol{E}$ and $\delta\boldsymbol{H}$ are also zero since σ, ω, ϵ are clearly non-zero system parameters. Thus, the solutions to Maxwell's equations must be unique.

Linearity

To prove linearity, one may consider fields generated by two sources $\boldsymbol{J}_1$ and $\boldsymbol{J}_2$. These fields must satisfy the equations

$$\nabla \times \boldsymbol{E}_1 = -j\omega\mu \boldsymbol{H}_1 \qquad (2\text{-}28)$$

$$\nabla \times \boldsymbol{H}_1 = \boldsymbol{J}_1 + j\omega\epsilon \boldsymbol{E}_1 \qquad (2\text{-}29)$$

for source $\boldsymbol{J}_1$ and

$$\nabla \times \boldsymbol{E}_2 = -j\omega\mu \boldsymbol{H}_2 \qquad (2\text{-}30)$$

$$\nabla \times \boldsymbol{H}_2 = \boldsymbol{J}_2 + j\omega\epsilon \boldsymbol{E}_2 \qquad (2\text{-}31)$$

for source $\boldsymbol{J}_2$.

Given the total fields $\boldsymbol{E}$ and $\boldsymbol{H}$ obtained by adding both systems, it becomes clear that linearity for electromagnetic systems holds. That is,

$$\boldsymbol{E} = \boldsymbol{E}_1 + \boldsymbol{E}_2 \qquad \boldsymbol{H} = \boldsymbol{H}_1 + \boldsymbol{H}_2 \qquad (2\text{-}32)$$

Surface Equivalence

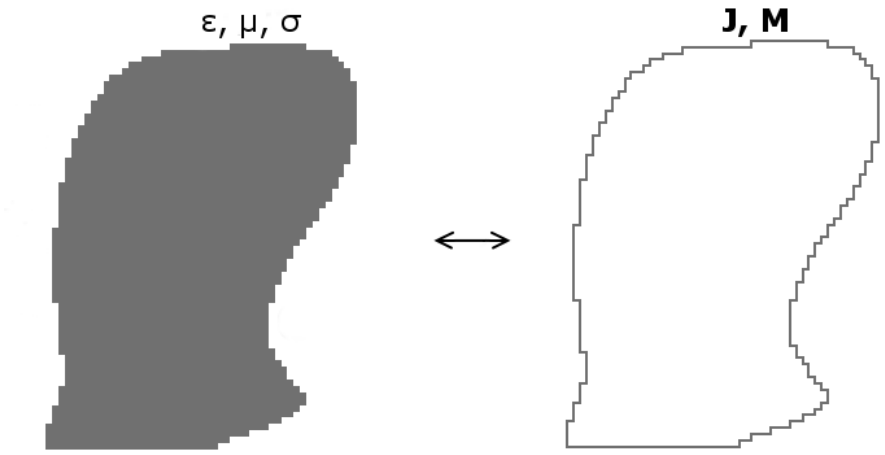


Figure 2-4: A crude representation of the surface equivalence principle, which replaces a full material system with equivalent sources.

With the understanding that most numerical solutions to an electromagnetic system will be obtained not with the real, physical system but instead with a meshed approximation of that system, it is important to show that one may derive an equivalent source whose solution matches the unique solution of the system being modeled. The surface equivalence principle states that given electromagnetic sources, $\boldsymbol{E}$ and $\boldsymbol{H}$, within a region, sources $\boldsymbol{E}_e$ and $\boldsymbol{H}_e$ which produce the same field within that region are said to be equivalent. This can be seen in Figure 2-4.

This provides a distinct advantage over the use of the real system in solving an electromagnetic problem. Oftentimes in electromagnetic systems, the solution to the radiated or scattered fields may be difficult or impossible to solve analytically using known integration techniques. While such solutions exist, such as for a current flowing through a dipole or loop, in general, the problems that one might consider do not always fall neatly into these known cases. In this situation, it becomes convenient to decompose the system into a series of equivalent elements, which can then be solved more easily through a known or numerical method, instead of solving the system directly.

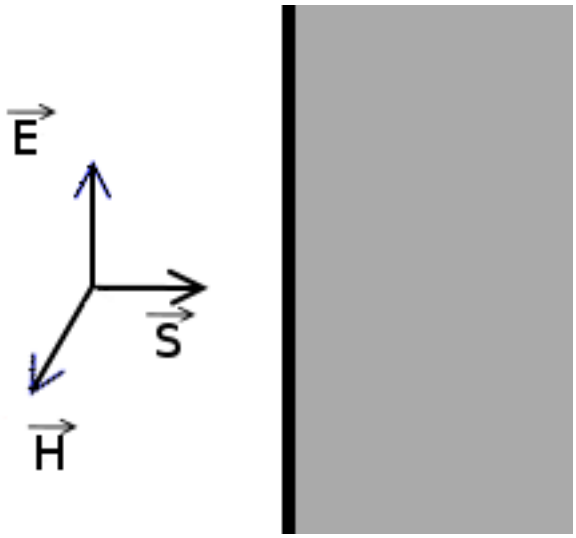


Figure 2-5: A simple diagram of an electromagnetic interaction with a boundary.

Given fields $\boldsymbol{E}_i$ and $\boldsymbol{H}_i$ within the boundary of the surface and $\boldsymbol{E}_o$ and $\boldsymbol{H}_o$ in Figure 2-5, the equivalent sources can be determined from the boundary conditions easily as

$$\boldsymbol{J}_e = \hat{n}\times(\boldsymbol{H}_o - \boldsymbol{H}_i) \tag{2-33}$$

$$\boldsymbol{M}_s = -\hat{n}\times(\boldsymbol{E}_o - \boldsymbol{E}_i) \tag{2-34}$$

Using these equivalent surface currents, the radiation equations can be used to derive the electric field integral equation (EFIE) and magnetic field integral equation (MFIE) [7]

$$\boldsymbol{E}(r) = -\oiint_S \nabla\times\overline{\overline{G}}\,(R)\cdot\hat{n}'\times\boldsymbol{E}(\boldsymbol{r}')dS' + jk_0Z\oiint_S \overline{\overline{G}}\,(R)\cdot\hat{n}'\times\boldsymbol{H}(\boldsymbol{r}')dS' \tag{2-35}$$

$$\boldsymbol{H}(r) = -\oiint_S \nabla\times\overline{\overline{G}}\,(R)\cdot\hat{n}'\times\boldsymbol{H}(\boldsymbol{r}')dS' - \frac{jk_0}{Z}\oiint_S \overline{\overline{G}}\,(R)\cdot\hat{n}'\times\boldsymbol{E}(\boldsymbol{r}')dS' \tag{2-36}$$

where $R = |\boldsymbol{r} - \boldsymbol{r}'|$, $\boldsymbol{r}$ and $\boldsymbol{r}'$ are the observation and integration points, $\hat{n}'$ is the unit normal at the integration point, and $\overline{\overline{G}}$ is the dyadic Green's function which may be written as

$$\overline{\overline{G}} = -\left(\overline{\overline{I}} + \frac{\nabla\nabla}{k_0^2}\right) G_0(R) \qquad (2\text{-}37)$$

where $\overline{\overline{I}}$ is the unit dyad and the scalar Green's function is given as

$$G_0(R) = \frac{e^{-jk_0R}}{4\pi R} \qquad (2\text{-}38)$$

Using (2-35) and (2-36), the surface equivalent currents, and vector identities, the integral equations, which may be used in a numerical solution to an electromagnetic system, may be derived. These expressions can be written as,

$$\boldsymbol{E} = \oiint_S \Big\{\boldsymbol{M}(\boldsymbol{r}')\times\nabla G_0(\boldsymbol{r},\boldsymbol{r}') - jk_0Z\boldsymbol{J}(\boldsymbol{r}')G_0(\boldsymbol{r},\boldsymbol{r}') - j\frac{Z}{k_0}[\nabla'\cdot\boldsymbol{J}(\boldsymbol{r}')]\nabla G_0(\boldsymbol{r},\boldsymbol{r}')\Big\}dS' \qquad (2\text{-}39)$$

$$\boldsymbol{H} = \oiint_S \Big\{-\boldsymbol{J}(\boldsymbol{r}')\times\nabla G_0(\boldsymbol{r},\boldsymbol{r}') + j\frac{k_0}{Z}\boldsymbol{M}(\boldsymbol{r}')G_0(\boldsymbol{r},\boldsymbol{r}') + j\frac{1}{k_0Z}[\nabla'\cdot\boldsymbol{M}(\boldsymbol{r}')]\nabla G_0(\boldsymbol{r},\boldsymbol{r}')\Big\}dS' \qquad (2\text{-}40)$$

The Finite Element Method

The finite element method has a long history in engineering due to its usefulness in numerically solving integral equations, such as in equations (2-39) and (2-40). While its origins are uncertain, Courant is believed to be among the first to describe the discretization of partial differential equations (PDEs) into a representation that is similar to the finite element method [8] although his work built upon developments by other researchers. The finite element method began to mature and see heavy development as computer technology progressed and aviation

engineers began using it in their models [9, 10]. These advances, no doubt, was encouraged by the development of the bipolar junction transistor, integrated circuits and early computer technology.

The finite element method is attractive due to its ability to handle the discretization of complex geometry consisting of dissimilar materials. One may also use the finite element method to determine localized responses allowing one to not only see the global effects, such as in electromagnetic scattering, but also the fields locally on an element which have been used to determine that scattering. The finite element method has the distinct advantage of generating sparse matrices, which can result in a typically low memory requirement.

A disadvantage of the finite element method in a boundary-value problem is that a method must be used to enclose the system. When paired with an appropriate boundary condition, an electromagnetic scattering or radiation problem can be successfully applied to a finite element formulation. A common technique to do this is to use an absorbing boundary condition or perfectly matched layer (PML) which can then truncate the system. This technique, however, requires the discretization of much of the half-space - that is, the whole of space above or below the system - such that the wave not be distorted by near-field effects. Furthermore, an improperly matched layer may create additional error in the finite-element system by inducing non-existent scattering along the boundaries.

Another technique is to pair the finite element method with a hybridization technique coupling the fields with the results of a boundary-integral method. While boundary-integral systems typically require fully populated matrices, these methods, which only treat the boundaries two-dimensionally, require much less unknowns than a three-dimensional finite-element system. When integrated with acceleration techniques, this process becomes more attractive than the use of a fully meshed finite element system coupled with a matching technique like PML.

Discretization of a Domain into a Finite Element Mesh

The finite element method achieves its ability to represent complex geometries through the process called meshing or discretization. Typically, a geometry will be discretized into elements with as few edges or nodes as possible. In two-dimensions, the geometry is typically discretized into a triangular or quadrilateral mesh. In three dimensions, the most convenient representations are into eight-node brick elements, six-node prismatic elements, or five node tetrahedral elements. Each three-dimensional representation may represent an increasing degree of complexity.

A simple mesh may be seen in Figure 2-6. In this example, a two-dimensional circular material is represented. This simple scenario demonstrates the benefit of using a mesh, which is constructed from elements with fewer nodes, i.e. in two-dimensions a triangle. In order for the quadrilateral mesh to approximate a circle, many more elements are necessary. Conversely, a mesh that represents an object that is restricted to quadrilateral objects would be better-represented using quadrilateral elements since it requires twice as many triangles to build a mesh of a given fineness.

For generality, it is best to use a mesh that is built from elements with the least amount of edges necessary to represent the any object in the dimensionality in which variations are desired. That is, one should use triangles for two-dimensional objects, prisms for layers of two-dimensional objects, and tetrahedrons for three-dimensional objects. In this research, only layered two-dimensional objects will be considered. Thus, prismatic elements are a logical choice.

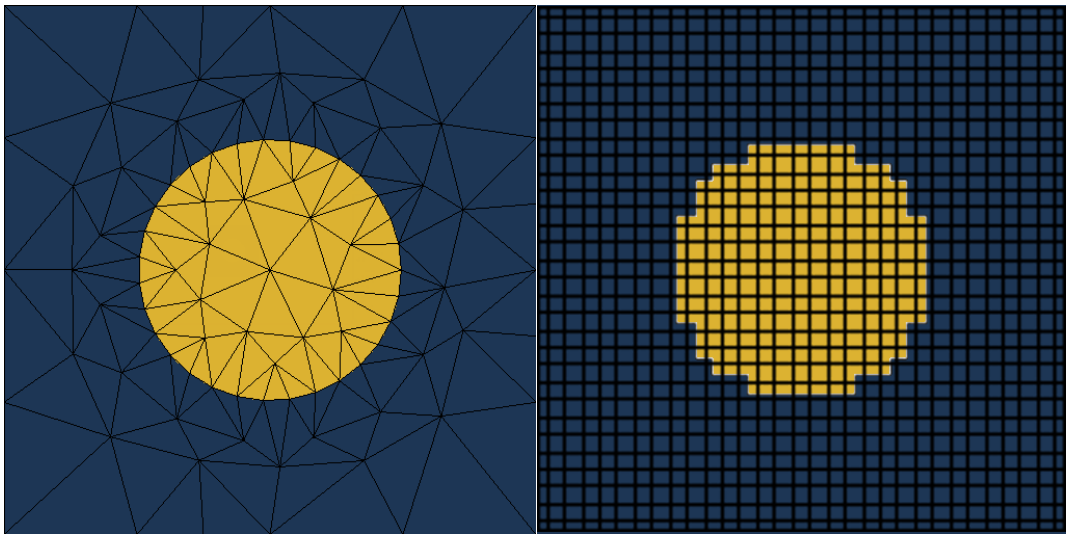


Figure 2-6: Example meshes representing a circle. On the left, a triangular mesh is used requiring 170 triangles. On the right, the mesh is built from square pixels using 900 pixels. The edges are drawn for emphasis.

The finite element process is based on the idea that at some level of fineness, the wave response may be approximated by some other function, which is of a smaller order along an element. The simplest case occurs when the wave can be considered linear along the element. In general, this holds when the element is less than *1/10* of the wavelength within the material. This allows the integral equations to be recast into a set of weighted averages, which can be represented by a matrix

$$[Z][J] = [E] \qquad (2\text{-}41)$$

where Z is a matrix containing the finite element impedance functions, E is the excitation and J is the induced currents which are unknown. Typically, this system of equations is large, sparse and symmetric. For this reason, the solution lends itself to a specialized algorithm such as the biconjugate gradient instead of direct matrix inversion, which is computationally intensive and requires a large amount of memory.

The Weak Form of a Differential Equation

The key to setting up a finite element problem in a way that a computer can easily solve is by expanding the integral equation into what is known as its weak form. In other words, the wave equation will be solved in a way that is valid over some test domain, in this case along the mesh.

To begin, consider the general form of the wave equation for an electric field

$$\nabla\times(\nabla\times\boldsymbol{E}) - k^2\boldsymbol{E} = -jk_0Z_0\mu\boldsymbol{J} - \nabla\times\boldsymbol{M} \qquad (2\text{-}42)$$

In this example, the sources and material parameters are known qualities but the field is unknown. Rather than solve the fields analytically, the finite element method uses what is known as the weighted residual method to minimize the difference between an approximation of those fields and the fields, as they would exist in reality.

First, recast the equation into the residual form

$$\nabla\times\left(\nabla\times\boldsymbol{E}^{int}\right) - k^2\boldsymbol{E}^{int} + jk_0Z_0\mu\boldsymbol{J} + \nabla\times\boldsymbol{M} = \boldsymbol{R} \qquad (2\text{-}43)$$

where $\boldsymbol{E}^{int}$ is the finite element approximation of the electric field inside the mesh. By subdividing the problem space into a mesh, the variations become small enough that the field on the ith element may be approximated by some basis function $\boldsymbol{W}_i$ such that

$$\int_V \boldsymbol{R}\cdot\boldsymbol{W}_i dV = 0 \qquad (2\text{-}44)$$

From this, the finite element equations can be formed. By expanding the weighted residual equation (2-44) by inserting (2-43) into $\boldsymbol{R}$, it may be written fully as

$$\int_V \nabla\times\left(\nabla\times\frac{\boldsymbol{E}^{int}}{\mu_r}\right)\cdot\boldsymbol{W}_i dV - k_0^2\int_V \epsilon_r\boldsymbol{E}^{int}\cdot\boldsymbol{W}_i dV \qquad (2\text{-}45)$$

$$= -jk_0Z_0\int_V \boldsymbol{J}^i\cdot\boldsymbol{W}_i dV - \int_V \nabla\times\frac{\boldsymbol{M}^i}{\mu_r}\cdot\boldsymbol{W}_i dV$$

where the material parameters, ϵ_r and μ_r, have been extracted to emphasize their contribution to the field localized within the element. Since the right-hand side of (2-45) contains only known quantities, it may be simplified as

$$-jk_0Z_0\int_V \boldsymbol{J}^i\cdot\boldsymbol{W}_i dV-\int_V \nabla\times\frac{\boldsymbol{M}^i}{\mu_r}\cdot\boldsymbol{W}_i dV=f_i \qquad (2\text{-}46)$$

Furthermore, the left-hand side may be simplified using integration by parts and invoking Green's theorem [7] arriving at the weak formulation.

$$\int_V \frac{\nabla\times\boldsymbol{E}^{int}\cdot\nabla\times\boldsymbol{W}_i}{\mu_r}dV \qquad (2\text{-}47)$$

$$-k_0^2\int_V \epsilon_r\boldsymbol{E}^{int}\cdot\boldsymbol{W}_i dV$$

$$-jk_0Z_0\oint_S \hat{n}\times\boldsymbol{H}^{int}\cdot\boldsymbol{W}_i dS=f_i$$

This fully formed weak formulation may be discretized allowing a computer to solve for the electric field provided the magnetic field, $\boldsymbol{H}^{int}$, may be solved for using a boundary condition which couples the internal and external magnetic fields.

Discretization of the Weak Formulation

The standard method to discretize integral equations such as the one in (2-47) is to introduce a discrete formulation for the unknown fields, in this case $\boldsymbol{E}^{int}$ and $\boldsymbol{H}^{int}$, and to choose a weighting function which will yield an approximation of the real fields.

To begin, it will be noted that for a fine meshing, that is one in which follows the rule of thumb that the elements are smaller than a tenth of the wavelength, it is appropriate to approximate the fields along those elements as a constant value multiplied by a basis function, which varies within the element. This effectively recasts the total fields as a piecewise linear

function over the entire domain. The benefit of this technique is it removes variability from the fields and places it in a function over which the variation is known a priori. This allows the unknown fields to be removed from the integral limiting integration to known, solvable quantities only. Therefore, the total field -- be it electric, magnetic or other – may be represented as the sum of all the basis functions over the entire domain.

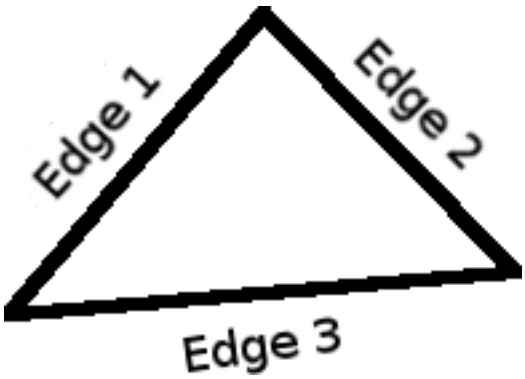


Figure 2-7: Triangle element for use with triangle edge elements.

As an equation, one may write

$$U(\boldsymbol{r}) = \sum_{e=1}^{N_e} \sum_{i=1}^{N_i} U_i^e N_i^e(\boldsymbol{r}) \qquad (2\text{-}48)$$

where $U(\boldsymbol{r})$ is some arbitrary, unknown field, U_i^e is a constant approximation of that field on the *i*th piece of the *e*th element. $N_i^e(\boldsymbol{r})$ is a basis function chosen from the type of element and subdivided pieces, typically nodes or edges, into which the domain has been discretized. N_i is the number of pieces per element, and N_e is the total number of elements into which the domain has been decomposed.

A simple method for choosing a basis function is to assume that at the size of the discretization the change in fields from one node to another can be approximated as a low order polynomial. For a first order basis function applied to a one-dimensional finite element problem, this may be written as

$$N_i(x) = \frac{x - x_{k-1}}{x_k - x_{k-1}} \quad (2\text{-}49)$$

Whereas for a triangular element, such as the one shown in Figure 2-7, the basis function becomes slightly more complicated. Assuming any point on the triangle may be described by the functions

$$x = x_1 + (x_2 - x_1)\xi + (x_3 - x_1)\upsilon \quad (2\text{-}50)$$

$$y = y_1 + (y_2 - y_1)\xi + (y_3 - y_1)\upsilon \quad (2\text{-}51)$$

where x_1, x_2, and x_3 are the x coordinates of the triangle nodes, y_1, y_2, and y_3 are the y coordinates of the triangle nodes, and ξ and υ are local coordinates within the triangle ranging from 0 to 1. From this, a first order shape function may be derived as

$$u(\xi, \upsilon) = c_1 + c_2\xi + c_3\upsilon \quad (2\text{-}52)$$

from which the triangle bases can be derived as

$$N_1 = 1 - \xi - \upsilon \quad (2\text{-}53)$$

$$N_2 = \xi \quad (2\text{-}54)$$

$$N_3 = \upsilon \quad (2\text{-}55)$$

The designer has the freedom to decide on the type of basis function to be used. In this case, a shape function that varies linearly along the edges of the triangle has been described. One may also choose bases that vary at a higher order at a cost of higher computational complexity. Other common bases are based on the node locations rather than on the variations along the edges between nodes although this may come at a cost to accuracy [11].

Figure 2-8: A single prismatic element.

For this work, the finite element code uses prismatic edge elements, such as the one shown in Figure 2-8, with basis functions as described by Volakis and reproduced below in equations (2-56) through (2-63)[7]. In these equations, $\boldsymbol{L}_i^e$ are triangular area basis. Here, the triangles are described as Δ_{ijk} where i, j, and k are arbitrary triangle nodes and the basis functions are calculated at a point, P, within the triangle. The parameter s is a normalization parameter that is zero at the bottom and unity at the top. Finally, the basis functions $\boldsymbol{v}$ are used to describe the vertical edges along the prism and are specifically designed to minimize tangential discontinuities between the edge elements.

$$\boldsymbol{W}_k^e = N_{ij}^e = b_{ij}\left(\boldsymbol{L}_i^e \nabla \boldsymbol{L}_j^e - \boldsymbol{L}_j^e \nabla \boldsymbol{L}_i^e\right)s, \qquad i,j = 1,2,3; k = 1,2,3 \tag{2-56}$$

$$\boldsymbol{W}_k^e = N_{ij}^e = b_{ij}\left(\boldsymbol{L}_i^e \nabla \boldsymbol{L}_j^e - \boldsymbol{L}_j^e \nabla \boldsymbol{L}_i^e\right)(1-s), \qquad i,j = 4,5,6; k = 4,5,6 \tag{2-57}$$

$$\boldsymbol{W}_k^e = \widehat{\boldsymbol{v}}_i(\xi,\upsilon)\boldsymbol{L}_i^e(\xi,\upsilon), \qquad i,j = 1,2,3; k = 7,8,9 \tag{2-58}$$

$$\boldsymbol{L}_1^e = \frac{Area(\Delta_{P23})}{Area(\Delta_{123})} \tag{2-59}$$

$$\boldsymbol{L}_2^e = \frac{Area(\Delta_{P31})}{Area(\Delta_{123})} \tag{2-60}$$

$$\boldsymbol{L}_3^e = \frac{Area(\Delta_{P12})}{Area(\Delta_{123})} \tag{2-61}$$

$$\boldsymbol{v}(\xi,\upsilon) = \sum_{i=1}^{3} \boldsymbol{L}_i^e(\xi,\upsilon)\widehat{\boldsymbol{v}}_i \qquad (2\text{-}62)$$

$$\widehat{\boldsymbol{v}}(\xi,\upsilon) = \frac{\boldsymbol{v}(\xi,\upsilon)}{|\boldsymbol{v}(\xi,\upsilon)|} \qquad (2\text{-}63)$$

These prismatic elements are a trade-off between cubic elements, which produce few unknowns while sacrificing generalization of geometries, and tetrahedral elements, which produce many unknowns but can describe virtually any geometry. With a prismatic element fewer unknowns are required than in a tetrahedral mesh but generalization of a geometry is maintained within a two-dimensional plain making it ideal for the simulation and analysis of a frequency selective surface, patch antenna or other similar devices. Furthermore, meshing is simpler as fewer constraints are required since they only need to be done within the plane of the surface and coupling with the boundaries is automatically enforced.

Regardless of the type of meshing or basis functions used in the discretization of a system, ultimately the basis functions will be applied to the finite element approximation given in (2-48) to build a linear system of the form

$$[A][x] = [b] \qquad (2\text{-}64)$$

Where $[A]$ is a sparse matrix built from the basis equations, $[x]$ are the unknowns seen in (2-48) as U_i^e, and $[b]$ is a vector built from the excitations, such as an incident plane wave, or boundary conditions. As a sparse system, it is well suited for use with an iterative solver such as the biconjugate gradient, instead of a traditional solving algorithm like pure inversion or LU decomposition.

Typically, a finite element problem can be solved through the steps outlines below.

- Apply a meshing algorithm to decompose the system into one to which basis functions may be applied.

- Build the finite element matrix by applying basis functions to the weak form of the PDE.
- Apply boundary conditions to limit the domain of the problem.
- Solve using an iterative method.

Having discussed the discretization of the discretization of the weak form of a differential equation it is appropriate to move onto the boundary conditions leaving meshing and iterative methods to a later chapter since they do not directly relate to the finite element method.

Boundary Conditions in the Finite Element Method

A robust boundary condition serves two purposes in a finite element code. First, it truncates the mesh to a domain limiting the area over which is necessary to compute the fields. Second, the boundary conditions provide a basis from which a unique solution to the partial differential equations may be determined. While in some situations, such as within a waveguide or resonance chamber, the boundary conditions are well understood, oftentimes it is in the interest of an engineer to determine the scattering or radiation responses of fields propagating into free-space or far enough away that one may consider that to be an accurate approximation. To eliminate the necessity of meshing unnecessary space, techniques have been developed which truncate the domain to within the bounds of an artificial boundary. In this analysis, three methods will be discussed which may be applied to a finite element problem to truncate the boundaries by minimizing artificial reflections: absorbing boundary conditions, perfectly matched layer, and a hybrid boundary integral.

Absorbing Boundary Conditions

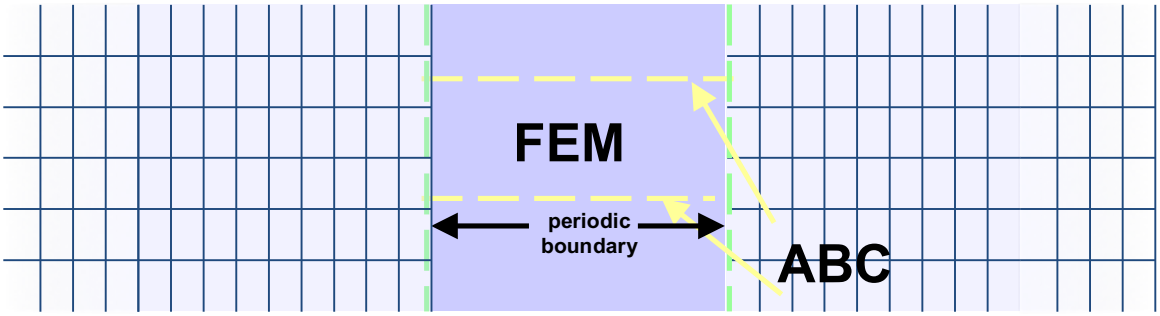


Figure 2-9: A schematic arrangement for an absorbing boundary condition.

Absorbing boundary conditions were an early method of truncating the finite element domain. These methods are all based on the idea that within the simulation domain all scattered or radiated fields, which propagate away from the material being considered, is equivalent to lost energy. Therefore, no energy incident upon the boundary is perturbed by the boundary. Mathematically, this is equivalent to simulating the effects of an anechoic chamber. A problem with these methods, such as those based on Enquist and Majda [12] or the Bayliss-Turkel boundary conditions [13], is that they are sensitive and tend to be problem specific.

Consider the situation seen in Figure 2-9 in which an arbitrary field, U is incident upon the artificial boundary. A simple artificial boundary condition obeys the following condition.

$$\frac{dU}{dn} + \alpha U = 0 \qquad (2\text{-}65)$$

In other words, the change in the field in the normal direction to the boundary is equal to the field multiplied by some constant.

This formulation is first order and has the obvious problem that it ignores higher order effects. Another drawback is that typically absorbing boundary conditions require the artificial boundary to be placed a few wavelengths away from the object being simulated requiring unnecessarily large meshes and matrices. However, despite this, these boundary conditions are

useful do to their simplicity. An improved absorbing boundary condition takes into account the second derivative as well. This condition may be written as

$$\frac{dU}{dn} = \alpha U + \beta \frac{\partial^2 U}{\partial s^2} \qquad (2\text{-}66)$$

where s represents the field components along the surface of the artificial boundary. In both the first and second order cases, the selection of α and β is problem dependent.

An absorbing boundary condition is discretized similar to the way a normal finite element problem would be discretized. Consider the discretization of a scattering magnetic field,

$$\boldsymbol{H}^{scat} = \sum_{s=1}^{N_s} \sum_{i=1}^{2} H_i^{scat} L_i^e(\boldsymbol{r}) \qquad (2\text{-}67)$$

Such that the N_s is the number of elements on the surface of the artificial boundary and the absorbing boundary integral equation,

$$\int_{C^{bound}} \frac{1}{\epsilon_r} L_j \hat{n} \cdot \nabla \boldsymbol{H}^{scat} ds = \int \frac{1}{\epsilon_r^e} L_j (\alpha \boldsymbol{H}^{scat} + \beta \frac{\partial^2 \boldsymbol{H}^{scat}}{\partial s^2}) ds \qquad (2\text{-}68)$$

may be rewritten using integration by parts and discretized as

$$\int_C \frac{1}{\epsilon_r} L_j \hat{n} \cdot \nabla \boldsymbol{H}^{scat} = \sum_{e=1}^{N_e} \sum_{i=1}^{N_i} \frac{1}{\epsilon_r^s} [\alpha \int_C L_j L_i ds - \beta \int_C \frac{\partial L_j}{\partial s} \frac{\partial L_i}{\partial s} ds \,] \qquad (2\text{-}69)$$

which yields the element equations

$$\sum_{e=1}^{N_e} [A^e][H^{scat,e}] + \sum_{s=1}^{N_s} [B^s][H^{scat,e}] = \sum_{e=1}^{N_e} [b^e] \qquad (2\text{-}70)$$

whose individual elements can be developed based on the problem. These absorbing boundary element equations may be inserted into a full finite element system as

$$\begin{bmatrix} [A^{II}] & [A^{IS}] \\ [A^{SI}] & [A^{SS}] \end{bmatrix} \begin{bmatrix} H^{int} \\ H^{bound} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & [B] \end{bmatrix} \begin{bmatrix} H^{int} \\ H^{bound} \end{bmatrix} = [b] \qquad (2\text{-}71)$$

Perfectly Matched Layer

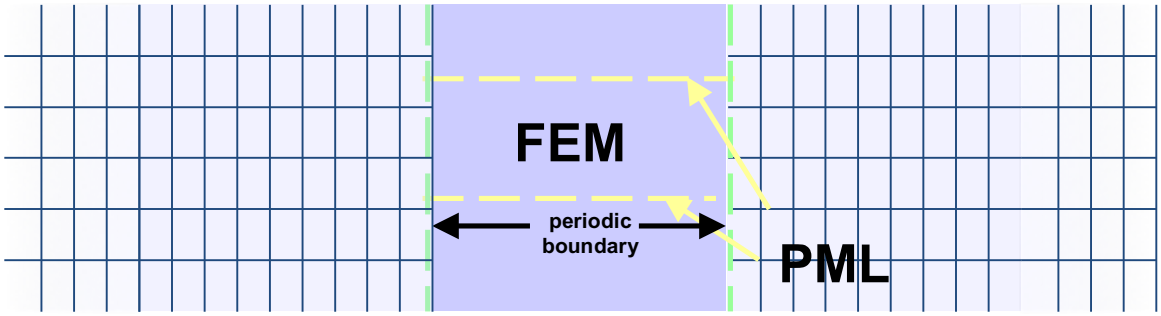


Figure 2-10: A schematic arrangement for a perfectly matched layer.

The perfectly matched layer was designed to solve many of the problems that existed in the traditional absorbing boundary conditions. Like the absorbing boundary conditions, the perfectly matched layer is designed to take advantage of the mathematical equivalency of a field propagating to infinite and the field being completely absorbed. The difference is that in a domain truncated by an absorbing boundary condition, aside from ideal situations, there is no guarantee that reflections will be eliminated. The absorbing boundary condition maximized absorption in the partial differential equations at the boundary whereas the perfectly matched layer minimized reflection which encouraging absorption in the material of the artificial boundary.

Originally proposed by Beringer [14], the perfectly matched layer is built upon the insight that within the layer Maxwell's equations must also be valid. Thus, by proposing a layer in which the impedances match the impedance from which the fields arrive, no field is reflected. Thus, the code is free to absorb the field over a few layers eliminating undesirable reflections.

This method is commonly used due to its robustness and simplicity. A major drawback however is that the layers must be meshed just as the simulation domain itself causing a dramatic increase in the memory necessary to solve the problem. Furthermore, these absorbing boundary methods are valid for continuous problems but, in a computer simulation, the problem is

discretized. Therefore, it is possible that with improper meshing inaccuracies or convergence issues will occur.

As an example, consider the scattering problem seen in Figure 2-10. The numerical system in a perfectly matched layer problem is a special case of the absorbing boundary condition in (2-71) except that $B = 0$ and the relative permittivity and permeability are tensors that, for a wave traveling in the $\hat{z}$ direction, may be set as

$$\bar{\bar{\epsilon}} = \bar{\bar{\mu}} = \begin{bmatrix} c & 0 & 0 \\ 0 & c & 0 \\ 0 & 0 & 1/c \end{bmatrix} \qquad (2\text{-}72)$$

where c is a complex constant used to maximize absorption and minimize reflection.

The Boundary Integral as a Boundary Condition

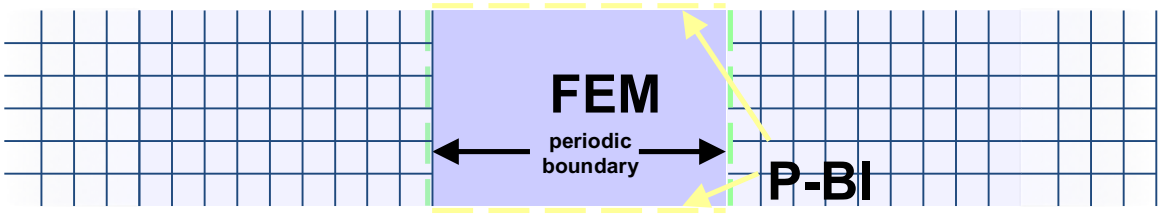


Figure 2-11: A schematic image of the periodic finite element boundary integral problem.

The boundary integral approach to handling boundary conditions is radically different from the absorbing boundary conditions described above. Rather than approximate the effect of an incident wave upon an absorbing boundary, the boundary integral method calculates an exact solution of the partial differential equations on the boundary. This can be seen graphically in Figure 2-11 which demonstrates the regional nature of a finite element boundary integral system.

To do this, one may use Kirchhoff's boundary integral with respect to the magnetic field using the following expressions [15, 16].

$$H_z = H_z^i + H_z^{scat} \tag{2-73}$$

$$H_z(\boldsymbol{r})|_{\boldsymbol{r}\in C^{outer}} \tag{2-74}$$

$$= H_z^i - \oint_C \left\{ \frac{\partial H_z(\boldsymbol{r}')}{\partial n'} G^{2D}(\boldsymbol{r},\boldsymbol{r}') - [\hat{n}' \cdot \nabla' G^{2D}(\boldsymbol{r},\boldsymbol{r}')] H_z(\boldsymbol{r}') \right\} dl'$$

$$G^{2D}(\boldsymbol{r},\boldsymbol{r}') = -\frac{j}{4} H_0^{(2)}(k_0|\boldsymbol{r}-\boldsymbol{r}'|) \tag{2-75}$$

where $G^{2D}(\boldsymbol{r},\boldsymbol{r}')$ is a Green's function and $H_0^{(2)}(k_0|\boldsymbol{r}-\boldsymbol{r}'|)$ is a second order Hankel function. The boundary integral in (2-74) may be discretized using Galerkin's method and approximated in a similar way to what is done in the finite element method. First, for simplicity, let the partial derivative of the magnetic field be rewritten as

$$\psi(\boldsymbol{r}) = \frac{\partial H(\boldsymbol{r})}{dn} \tag{2-76}$$

A convenient way to discretize the unknowns, $H_z(\boldsymbol{r})$ and $\psi(\boldsymbol{r})$ is to use a piecewise constant expansion in which the fields are represented by their average value within the element. This may be written as

$$\psi = \sum_{s=1}^{N_s} \frac{(\psi_1^s + \psi_2^s)}{2} \Pi(\theta - \theta_s) \tag{2-77}$$

$$H_z = \sum_{s=1}^{N_s} \frac{\left(H_{z_1}^s + H_{z_2}^s\right)}{2} \Pi(\theta - \theta_s) \tag{2-78}$$

where $\Pi(\theta - \theta_s)$ is a pulse function that is one within the element and zero otherwise and ψ_1^s, ψ_2^s, $H_{z_1}^s$, and $H_{z_2}^s$ are the values at the bounds of the elements. This simple approximation can be inputted into the boundary integral equation yielding a residual equation, which will help in forming the boundary integral numerical system. Inputting (2-77) and (2-78) into (2-74) and rearranging yields

$$H_z(\boldsymbol{r}) + \sum_{s=1}^{N_s}[\frac{\psi_1^s + \psi_2^s}{2}\int_{C_s} G^{2D}(\boldsymbol{r},\boldsymbol{r}')dl' - \frac{\boldsymbol{H}_1^s + \boldsymbol{H}_2^s}{2}\int_{C_s}\hat{n}'\cdot\nabla' G^{2D}(\boldsymbol{r},\boldsymbol{r}')dl'] - H_z^i(\boldsymbol{r}) = R(\boldsymbol{r}) \quad (2\text{-}79)$$

Using the weighted residual method with $W(\boldsymbol{r}) = \delta(\phi - \phi_u)$, where the subscript u represents the test scenario at an unknown, yields

$$\int_C R(\boldsymbol{r})W(\boldsymbol{r})dl = 0 \quad (2\text{-}80)$$

which will allow a system of equations over each element to be build using for a boundary integral system with test pointed at the center of each element. These can be written in the form of the following equations.

$$\frac{\boldsymbol{H}_{z1}^{u1} + \boldsymbol{H}_{z2}^{u1}}{2} + \sum_{s=1}^{N_s}\left(\frac{\psi_1^s + \psi_2^s}{2}G_{s1,u1} - \frac{\boldsymbol{H}_{z1}^{s1} + \boldsymbol{H}_{z2}^{s2}}{2}G_{\nabla_{s1,u1}}\right) = \boldsymbol{H}_z^i(\boldsymbol{r}_{u1}), \quad (2\text{-}81)$$

$$u_1 = 1,2,\dots,N_s$$

$$G_{s1,u1} = -\frac{jr_C}{4}\int_{\phi_{s1}-\frac{\Delta\phi}{2}}^{\phi_{s1}+\frac{\Delta\phi}{2}} H_0^{(2)}\left(\left|2k_0 r_C \sin\left(\frac{\phi_{u1}-\phi'}{2}\right)\right|\right)d\phi' \quad (2\text{-}82)$$

$$G_{\nabla_{s1,u1}} = -\frac{d}{dn}\int_{\phi_{s1}-\frac{\Delta\phi}{2}}^{\phi_{s1}+\frac{\Delta\phi}{2}} G^{2D}(\boldsymbol{r},\boldsymbol{r}')|_{r=r_{s1}} \quad (2\text{-}83)$$

Such that (2-81) forms the basis for the boundary integral portion of the system of equations while (2-82) and (2-83) are Green's function integrals, which must be calculated.

With the boundary integral recast in the form of a linear system, it may now be used in a full finite element problem as in the previous absorbing boundary condition examples as

$$\begin{bmatrix}[A^{II}] & [A^{IS}]\\ [A^{SI}] & [A^{SS}]\end{bmatrix}\begin{bmatrix}H^{int}\\ H^{bound}\end{bmatrix} + \begin{bmatrix}0 & 0\\ 0 & [BI]\end{bmatrix}\begin{bmatrix}H^{int}\\ H^{bound}\end{bmatrix} = [b] \quad (2\text{-}84)$$

where $[BI]$ is the boundary integral system.

Finally, it is important to consider coupling the finite element and boundary integral systems. Doing this will allow the fields determined from the boundary integral to be continuous into the finite element region guaranteeing an accurate solution for a given excitation. The complete finite-element boundary-integral equations are

$$\int_V \frac{\nabla\times\boldsymbol{E}^{int}\cdot\nabla\times\boldsymbol{W}_i}{\mu_r}dV \qquad (2\text{-}85)$$

$$-k_0^2\int_V \epsilon_r\boldsymbol{E}^{int}\cdot\boldsymbol{W}_i dV - jk_0Z_0\oint_S \hat{n}\times\boldsymbol{H}^{int}\cdot\boldsymbol{W}_i dS = f_i^{int}$$

$$-\frac{1}{2}\oint_S \left[\boldsymbol{Q}_i\cdot\left(\hat{n}\times\boldsymbol{H}^{int}\right)\right]dS - \oint_S\oint_S \boldsymbol{Q}_i\cdot\left[\hat{n}\times\nabla\times\overline{\overline{G}}\times\hat{n}'\right]\cdot\boldsymbol{H}^{int}dS'dS \qquad (2\text{-}86)$$

$$-jk_0Y_0\oint_S\oint_S \boldsymbol{Q}_i\cdot\left[\hat{n}\times\nabla\times\overline{\overline{G}}\times\hat{n}'\right]\cdot\boldsymbol{E}^{ext}dS'dS = f_i^{ext}$$

where the internal testing functions are indicated by $\boldsymbol{W}_i$, the external testing functions are indicated by $\boldsymbol{Q}_i$, and the external excitation force, f_i^{ext}, can be determined for each element by applying boundary conditions.

$$f_i^{ext} = -\oint_S \boldsymbol{Q}_i\cdot\hat{n}\times\left[\boldsymbol{H}^{inc}+\boldsymbol{H}^{ref}\right]dS \qquad (2\text{-}87)$$

Finally, the coupling condition can be determined using electric field continuity boundary conditions such that the following equation is satisfied.

$$\oint_S \boldsymbol{Q}_i\cdot\hat{n}\times\left(\boldsymbol{E}^{int}-\boldsymbol{E}^{ext}\right)dS = 0 \qquad (2\text{-}88)$$

With these conditions satisfied, the finite element boundary integral can be discretized and used to solve generalized electromagnetic scattering problems.

This type of formulation should provide improved accuracy over the use a simulated boundary condition at the cost of increased computational complexity. This is summarized in Table 2-1.

Absorbing Boundary Condition	Perfectly Matched Layer	Boundary Integral
Fast	Fast	Slow
Memory intensive	Memory intensive	Low memory requirements
Possible artificial reflects	Artificial reflections less likely	No artificial reflections
Approximate solution	Approximate solution	Exact solution neglecting numerical integration errors

Table 2-1: A summary of three important boundary conditions for use in a finite element system.

Periodic Boundary Conditions

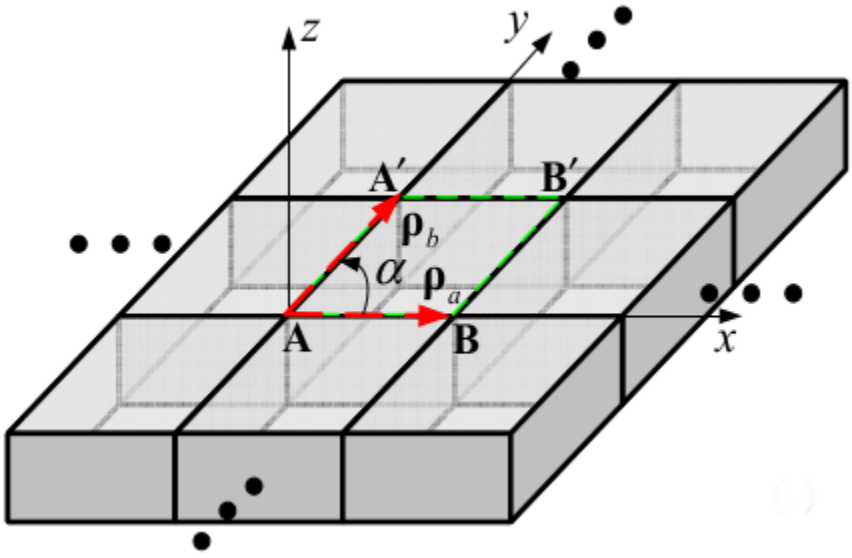


Figure 2-12: An example of two-dimensional periodic unit cells.

The final form of boundary conditions that will be considered are periodic boundary conditions. These conditions are easy to conceptualize and visualize. For example, consider a two-dimensional system such as the one shown in Figure 2-12. In this case, the simulation

domain has been divided into several identical elements that are repeated ad infinitum in the horizontal and vertical directions. While in reality no surface is truly infinitely periodic, this assumption serves as a valid simplification for surfaces in which the unit cells extend several wavelengths from the simulation domain. If a designer is concerned with the field effects near the edges of a device or cannot guarantee that the majority of the energy will be concentrated uniformly within the center of the device, it is recommended that a finite element method using another boundary condition be employed.

Each individual element, known as a unit cell, must match on opposing boundaries in both the physical positioning of the elements and the force generated on the elements by induced fields. This can be conceptualized mathematically as

$$\nabla^m \boldsymbol{E}(x,y) = \nabla^m \boldsymbol{E}(x + i\rho_a, y + j\rho_b); \quad i = 0,1,2,\dots;\ j = 0,1,2,\dots, m = 0,1,2,\dots \tag{2-89}$$

where ρ_a and ρ_b are the periodicities in the x and y directions.

Using a periodic boundary condition not only simplifies the calculation of the fields at the boundaries but also the calculation of the fields within the unit cells as well. In particular, one may take advantage of Ewald summations [17], and periodic Green's functions [18].

An Ewald summation is a method for calculating the long-range electric interactions in periodic electromagnetic systems. In an electromagnetic system, the field interaction between elements occur inversely proportional to the distance between the elements. That is, it is proportional to $\frac{1}{|\boldsymbol{r}-\boldsymbol{r}'|}$. Using a simple summation for such a system will lead to a large calculation that is not guaranteed to converge. The Ewald method divides this interaction into two separate calculations that converge rapidly. Using the error function, one may divide the summation into two complementary sums such that

$$\frac{1}{r} = \frac{\text{erf}\left(\frac{1}{2}\sqrt{\eta}r\right)}{r} + \frac{erfc\left(\frac{1}{2}\sqrt{\eta}r\right)}{r} \quad (2\text{-}90)$$

where the error functions in this situation do not represent any kind of error. Instead, these functions were chosen due to their convergence properties near and far from the interaction. These functions are defined as follows

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt \quad (2\text{-}91)$$

$$erfc(x) = 1 - \text{erf}(x) = \frac{2}{\sqrt{\pi}}\int_x^\infty e^{-t^2}\,dt \quad (2\text{-}92)$$

In this code, rather than use the real-valued error functions, the complex form, also known as the Faddeeva function, will be used, which can be defined as

$$w(z) = e^{-z^2}erfc(-iz) \quad (2\text{-}93)$$

Using the Ewald transformation, the Green's functions in a periodic system may be modified into periodic Green's functions. These functions may be inserted as normal into the derivation of a finite element or boundary integral system. By calculating the Green's functions using an Ewald summation, the contributions to the Green's function from far-off unit cells are simplified into the converged value. For ease of calculation, one may split the Green's function calculations into special cases for the spatial and spectral. These can be expressed mathematically as

$$G_{spatial,periodic} = 0.5\sum_{m,n}[e^{-jkr_{mn}}cerf\left(r_{mn}k - \frac{j}{2}\right) + e^{jkr_{mn}}cerf\left(r_{mn}k + \frac{j}{2}\right)]\frac{e^{k_x\rho_x + k_y\rho_y}}{4\pi r_{mn}} \quad (2\text{-}94)$$

$$G_{spectral,periodic} = \sum_{m,n}\frac{e^{-j(k_{t,x}x + k_{t,y}y)}}{j2\rho_a\rho_b\sin(\gamma)\,k_{sp}}cerf\left(\frac{jk_{sp}}{2}\right) \quad (2\text{-}95)$$

where the Hankel functions of the second kind, used in the general form of a Green's function, have been simplified to exponential functions, the summations over m and n are rotated such that they spiral out from the element which is being calculated and the distance value for the periodic contribution at a point is represented using

$$r_{mn} = \sqrt{(x-\rho_x)^2 + \left(y-\rho_y\right)^2} \quad (2\text{-}96)$$

$$\rho_x = m\rho_a - n\rho_b \cos(\gamma) \quad (2\text{-}97)$$

$$\rho_y = n\rho_b \sin(\gamma) \quad (2\text{-}98)$$

$$k_{sp} = \sqrt{k^2 - k_{t,x}^2 - k_{t,y}^2} \quad (2\text{-}99)$$

where ρ_a and ρ_b are the periodicities in the horizontal and vertical directions of the unit cell, and γ is the angle of the parallelogram which encloses the unit cell.

Calculation of the Scattered Field

Once a finite element system is solved, regardless of the type of boundary conditions used, the solution of the system contains values representing the induced fields or currents at each particular element. This is useful in that it allows an engineer or designer to get a picture of the physical response of a system but it is not directly useful for the sort of thing one might desire from an electromagnetic system. Typically, in an electromagnetic system, what is of interest is instead the scattering or radiation that occurs in that system.

In this research, I have focused on the scattering response of an electromagnetic system from an incident unit-magnitude plane-wave. Using a unit-magnitude incident wave allows a true response to be calculated using the linearity relation by scaling the unit-scattering response by the actual power of an incident wave. The use of plane waves is a valid assumption for most

scattering cases. For example, an incident wave from a spherical source when traveling a far enough distance can be approximated as planar after it has traveled a distance of several wavelengths assuming that the portion of the wave hitting the scattering media is small in comparison. Similarly, a beam from a laser source or exiting a fibre-optic cable typically has a beam pattern, which can be described as Gaussian. If the beam waist is large compared to the size of a unit cell of the scattering object, approximating the incident field as a plane wave may be justified.

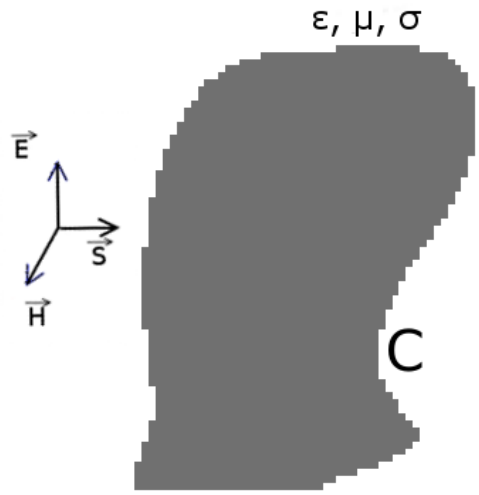


Figure 2-13: An arbitrary object upon which a plane wave is incident.

To calculate the scattering from an incident plane wave for an arbitrary system, such as the one in Figure 2-13, one may solve Kirchhoff's integral equation for an arbitrary wave, $U(\boldsymbol{r})$ usually written as

$$U^{scat}(\boldsymbol{r}) = -\oint_C \{\hat{n}\cdot\nabla'(U(\boldsymbol{r}')G^{2D}(\boldsymbol{r},\boldsymbol{r}') - [\hat{n}'\cdot\nabla' G^{2D}(\boldsymbol{r},\boldsymbol{r}')]U(\boldsymbol{r}')dl' \qquad (2\text{-}100)$$

where U can be the scattered electric or magnetic fields and the Green's function is a zeroth-order Hankel function of the second kind,

$$G^{2D}(\boldsymbol{r},\boldsymbol{r}') = -\frac{j}{4}H_0^{(2)}(k_0|\boldsymbol{r}-\boldsymbol{r}'|) \qquad (2\text{-}101)$$

This Green's function can be thought of as the field generated by the individual element at $\boldsymbol{r}'$.

Kirchhoff's integral equation can be used to calculate the scattered or radiated fields since the current, which generates a radiated field, can be determined from the fields using Maxwell's equations and normal electromagnetic boundary conditions, i.e.

$$\boldsymbol{J} = \hat{n}\times\boldsymbol{H} = -\frac{j\omega\epsilon_0}{k_0^2}\hat{n}\times[\hat{z}\times\nabla\boldsymbol{E}_z] \qquad (2\text{-}102)$$

One may insert the equivalent electric and magnetic sources into the integral equation to determine the scattered electric field for a general problem as

$$E_z^{scat}(\boldsymbol{r}) = \oint_C \{-jk_0Z_0J_z(\boldsymbol{r})G^{2D}(\boldsymbol{r},\boldsymbol{r}') + M_t[\hat{n}'\cdot\nabla'G^{2D}(\boldsymbol{r},\boldsymbol{r}')]\}dl' \qquad (2\text{-}103)$$

Overview

The goal of this chapter was to lay the mathematical foundations upon which Maxwell's equations may be decomposed into a system of equations that may be solved more easily using computational techniques. Using standard techniques, the equations were modified into their weak formulations, which allow a linear approximation of the unknown fields to be extracted from the integrals leaving an integral that is solvable using standard techniques.

As an integral based method, the finite element method will not give an exact solution on its own. To account for this, several boundary condition techniques were mentioned and discussed. Most common finite element codes rely on techniques, which approximate the boundary conditions through artificially constructed regions that must also be constructed and simulated through the same techniques as the primary finite element domain. On the other hand,

an exact solution can be had by integrating over the boundary of the domain instead. This is not without its problems as, like in the finite element construction, these boundary integrals are not always directly solvable. For some problems, this does not lead to many problems and the problem may still be solved in a reasonable time. However, if the problem is to be generalized into any type of boundary structure, particularly with a triangular decomposition, this leads to computational complexity that must be discussed further.

Chapter 3
Programming Considerations for the Finite Element Boundary Integral Method

Background

A fully functional, complete electromagnetics simulation suite must consider two primary focuses: design and analysis.

The previous chapter described the mathematical foundations for the latter. In addition, when it comes to implementation in software, analysis must also take into account many factors as well. Besides discretization of the structure into individual elements, one must also take into account limitations with the computer as well. These include things like precision and structuring the algorithms in a way that is well suited for accelerated calculation in parallel.

There are additional considerations that must be made prior to analysis even beginning as well. These can all be summarized under the general label of design. These include things like building a mesh, efficient representations of objects to be modelled and optimization, which ties the design and analysis together.

Additionally, as the previous chapter has mentioned, the boundary integral, while providing an exact solution to the fields at the boundary, may possibly lead to many computational issues. Particularly, without modification, an electromagnetic boundary that has been truncated for a solution in a boundary integral creates a boundary integral matrix with $O(N^2)$ computational complexity. In addition, the calculation of each element of that matrix has its own complexity, which can potentially grow the time required to calculate the system to an unreasonable level.

This chapter will focus on how to address these considerations.

Mesh Generation: Brick Versus Prismatic Elements

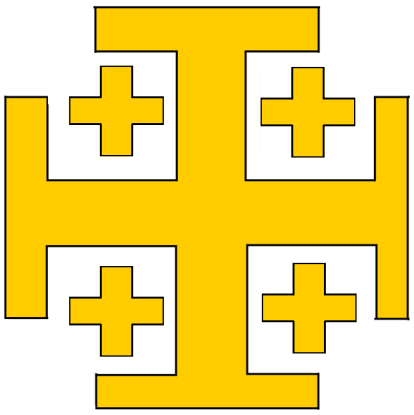


Figure 3-1: A single unit cell of an arbitrary frequency selective surface that has been built using brick elements.

The process of mesh generation involves representing an object by a set of polygons, which approximate the geometric domain of that object. While this process can be done using polygons of any number of edges, typically these are done using polygons with three or four edges, i.e. with triangle or quadrilateral polygons.

When designing a periodic frequency selective surface in two-dimensions, it is helpful to determine which constraints and characteristics of the problem might be suited to which type of simulation. The reason for this is an object which is modeled using just quadrilaterals may be simulated much more quickly due to regularity in the angles of the individual elements and, if desired, the size of the elements themselves. For example, if a regular grid of brick elements such as the one in Figure 3-1 is used, not only is there a large reduction in memory since the mesh may be simply calculated on the fly rather than stored but the finite element and boundary integral matrices can be calculated quickly using the simple equations seen below.

$$I_1 = \begin{cases} \frac{L_{TE}}{6}\left(c_1\frac{L_{NTE1}}{L_{NTE2}} + c_2\frac{L_{NTE2}}{L_{NTE1}}\right), if\ TE = SE. \\ -\frac{L_{NTSE}}{6}c_3, \qquad\qquad otherwise \end{cases} \quad (3\text{-}1)$$

$$I_2 = \begin{cases} \frac{c_4}{36}\prod_{D=1}^{3} L_D, if\ TE = SE. \\ 0, \qquad\quad otherwise. \end{cases} \quad (3\text{-}2)$$

$$Z_{FE} = \frac{I_1}{\mu} - k^2 I_2 \epsilon \quad (3\text{-}3)$$

where, c_1, c_2, c_3, and c_4 are scaling constants depending on the direction of the element. TE represents the current test element and SE represents the current source element used for calculation of the individual finite element impedance matrix entry. L_{TE} represents the length of the test edge, L_{NTE} represents the lengths of the edges that are not under test, and L_{NTSE} represents the length of the edge that is not the test or source. L_D is the length of the element in one of the cardinal directions, and Z_{FE} is an individual entry into the finite element impedance matrix.

From these equations, it becomes clear that in a brick element simulation, there are many opportunities to simplify calculation and speed up overall computation of the entire impedance matrix. For example, if the simulation is done using a mesh that is built entirely from cubic elements, the calculation of each I term becomes a constant since all elemental length are constant. Thus, variations in the impedance matrix comes from the material parameters only.

The boundary integral terms are slightly more complicated since they require numerical integration of the Green's functions and must consider contributions from all elements. Still, regularity in the geometry allows numerical and analytical integration to be calculated simply by forcing many potential contributions to be perpendicular to the elements. This allows the Ewald's summation to converge somewhat quickly compared to a model in which such a condition is not

enforced. Furthermore, calculation of the boundary integral using quadrilateral elements allows several terms in the boundary integral to be calculated analytically rather than numerically.

Figure 3-2: A pixel based unit cell that features metallic elements that touch on the corner.

If a brick element based finite element simulation is efficient and fast, one might wonder why one would use a triangular form instead. While a brick element system is useful for simulations that consist solely of dielectrics, where one can usually simplify geometries by inputting parameters that use equivalent parameters, or one in which a frequency selective surface built solely out of square metallic patches is desired. There are many cases where such a simulation does not guarantee accuracy. In the previous chapter, an example was given of a circular patch built from brick elements, which required an extremely fine mesh, in compared to one which used prismatic elements. Another example, perhaps more problematic would be the one seen in Figure 3-2. In this example, a frequency selective surface has been generated which has pixels that only touch at the corners. This is problematic because in a metallic patch, the finite element works by determining the fields that result from induced currents on the metals. If there

is an isolated metallic square, one may be concerned with the accuracy of the current approximations that are unable to flow to other nearby elements. In practice, it is difficult to determine if such small isolated elements have a strong effect on the overall scattering profile. Furthermore, such elements may be vestigial having been the result of an element with greater effect that occurred in a previous generation of the evolutionary algorithm. This sort of mesh is not ideal because it does little to reassure an engineer of the physical bases for the scattering profile.

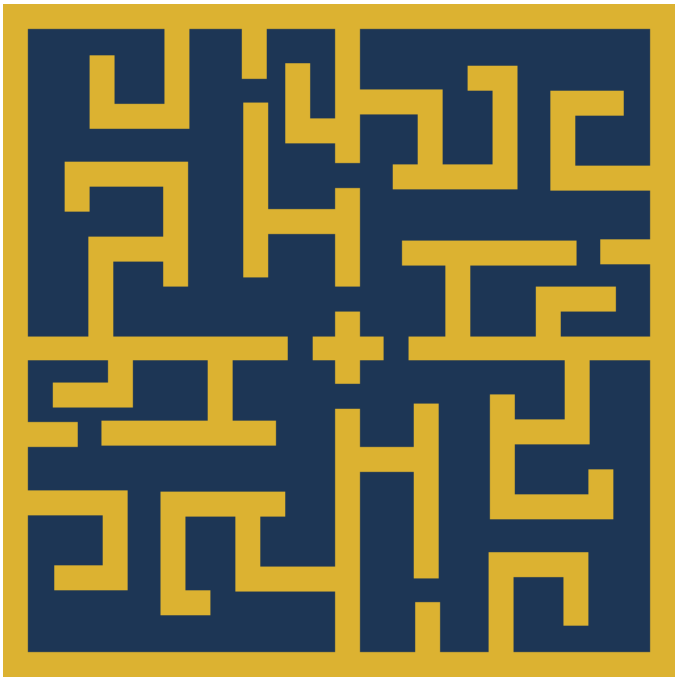


Figure 3-3: A maze-like brick element metallic screen.

One may continue to use a brick element approach by imposing manufacturability constraints on the optimization. This approach, however, requires the optimizer to do several checks each time a mesh is generated which adds programming complexity and possibly inconsistencies resulting in parameters that produce meaningful changes in the fitness value in some evaluations and no significant effect on the fitness values in others. This may have an effect on the overall convergence to a minimum. Another approach is to use a mesh generation function

that is guaranteed not to have such characteristics. The benefit of this approach is it simplifies the number of unknowns in an optimization. For example, one may use a maze generation algorithm in their optimizer which is guaranteed to generate a mesh that is optimized for a brick element approach with less parameters than optimizing each value individually [19, 20]. These algorithms generate grids, which will look similar to the one in Figure 3-3. This comes at a cost, however, since using this type of algorithm will reduce the overall variability in the type of models that one may generate.

Figure 3-4: Triangular elements eliminate situations where metallic corners are touching.

A prismatic mesh is an ideal compromise because it maintains regularity in the vertical direction, which makes it simple to maintain periodic boundary conditions and, in most cases, a metallic frequency selective will not need variations in the vertical direction. As has already been shown, it makes it simple to model curves. Most importantly, it makes it easy to eliminate floating metallic elements. For example, looking back at the mesh in Figure 3-2, one may regenerate it using a triangular mesh generation routine and arrive at a mesh similar to the one

seen in Figure 3-4 depending on how the paths are defined. This mesh not only better represented the geometry of the material but also no longer has a risk of having ambiguities in the current flow.

Accelerated Calculation of the Boundary Integral Using the Adaptive Integral Method

Using prismatic meshes over brick-elements does have a significant drawback, however. The construction of the numerical system is a time consuming process. In a standard prismatic element FEBI system, there are thousands of unknowns which require NxN equations to be calculated to fill the impedance system, where N is the number of unknowns. In addition, each element is electromagnetically dependent on every other element, which requires the computation of N equations to calculate each matrix component. Thus, in a worst-case scenario, calculating one frequency point using the standard prismatic periodic FEBI can take days. The first way to alleviate this process is to simplify the calculation of the impedance system. An effective way to do this is to use the Adaptive Integral Method (AIM) [21]. This not only reduces the system to an $O(N\ log\ N)$ problem but also simplifies the computation of each component of the impedance matrix. In addition, AIM also provides a way to speed up the solution of the system using FFT in conjunction with an iterative solver such as the bi-conjugate gradient (BiCG) method.

AIM can be utilized to provide a dramatic increase to the computation time of the FEBI method. First, a uniform AIM grid, such as the one seen in Figure 3-5, can be applied to the unit cell on the top and bottom surfaces. This grid allows the RWG basis functions to be approximated as the distribution on the grid. The interaction between two AIM cells sufficiently far away can then be accurately approximated. Thus, only the interactions between nearby elements need to be calculated using the traditional method as in (3-5). The periodic Green's functions in the far AIM cells form a Toeplitz matrix. Matrices of this type, when an FFT is performed form a one-

dimensional vector from which the computation time required in the BICG multiplications will be decreased.

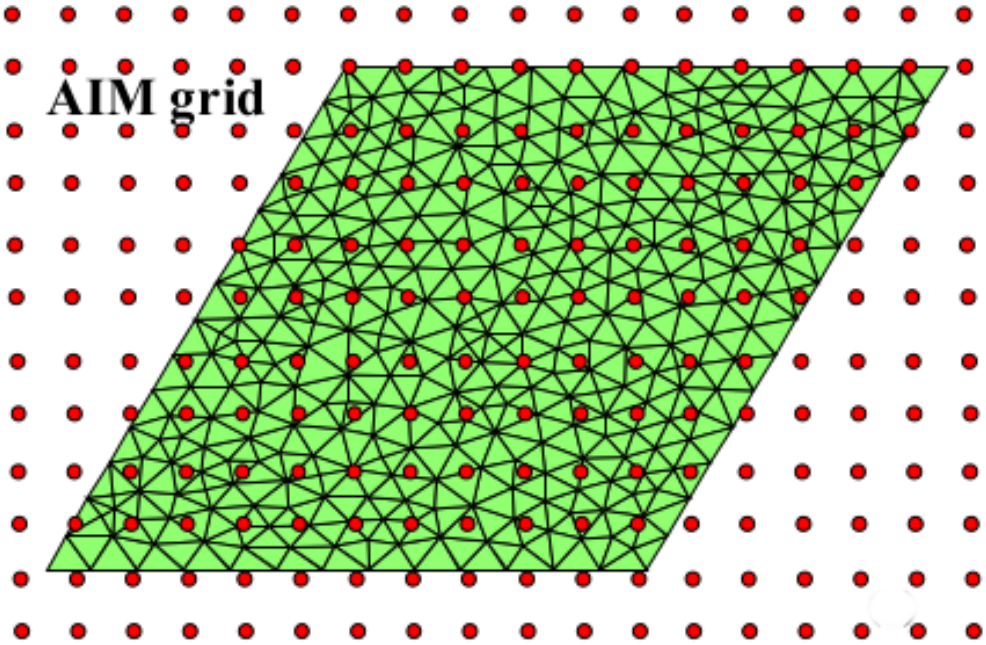


Figure 3-5: An example AIM grid. The near-field contributions are calculated as normal whereas the fields several grids away are analyzed in bulk.

$$\left\{[Z_{FE}] + \begin{bmatrix} Z_{BI} & 0 \\ 0 & 0 \end{bmatrix}\right\} \cdot \begin{bmatrix} E^{BI} \\ E^{FE} \end{bmatrix} = \begin{bmatrix} V \\ 0 \end{bmatrix} \quad (3\text{-}4)$$

$$Z_{BI} \cdot \boldsymbol{E}^{BI} \approx [\boldsymbol{Z}_{BI(near)} + \boldsymbol{Z}^{AIM}_{BI(far)}] \cdot \boldsymbol{E}^{BI} \quad (3\text{-}5)$$

Thus, by using the adaptive integral method, it becomes possible to gain the benefits of determining nearly exact boundary conditions using integral expressions while reducing the computational complexity to one that is closer to linear.

In addition to simplifying the convergence of the calculation of elements of the boundary integral impedance matrix, the adaptive integral method also ensures that the matrix is Toeplitz,

i.e. one in which the values on the diagonals are all constant. This can be described mathematically as

$$M_{Toeplitz} = \begin{bmatrix} x_0 & x_1 & x_2 & \cdots & x_{n-1} \\ x_{-1} & x_0 & x_1 & \cdots & x_{n-2} \\ x_{-2} & x_{-1} & x_0 & \cdots & x_{n-3} \\ \vdots & \cdots & \cdots & \ddots & \vdots \\ x_{-n+1} & x_{-n+2} & x_{-n+3} & \cdots & x_0 \end{bmatrix} \quad (3\text{-}6)$$

The key advantage of this banded nature is found in the ability to accelerate the calculation of a matrix-vector product, which is used in iterative techniques to solve linear systems.

Generation of Triangular Meshes Using Delaunay Triangulation

The process of subdividing a set of points, $\boldsymbol{P}$, into triangles such that the points do not lie within the triangles is known as a triangulation. In general, a triangulation is not limited to a planar two-dimensional object and may be extended to higher dimensions using simplices such as tetrahedral objects in three dimensions. In this work, a two-dimensional triangulation extruded in the vertical dimension to form prismatic elements will be used.

In a finite element problem, it is important that the elements are somewhat regular without having extremely small angles as well as limiting the overall number of triangles, in effect reducing the number of unknowns in the finite element problem. A commonly used method designed to maximize the minimum angle in a triangulation is known as Delaunay triangulation, named after the work done by Boris Delaunay in 1934 [22]. This triangulation has many properties that make it useful for finite element simulations such as

- The union of all of the triangles forms the complex hull of the points used to produce the triangulation.

- Any triangulation of point set, $\boldsymbol{P}$, consisting of n points and k points on the complex hull contains at most $2n - 2 - k$ triangles.
- The triangulation maximizes the minimum angle in a triangulation but does not necessarily minimize the maximum angle or edge length.
- Any circle passing through an edge in a Delaunay triangulation, AB, does not contain any other point in the point set, $\boldsymbol{P}$, used to seed the triangulation.

This final property, which is key to the Delaunay triangulation process, is maintained using a determinant relation. Delaunay found that any point, D, lies within a circumcircle enclosing a triangle, ABC, if and only if the following relationship is satisfied.

$$\begin{vmatrix} A_x & A_y & A_x^2 + A_y^2 & 1 \\ B_x & B_y & B_x^2 + B_y^2 & 1 \\ C_x & C_y & C_x^2 + C_y^2 & 1 \\ D_x & D_y & D_x^2 + D_y^2 & 1 \end{vmatrix} = \begin{vmatrix} A_x - D_x & A_y - D_y & A_x^2 - D_x^2 + A_y^2 - D_y^2 \\ B_x - D_x & B_y - D_y & B_x^2 - D_x^2 + B_y^2 - D_y^2 \\ C_x - D_x & C_y - D_y & C_x^2 - D_x^2 + C_y^2 - D_y^2 \end{vmatrix} \quad (3\text{-}7)$$

$$> 0$$

where A, B, and C are sorted in counterclockwise order.

There are many implementations of such triangulations. Rather than implement a new Delaunay triangulation library, the meshes in this work were made using code that takes advantage of the well-established Computational Geometry Algorithms Library (CGAL), which supports many important geometry algorithms has been optimized for efficiently generating meshes [23]. The C++ code has been included in Appendix A.

Using Bézier Surfaces for the Generation of Arbitrary Meshes

When designing unit cells for 2D periodic frequency selective surfaces, it is desirable to have a convenient way to represent the unit cell. In normal brick element codes, this can be done with a one-to-one pixel representation. That is, if a pixel is written as one number, it corresponds

to the material represented in the database by that same value. However, with triangular meshes it begins to become impractical to do an optimization of material values for each triangle. Even simple meshes may have hundreds of triangles with each triangle being represented by a set of vertex locations and their corresponding faces. It is in this situation that Bézier surfaces become useful as a mathematical tool.

Bézier surfaces are a two-dimensional extension of the Bézier curve. Mathematically, Bézier curves may be defined as a sum of series of Bernstein polynomials [24]. The conventional ith Bernstein polynomial of order n is defined on the interval $0 \leq u \leq 1$ as

$$B_{i,n}(u) = \frac{n!}{i!\,(n-i)!} u^i (1-u)^{n-i} \qquad (3\text{-}8)$$

Given n+1 geometrical control points, $\boldsymbol{p}_i$, the Bézier curve $c(u)$ is defined in terms of the nth order Bernstein polynomials $B_{i,n}(u)$ by

$$c(u) = \sum_{i=0}^{n} B_{i,n}(u)\boldsymbol{p}_i \qquad (3\text{-}9)$$

Because the weighting factors are control points in the x-y plane, the Bézier curve is determined by the weighted average of the control points where the individual weighting varies with the parameter, u. This allows for a geometrical interpretation of the Bézier curve as the weight of the control point is changed. As the position and weight of the control points is moved, the weights pull and tug at the overall geometrical representation, but the curve always remains smooth and inside the convex hull formed by the control point coordinates. Other polynomial formulations, such as the Taylor series, generally do not have such geometric interpretations or relationships with the resulting curve.

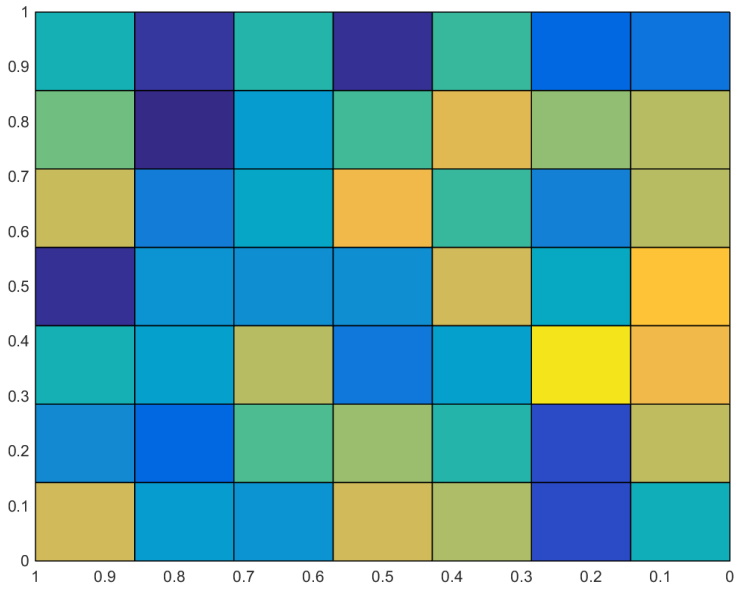
These Bézier curves can be easily extended to a two or higher dimensional surface. While a list of control points defines the Bézier curve, a grid of control points can do the same for a

Bézier surface. The basis function for this case is formed by multiplying two independent Bézier curves in their respective dimensions as follows:

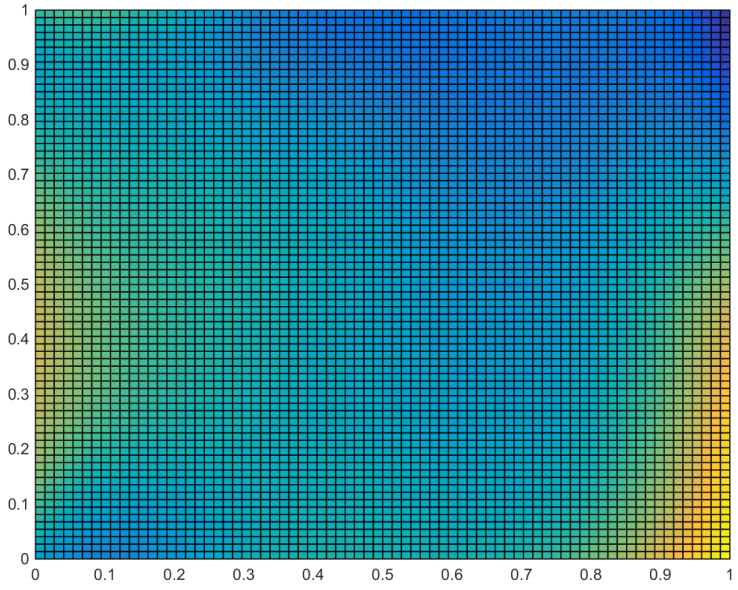
$$c(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} B_{i,m}(u)B_{j,n}(v)\boldsymbol{p}_{ij} \qquad (3\text{-}10)$$

where $\boldsymbol{p}_{ij}$ are the set of $(m+1)\times(n+1)$ control points. These two dimensional basis functions exert most of their influence in a region localized by the control points but are non-zero everywhere. Thus, to adapt the Bézier surface to the design of a metallic screen, a threshold on the resulting curve was applied so that every point above the threshold is determined to be metal and every point below it is determined to be simply a dielectric.

A key advantage of using a Bézier surface representation for frequency selective surface design is the ease of incorporating design rules into the optimization of the surface. The Bézier surface is effectively a two-dimensional sum of weighted polynomials. While it is not an interpolation technique and, therefore, the subsequent surface does not pass through the control points, the surface is guaranteed to remain within the convex hull of the control point matrix. In essence, the control points act as a pulling force on the resulting surface. Because of this, one may easily apply design rules, such as boundary conditions or symmetry, to the control points and these conditions are automatically transferred onto the Bézier surface. This feature, which can be seen in Figure 3-6, makes this representation incredibly useful for design and optimization of periodic frequency selective surfaces.



(A)



(B)

Figure 3-6: (A) A surface plot of the control points inputted into a Bézier surface algorithm. (B) The Bézier surface that resulted from the inputted control points in (A).

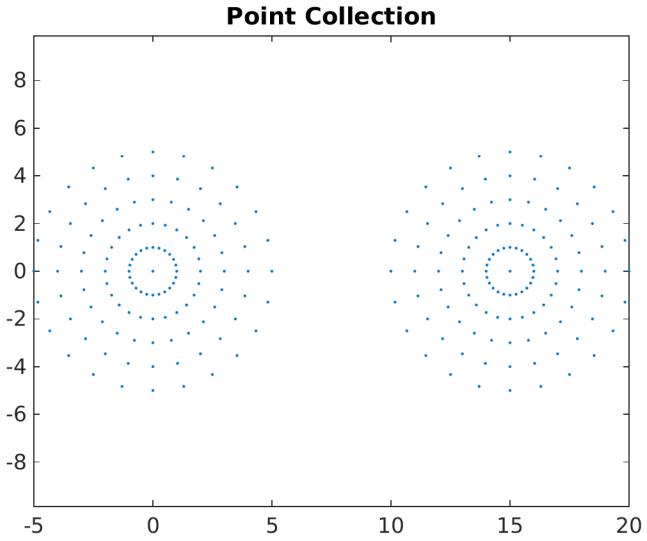
It should be noted that in the work described in this dissertation, the surfaces generated do not represent a true Bézier surface. In a true Bézier surface, the control points are allowed to vary and those variations are what changes the shape of the surface. In this work, in order to optimize frequency selective surface generation, the control points are held at fixed positions and the optimization parameters are instead gravitational force multipliers, which effect the overall pull of each control point. In effect, this transformation frees the optimizer from needing to optimize for multiple dimensionality reducing the number of parameters. Furthermore, this allows the traditional control point matrices to be larger than would be effective in an optimization routine allowing for a finer or courser mesh depending on the problem without requiring a large increase in the optimization parameters.

Using this method to adapt Bézier surfaces, which generates a height map across a two-dimensional plane, will yield a collection of points in that plane which represent the overall shape of the intended surface to be simulated. However, inputting those points directly into a triangulation algorithm to arrive at a mesh is ineffective and, at best, will generate far too many triangles, and, at worst, fail to generate a mesh altogether. Because of this, it important to derive a method which can be used to filter out excessive points from the surface.

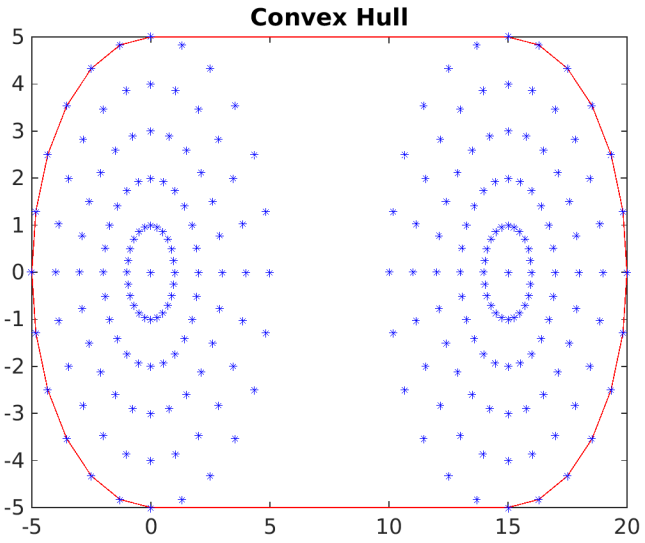
This is a complicated problem because not only must the internal points be removed but the remaining points must also be ordered in a way that the resulting polygon is simple, i.e. one in which the lines do not intersect and the sides form a closed path.

Given a collection of unconnected points, such as in Figure 3-7 (A), finding an efficient path through each point is a well-known complicated class of problems commonly called a travelling salesman problem. These problems, given no constraints, are called NP hard because they cannot be solved in polynomial time. These problems are usually computed using heuristic techniques such as a genetic algorithm.

In the case of this research, the additional constraint that no lines may be crossed can be applied to it, which allows the opportunity to use an extremely effective technique to find the correct path around the points. The technique that will be used is called the alpha shape. First proposed by Edelsbrunner et al. [25], the alpha shape is a generalization of the convex hull, i.e. the smallest set of points that encloses a shape. An alpha shape itself is related to the concept of the Delaunay triangulation in that the process of generating an alpha shape uses the same process of determining whether a point on the set is enclosed within a circumcircle in order to find the shape.



(A)



(B)

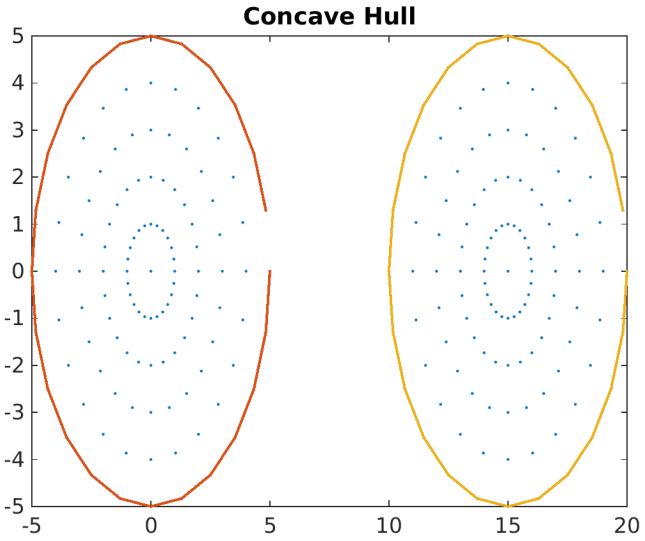


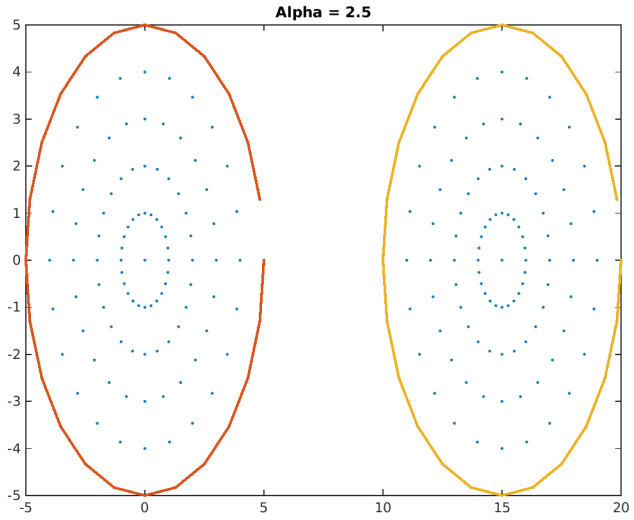
Figure 3-7: The difference between a convex and concave hull is shown. (A) The point collection used to generate the hulls. (B) An example of the convex hull. (C) An example of a concave hull.

Convex hulls are simple to find since all one must do is find the minimum number of edges which enclose the point set. However, it is intuitive that this technique will eliminate much of the information about the shape of those points. This is what makes the alpha shape useful. The alpha shape may find the convex hull or it may find the concave hull, i.e. the set of edges that not

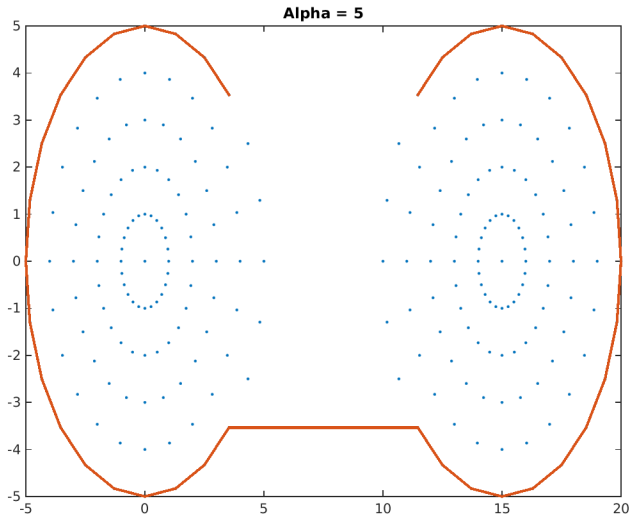
only enclose all the points but also follow the general shape of those points. Graphically, the difference between a convex and a concave hull may be seen in Figure 3-7.

This provides us a simple and powerful method to find any geometric shape from what would otherwise be an unconnected set of points. By controlling the alpha-parameter, i.e. the parameter used to control the radius of the circumcircles that carve out the underlying shape, one may easily determine a well-ordered path around the points that are above the threshold value.

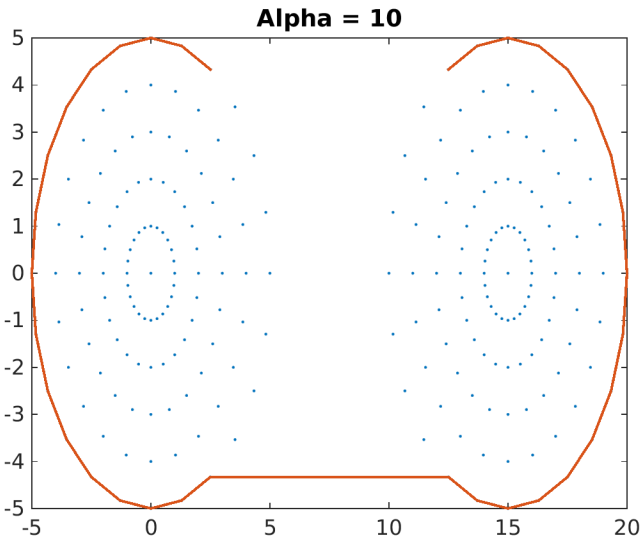
A good metaphor [26] for the process of generating an alpha shape for a set of points, ***P***, is to imagine that the points are chocolate chips which are embedded into a container of ice cream. The alpha shape is the result of scooping out the ice cream such that each scoop, or circumcircle, touches the chips but does not remove them. The shape is then defined using the set of locations of each chip in the order that the scoops were removed.



(A)



(B)



(C)

Figure 3-8: Three alpha shapes for a point collection. As alpha grows, the alpha shape approaches a convex hull.

Mathematically, one may define the circles in an alpha shape by their radius, α. These circles are translated iteratively through the domain and if a circle of radius α passes through two points in the point-set, a line is drawn connecting those points. By controlling the size of the

radius of the circles, one may find a shape that represents the convex hull or one that represents a concave hull depending on the chosen radius. Several alpha shapes for a single point set may be seen in Figure 3-8.

Alpha shapes are fast and effective because they determine the order of the points from the outside rather than from the points themselves, which is how one would normally approach a travelling salesman problem. Upon ordering the points, the Bézier surface can simply be inserted into the meshing algorithm resulting in a mesh such as the one in Figure 3-9 allowing it to be used in a FEBI simulation.

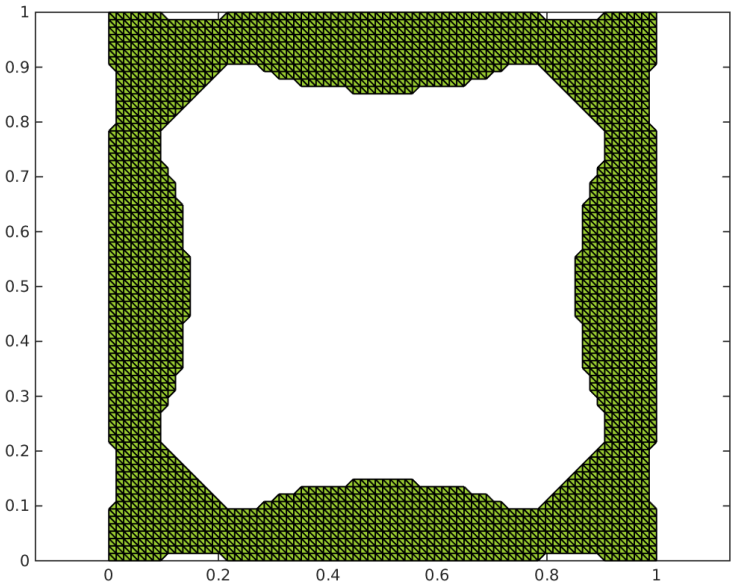


Figure 3-9: A Bézier surface following thresholding has been inputted into an alpha shape algorithm. The points used in the final meshing algorithm are extracted from this alpha shape for efficient meshing.

The MATLAB code that has been written to generate arbitrary Bézier surfaces for a PFEBI simulation has been included in Appendix B.

Mesh Optimization Using Covariance Matrix Adaptation Evolution Strategy

In addition to a simple mathematical representation, a robust optimization technique is useful for designing frequency selective surfaces. While there has been much work demonstrating the use of Monte-Carlo or more advanced techniques like Genetic Algorithm [27, 28], recently there has been a move towards other strategies like the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [29, 30].

The basic algorithmic structure of a CMA-ES optimization is similar to the genetic algorithm, which is described in Figure 3-10. CMA-ES expands on the evolutionary process of the genetic algorithm by evolving the distribution of fitness evaluations rather than the considering each function independently. The general process is described in Figure 3-11. As can be clearly seen, an immediate advantage over the GA is that the CMA-ES does not rely on the parameters used in the parent generation. Instead, each successive generation a population is generated from an approximation normal distribution of the previous generation's fitness evaluations in the direction of a minimum value. This makes the CMA-ES algorithm more robust since it is less likely to become stuck in the event that the algorithm arrives at a poor population.

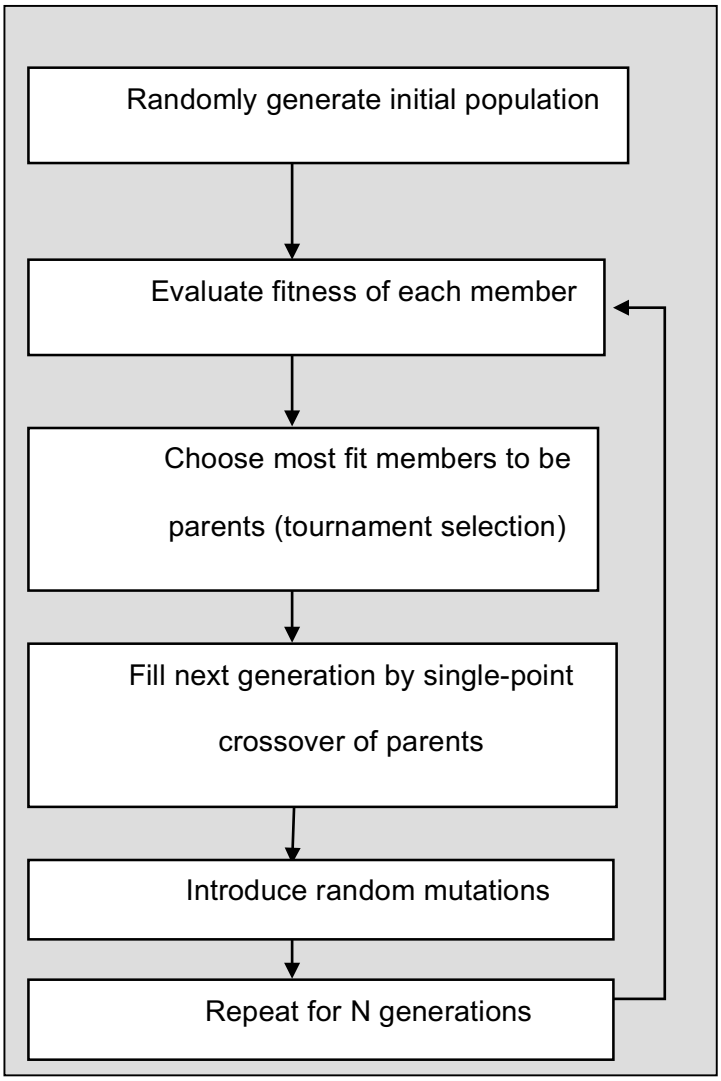


Figure 3-10: A flowchart that describes the general evolutionary process used in a genetic algorithm.

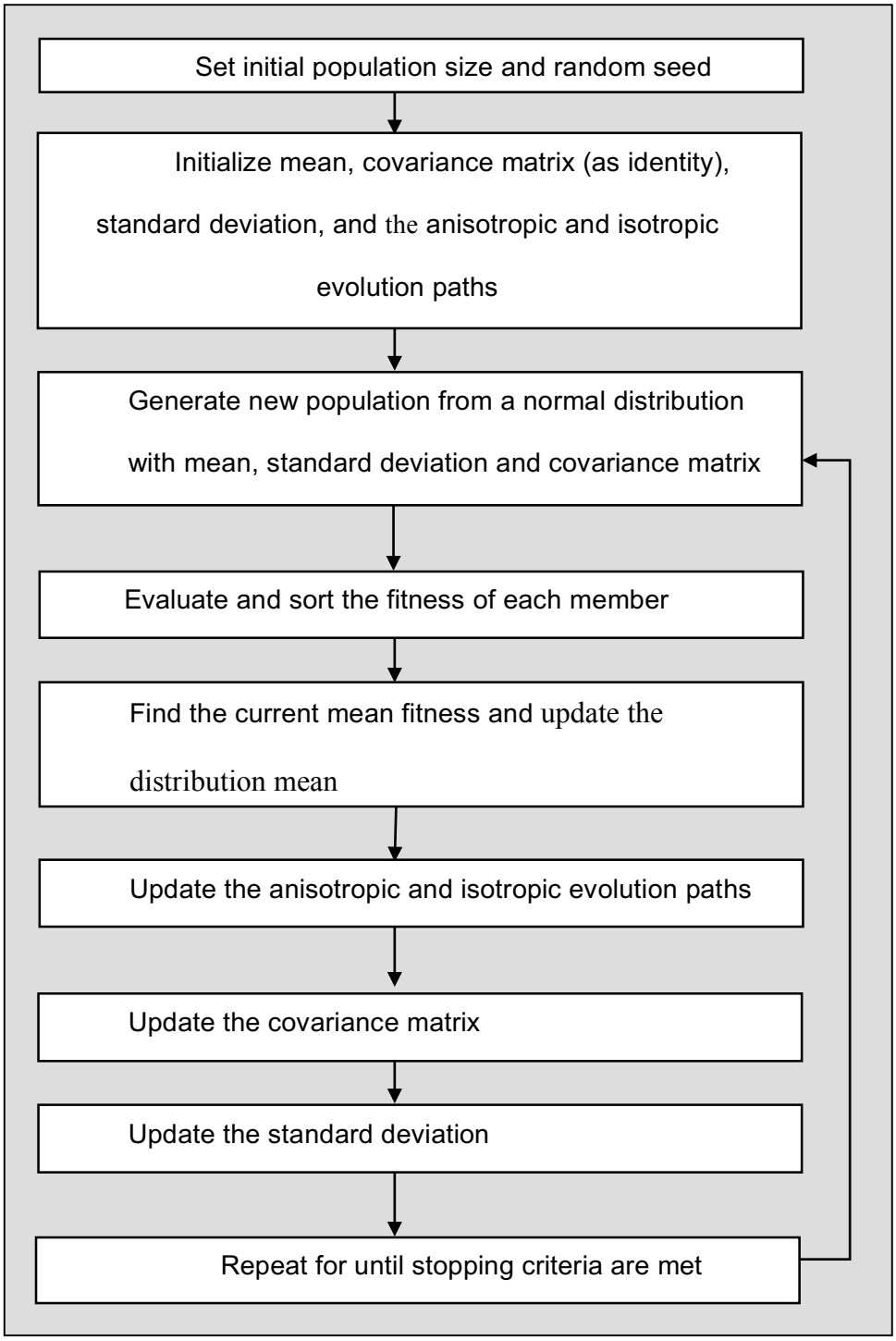


Figure 3-11: A flowchart that describes the evolutionary process of the CMA-ES algorithm.

This algorithm holds several key advantages over the well-known genetic algorithm. Perhaps key among them for this work is that it is a continuous algorithm. The traditional gene representation uses discrete binary values. While this method will work well for a brick element system where the material parameters are regular and discrete, it is ineffective for a triangulation, which has a variable number of elements that have been generated from a continuous representation such as a Bézier surface. For example, in a genetic algorithm to represent a number from zero to one, one must calculate the series

$$x = \frac{1}{2^N - 1} \sum_{i=0}^{N-1} x_i 2^i \qquad (3\text{-}11)$$

For an eight-bit representation not only are the number of variables increased but the real-value representation is limited to 256 discrete values. In a continuous system like CMA-ES, only one variable is necessary for each value and any value that may be represented within the limits of the floating-point precision of the computer may be used.

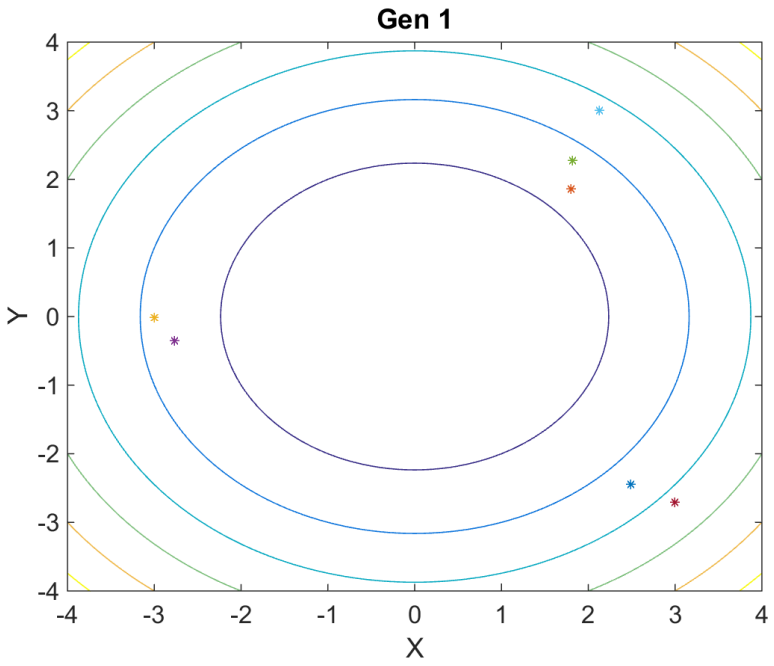
The basic formulation for a CMA-ES problem is as follows

$$x_k^{(g+1)} \sim m^{(g)} + \sigma^{(g)} \mathcal{N}\left(0, C^{(g)}\right) \; for \; k = 1, \ldots, \lambda \qquad (3\text{-}12)$$

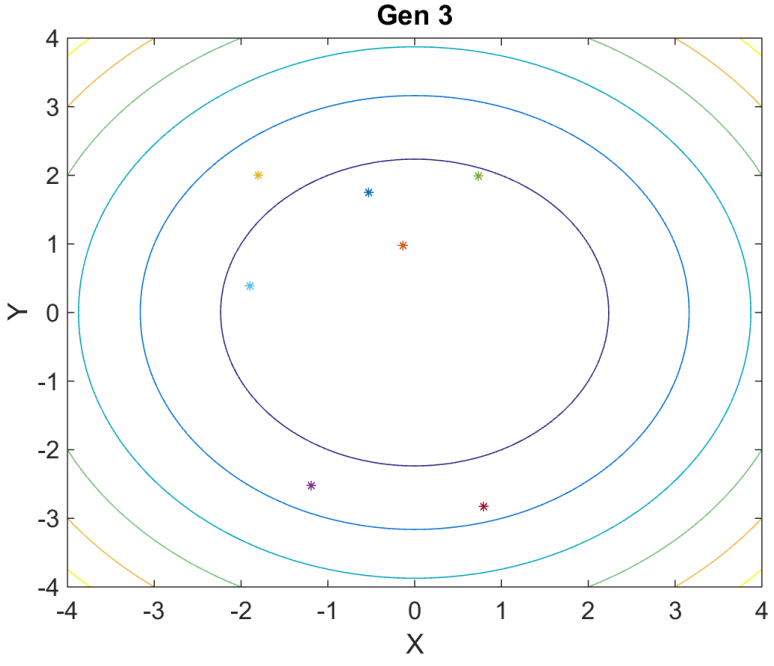
where g is the current generation, k is the current member of the population, m is the mean value at the current search generation, σ is the standard deviation to be used as a step size, and $\mathcal{N}\left(0, C^{(g)}\right)$ is a multivariate normal distribution with zero mean and covariance matrix $C^{(g)}$. This provides the key advantage over traditionally used stochastic evolutionary algorithms like GA. While GA works by intelligently modifying the population such that the fitness moves towards a minimum, CMA-ES operates by approximating a normal distribution of the covariance matrix at each generation. By constructing a statistical sample of all possible fitness values at each generation, CMA-ES also provides a distinct advantage in the population size of each generation of the optimization.

In GA, the probability of obtaining a minimum value increases as the size of the population increases. With a small population, unless many more generations and a large mutation rate are used, there is a risk that the optimization will either become stuck in a local minimum far from the true minimum or not find one at all. By increasing the mutation rate too high, GA loses its intelligence and becomes no different from a Monte Carlo, or completely random, optimization. On the other hand, if the population size is set too high, the optimization could possibly sample every possible variation of variables eliminating the point of using an optimization technique. While there is no ideal population size and mutation rate, typically one would use a population that is on the order of the number of variables and a mutation rate that encourages small changes in each child in order to provide a good chance to jump towards other minima without making changes too dramatic.

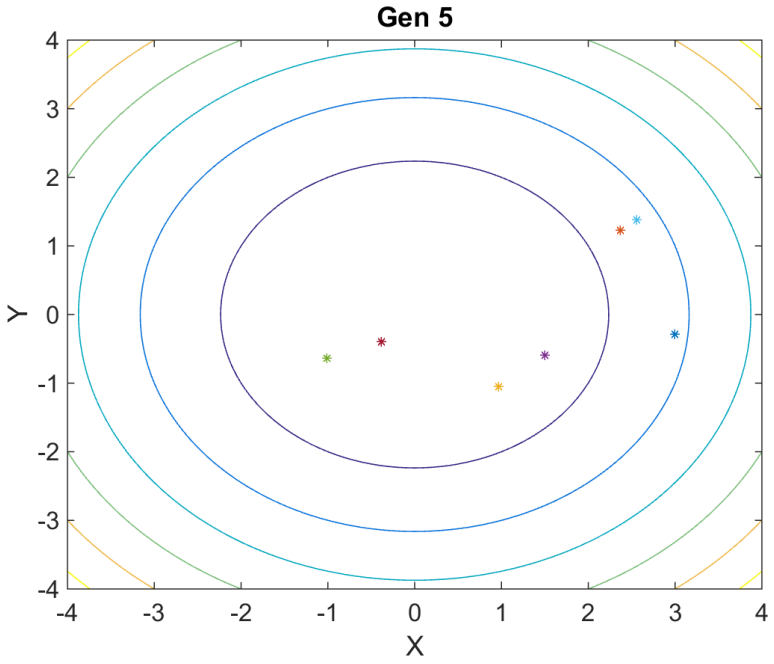
CMA-ES, by operating on the statistical distribution of the domain can be ran using a smaller population without such risks by taking advantage of statistical sampling theory. Just as in a presidential election where one may randomly sample a small subset of the population and extrapolate to the preferences of the population as a whole, at each generation, CMA-ES creates a random sampling of the fitness distribution of the global population and is able to move to a minimum. For example, a classic example used to demonstrate Newton's method for optimization is to determine the square root of a number. This is done by finding the minimum of the equation $x^2 = r$. To provide an overly simplified example for CMA-ES, Figure 3-12 demonstrates the use of CMA-ES to find center of a two-dimensional version of this problem where $x^2 + y^2 = r^2$. It can be seen that in each generation, an estimate of the covariance matrix is determined until it moves the distribution to the minimum.



(A)



(B)



(C)

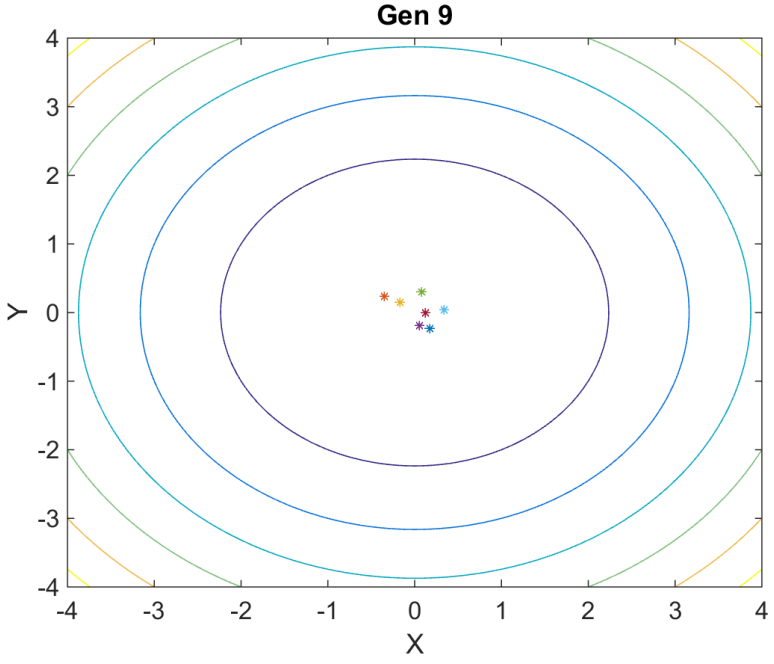


Figure 3-12: A CMA-ES optimization example in which the minimum of $x^2 + y^2 = r$ must be found within the bounds $[-3, 3]$.

Of course, the example shown in Figure 3-12 is a poor candidate for stochastic optimization and a solution would be more easily found using classical optimization techniques like Newton's method. This example, however, demonstrates quite well how the CMA-ES creates a random sample and narrows the distribution to something surrounding an optimal point.

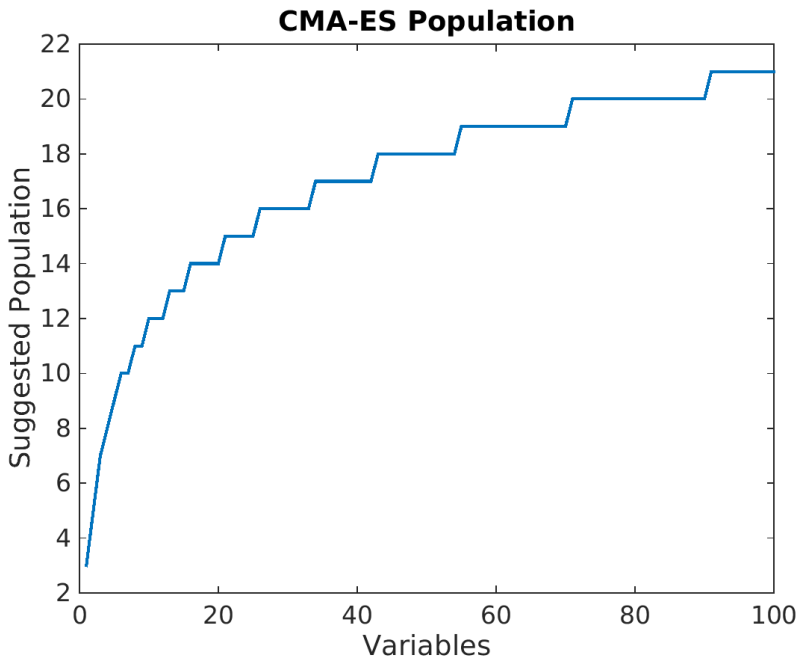


Figure 3-13: The recommended population growth rate with respect to number of variables using CMA-ES.

In fact, CMA-ES has a recommended population that grows logarithmically as in Figure 3-13 [31]. That is, for a system in which the number of variables is N, the suggested population size, λ, is

$$\lambda = 4 + 3floor(\ln N) \qquad (3\text{-}13)$$

where floor is the function that maps a real number to the nearest, smaller integer value.

This small population size combined with quick convergence makes CMA-ES the ideal optimization technique for a generic triangular meshing technique like the Bézier surface formulation. Accounting for symmetry and other manufacturing considerations, a typical Bézier surface is able to have a large amount of variability with thirty to forty control point variables. By

taking into account other parameters such as the unit cell size, it is rare for a CMA-ES optimization with strong variability to require more than seventeen population members.

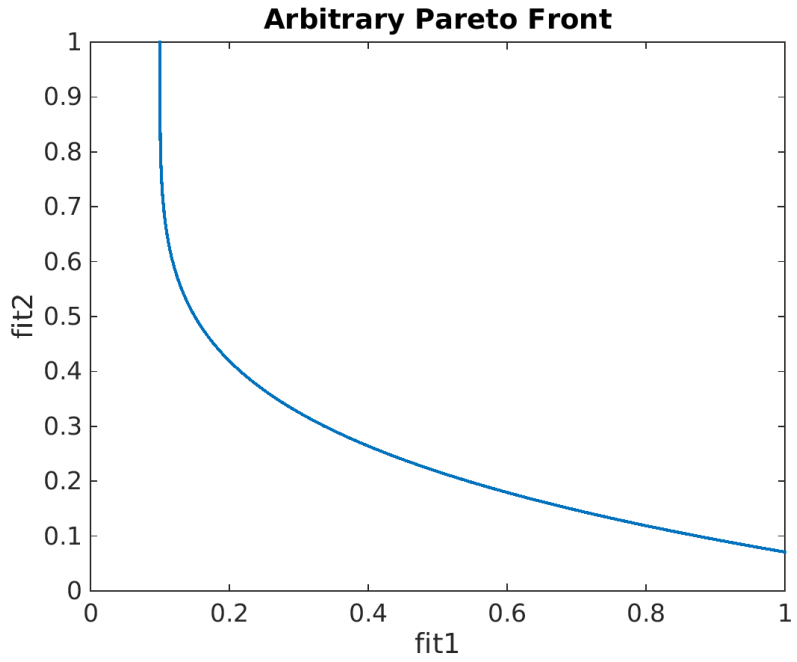
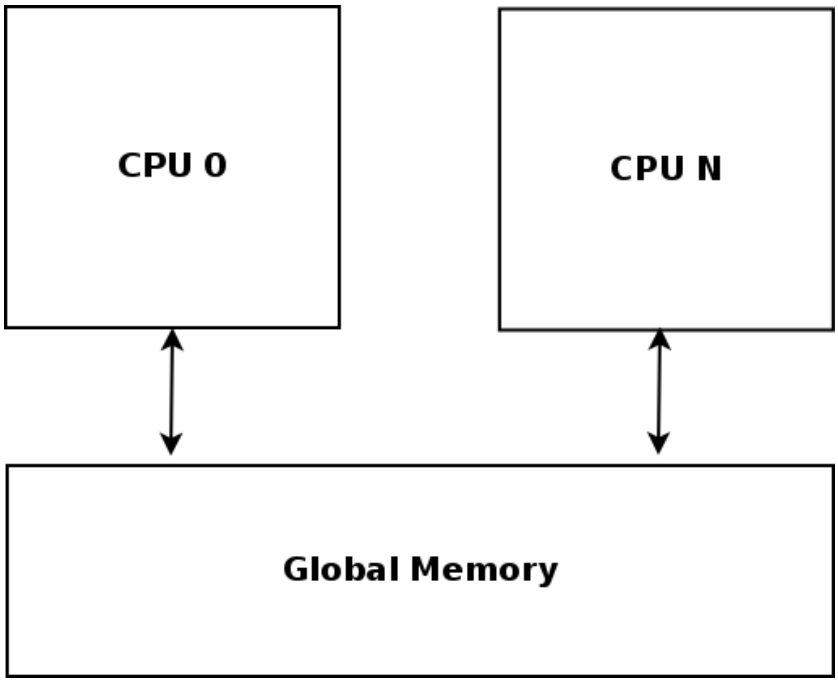
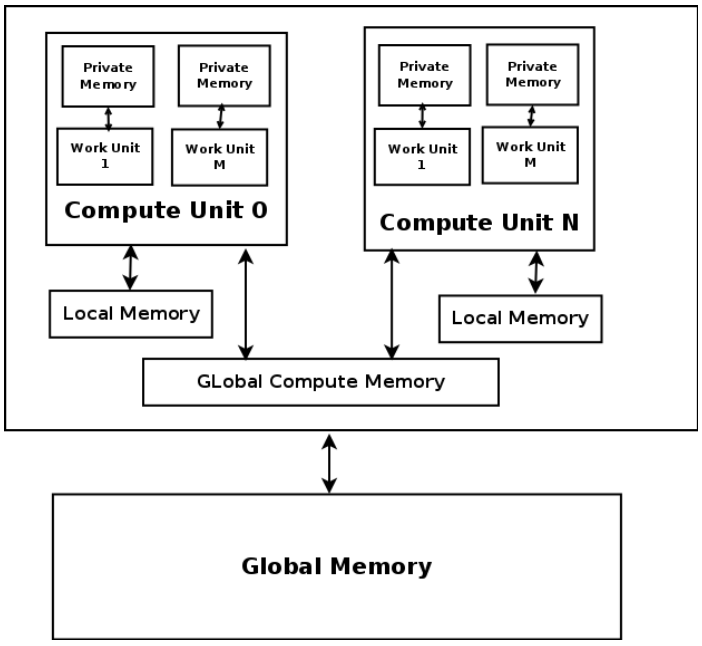


Figure 3-14: An arbitrary Pareto front to demonstrate the tradeoff between design requirements.

While the vanilla CMA-ES algorithm works well for many cases, there are times when a designer does not feel that it is able to find a target minimum at a fast enough rate. In these cases, many strategies may improve the convergence rate. One method, which has been discussed frequently, involves increasingly the population size as the generations increase [32]. A technique that I found worked well for Pareto front, i.e. problems in which there are multiple fitness values that may or may not be independent, such as the one seen in Figure 3-14, was to periodically analyze trends in the optimization and reseed the optimization using new starting points and parameters, which favored a desirable balance between fitness values. Ultimately, when using these sorts of optimizations, the strategy used is up to the designer.



(A)



(B)

Figure 3-15: Schematic diagrams of the work flow for software ran on the CPU (A) or a compute device such as a GPU (B).

Recently, there has been an exciting development in computer science that has caused a divergence in processor models. While the traditional idea of a CPU has continued to exist relatively unchanged since 8-bit CPUs became common in the 1970s, developing in a logical fashion as technology advanced by increasing instruction sets, bus sizes, transistor count, cores and so on, the graphics processing unit (GPU), which was originally seen as a co-processor whose purpose was to blit pixels together and transfer the resultant values to a memory range which would then be rastered onto a screen, has seen an interesting trajectory into something else entirely. Consumer demand, both through the increase in the market for three-dimensional video games and high performance computing arrays used to create processor which instead one designed to quickly do sequential operations has transformed into a powerful array of simple processors designed specifically to do thousands of vector calculations in parallel.

Figure 3-15 demonstrates a simple diagram of the differences between a basic CPU and a GPU. A typical modern CPU, such as the Intel i7 family of processors, consists of several, typically two or four, identical core processors, which have complex instruction sets. When software is compiled to run on one of these processors, assuming that the software is not multithreaded, it is converted to a set of instructions that the processor reads and converts to microcode, which is processed by one of the cores in sequence. These processors are optimized for generality so typically mathematical operations, unless there is a specific instruction designed to support that operation such as those provided by the Streaming SIMD Extensions instruction set, will not perform at an optimized rate unless the programmer or the compiler takes special care to account for it.

Specifically, the traditional model of a processor in a computer is not well optimized for large multi-dimensional arrays and, until recently with the development of the Advanced Vector Extensions instruction set, vector operations.

On the other hand, since graphics technology has shifted its focus from simple blitting operations to three dimensional vector processors, has become the ideal place to calculate these sorts of problems. The typical architecture of a GPU has a specific structure, which makes it attractive to exploit for certain problems. For example, every GPU has been designed under the assumption that it will be calculating values in a three or four element vector representing the color values: red, green, blue, and alpha or transparency. Furthermore, since the typical GPU has been designed under the assumption that it will calculate shading and location for triangular elements that are largely independent of one another, a typical consists of many parallel processors that each run at a rate that is slower than a typical CPU.

For example, a currently available high end GPU is the AMD Radeon R9 Fury X. This processor consists of 4096 stream processors that each have a core clock-rate of 1050 MHz and 4 GB of onboard memory. While on its face, these specifications are impressive, there are two important considerations one must make before writing code to be ran on a GPU. First, unless a specialized compute-oriented GPU like the NVidia Tesla is used, the GPU is also performing display operations and a large portion of the memory is blocked so that it can contain the current framebuffer, which is a bitmap of the currently displayed frame to be displayed on a monitor. Second, because the GPU is optimized for visual operations, a typical human eye is not able to process differences in color or position at a precision that one might be concerned with for scientific operations. Because of this, in general, GPUs process double precision, or 64-bit, operations at a rate that is much lower than the rate at which they process single precision, 32-bit, operations. Going back to the Radeon R9 Fury X, in the ideal case, it can perform 8192 billion single precision floating-point operations per second but only 512 billion double precision floating-point operations per second.

Furthermore, since the CPU and GPU do not share memory, each time a process is set up to be ran on a GPU, the data must be collected and placed in a GPU buffer that is transferred into

the GPU memory. This makes the use of a GPU for extremely large problems less attractive because unless the software is well programmed to account for a large difference in memory, software that is not well optimized might face bottlenecks transferring data to and from the GPU. For example, a large finite element problem, which has not been truncated using specific boundary conditions like periodicity and the boundary integral, can easily grow its memory footprint to over one hundred gigabytes. Such a problem would be difficult to justify for use on a GPU.

Finally, as a parallel-oriented processor, it is important to consider just what parallelization means in terms of computational code. The ideal problem for calculation in parallel is one in which there are many components, such as in a matrix, which are all completely independent of one another. However, interdependence does not necessarily rule out an application for parallel processing. If calculations are independent, the programmer can consider this by creating locks or mutually exclusive portions of the code where the two parallel processes must wait their turn and continue operating in parallel when they are not at the mutually exclusive point.

An extremely simple example of a problem in which most of the operations are independent but is not guaranteed to work properly in parallel is the dot product, that is $c = \sum_{i=1}^{n} a[i]b[i]$. This is a clear example of a case in which the majority of the computation is independent but the final calculation is not. Each multiplication between the elements in a and b are completely independent of each other but the answer, c, is not guaranteed to be accurate if this is programmed for parallelization naively. The reason is that if two operations are performed summing onto c at the same time, e.g. $c_{step} = c_{step-1} + a[j]b[j]$ and $c_{step} = c_{step-1} + a[k]b[k]$, both operations began with the same value for c unaware of each other resulting in a value that has overwritten the result from whichever process finishes first. This is called a race condition and can be addressed by isolating portions of the code in which this is possible and

making each process wait until other process finish performing that operation. By doing this, a dot product can still be calculated much more quickly in parallel but the improvement is somewhat less than the ideal situation of linearly increasing with the number of parallel processes.

This situation leads to the other common problem with a parallel calculation, a dead or live lock. This situation occurs where two processes that are running in parallel reach a point in the operation in which they are both waiting for the other to finish. Metaphorically, a deadlock is similar to an intersection in which all cars arrive at the same time and are each turning left. In this case, none of the other cars has the right-of-way and none can go until the other cars have gone. In reality, eventually one of the cars would make a move easing the traffic but in a rigid system like a computer, unless exceptions have been programmed for this case no car would ever be able to move. A live lock is similar but is best described by two people walking in opposite directions, to get around each other; they both move to try to get out of the way of the other. A live lock occurs when each person keeps moving in the same direction perpetually blocking the other person.

The key differences between CPU and GPU architectures are summarized in Table 3-1 below. As can be easily seen, when designing code to take advantage of the GPU, it must be designed with its strengths and weaknesses in mind. Particularly, the programmer must isolate portions of the code that are well-suited to parallel operation, ideally structure the data in a way that can be spread into a four-element vector such that they may be further calculated in parallel and, where possible, pre-calculate important double precision operations to reduce a significant bottleneck.

CPU	GPU
Sequential operation	Parallel operation
Simple to program	Must carefully consider possibility of bugs and locks
Calculations performed on a scalars	Calculations performed on 4-element vectors
Cheap memory	Expensive memory
Can quickly calculate double precision	Significant cost for double precision operations

Table 3-1: A summary of the architectural differences between a CPU and GPU architecture that might make one more attractive than the other for certain problems.

The final consideration for writing code to run on a GPU is the application-programming interface (API) to be used. Early attempts at GPU calculation operated in a way similar to writing software in an assembly language. Using this method, a clever programmer could manipulate the registers directly and do quick calculations. This method is not ideal because it limits the code to specific architectures and requires more advanced knowledge of the hardware than most programmers would normally have.

As these methods became more desired, two competing APIs emerged. The first widely used method is known as CUDA, or the Compute Unified Device Architecture. This method inherits many language features from C to allow a programmer to easily transfer data buffers to and from a GPU and perform calculations on them. However, as an NVidia technology, this API is restricted to NVidia GPUs.

As a response to the proprietary nature of CUDA, Apple created the Open Computing Language (OpenCL) API that is now managed by the graphics standard body, Kronos. Like CUDA, it is C based making it simple for an experienced programmer to adopt. Additionally, it has the advantage of not having a strict definition of a “compute device”. While CUDA code is

limited to only running on a GPU produced by NVidia or its partners, OpenCL can run on a GPU produced by any vendor or even a CPU or field programmable gate array. This has a huge advantage in that not only does it not restrict users to a particular vendor but also software written in OpenCL is much easier to debug since it can be ran over thousands of threads on a GPU or just one on a CPU. However, since it is a newer technology and many projects had already begun to be designed with CUDA in mind, it does not have as mature of a mathematical library requiring many basic functions to be written from scratch.

Modification of the Triangular Boundary Integral for GPU Applications

To begin, it is important to justify whether a periodic finite element boundary integral code is well suited for calculation on a GPU. As has been discussed in the previous section, offloading calculation onto a GPU has the potential to improve the calculation of a problem. However, if it is not well suited, such as a problem in which too many calculations are interdependent or a problem in which there is a significant amount of double-precision calculations in each parallel calculation such that it slows down overall operation enough to prevent an improvement in calculation time.

To begin, one may consider an arbitrary simulation as seen in Figure 3-16. This simulation is of a periodic gold FSS consisting of nine prismatic layers totaling $483.4\ nm$ in thickness and a square unit cell with sides that are $519.2\ nm$. This simulation is being evaluated in the important telecommunications band of $1.55\ \mu m$. It consists of two materials, a glass dielectric, which is seen in blue, and a gold screen, which is colored gold in the diagram. This mesh is built from a Bézier surface that allowed for high variability and consisted of 204 source points. Triangulation of this surface yielded 709 vertices and 1360 triangles per layer.

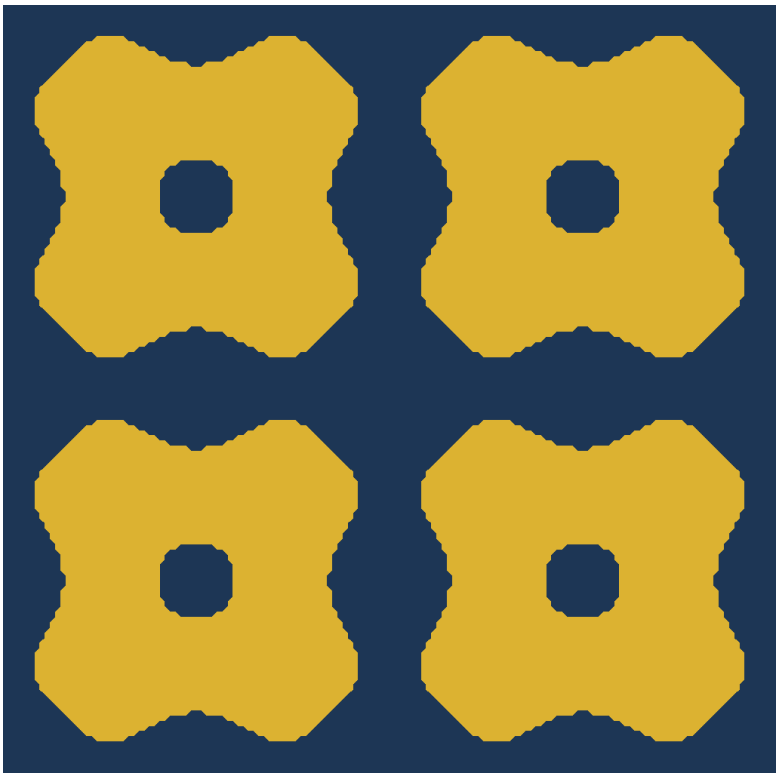


Figure 3-16: A two-by-two grid of an arbitrary FSS used to demonstrate the growth in the number of discretization unknowns. This surface extends infinitely in the horizontal and vertical dimensions.

Pre-processing this mesh determines that the finite element portion of the code has yielded 26520 unknowns and that the boundary integral portion consists of a much smaller amount of unknowns, specifically 2040.

Next, the structure of each constituent matrix must be considered. The finite element matrix is sparse, which means that the number of matrix elements to be calculated is linearly proportional to the number of unknowns. Furthermore, by embracing the weak formulation, the integration is done over the weighting functions, which are simple to calculate. Furthermore, the calculation of the finite element matrix has a large memory footprint since it requires the knowledge of each element and the material parameters thereof. The finite element matrix does not look to be a good candidate for GPU acceleration.

On the other hand, the calculation of the boundary integral matrix seems to be an ideal place to look for GPU acceleration with some modification. Before this can be considered, the

first thing that needs to be done is to apply the adaptive integral method. On its own, the boundary integral requires $O(N^2)$ storage. With the number of unknowns this problem requires, that would require a matrix with an astonishing 4194304 elements. Because these elements are double precision complex numbers which each require 16 bytes, this matrix alone would require $64\ GB$ of storage. The adaptive integral method reduces the number of elements to be calculated to $O(NlogN)$ which is much easier to work with. In reality, the number exact number of elements varies depending on the adaptive grids used and, in this simulation, using a large grid there are only 575066 total edges to compute. With each element requiring complicated integration and each element completely independent of the other, it becomes clear that the boundary integral with the adaptive integral method is the type of problem that is best suited for calculation on the GPU.

What makes this an even more attractive target for GPU calculation is the way in which each element is calculated. For each edge corresponding to an element on the impedance matrix that needs to be calculated, its impedance consists of its self-impedance and the contributions from nearby elements until it has reached convergence. This can be seen graphically in Figure 3-17. The adaptive integral method provides the advantages of allowing us to reduce the number of these interactions to consider by considering far-away contributions as a bulk contribution and it orders it in a way that is simple to iterate. In a GPU, this is useful because since each stream processor is itself multi-threaded as well as optimized for vectors, it becomes extremely simple to distribute these contributions across the processors multi-dimensionally.

Further acceleration can be done using the vector nature of the GPU. Each edge has an impedance that must be calculated in relation to the source edge and a distant edge. By distributing each of these calculations between the x and y elements of the calculation vector, the speed can effectively be doubled.

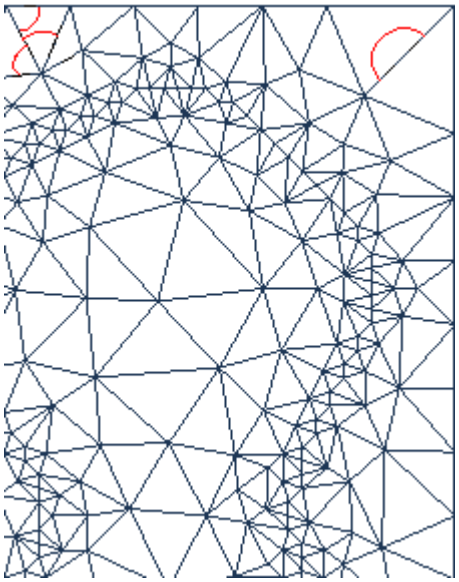


Figure 3-17: A crude demonstration of the edge interactions in the boundary integral. Interactions between edges are shown in red. In the top right, a self-interaction is shown.

In addition to algorithmic improvements, several considerations must be taken into account for these calculations with respect to GPU computation. One area that a programmer might immediately consider relates to the use of double precision.

A naïve attempt would be to use single precision except where necessary. However, single precision is only accurate to between six and nine significant digits. However, many of these edge contributions consist of differences that only occur at the sixth or higher significant digit. By using single precision numbers, there is a risk of propagating errors throughout the code.

Since the use of single precision representation will result in a result that is incorrect, with very few exceptions, this appears to be a poor choice to look for speed improvements.

Instead, the best path to GPU based optimization consists of two factors: taking advantage of the architecture and using built in functions.

It was mentioned earlier that the GPU is optimized around four-dimensional calculations whether they be red, green, blue, alpha or x, y, z, and w, where w is a scaling dimension. For example, one common calculation is the magnitude, $r = \sqrt{x^2 + y^2}$. By mapping the values into a

vector, for example $r = \sqrt{a.x^2 + a.y^2}$ or even $r = \sqrt{a.x * a.y + a.z * a.w}$, where $a.x = a.y$ and $a.z = a.w$, rather than multiple scalars, this can be calculated in much fewer cycles.

Another area that can dramatically improve the computation of elements is through judicious use of built-in functions [33]. Many GPUs have built in instructions to perform common complicated operations such as sin or cos. While in early GPUs most of these instructions were single precision only, with the increase in demand for compute devices, many higher end GPUs now include these operations at higher precision as well with the proper compilation flags selected. Because of this, one must be careful about which hardware the code is running on because without properly checking the hardware and case-based code modification it is possible that the software will not run or will not maintain proper accuracy.

In addition, it was earlier mentioned that the boundary integral calculation was improved using an Ewald transformation. This operation relies on the use of what is known as either the complex error or Faddeeva function. While the error function exists as a built-in function on GPU devices, it only exists for real valued operations. In order to calculate the Faddeeva function, a fast implementation of the operation was done the work of Poppe and Wijers as a guide [34].

Finally, another performance optimization technique that has been performed for the GPU-based boundary integral calculation. The various operations used in the overall calculation of the boundary integral relies on several values that are not independent of one another, or may be known a priori. For example, in the calculation of the Green's functions, there are operations in which the wave vector is divided by a variable that is a multiple of the wave vector. All cases in which this occurred were simplified to a constant value.

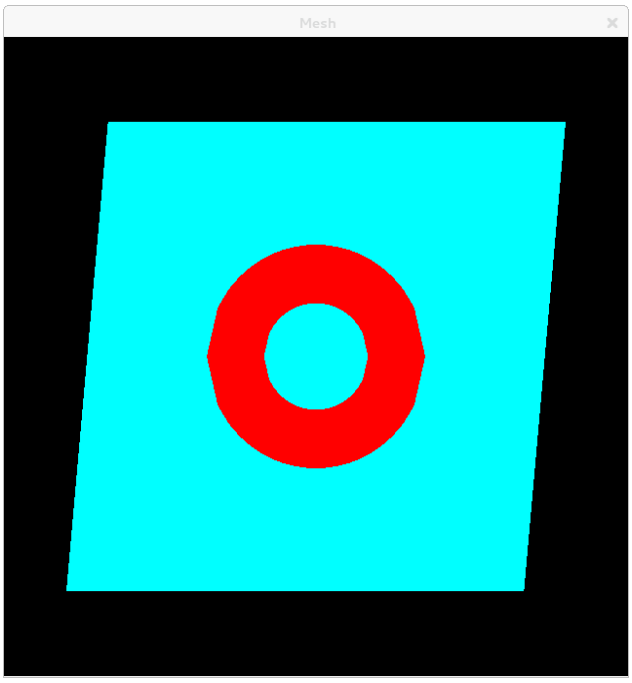
In general, however, the use of pre-computed values for performance optimization purposes do not tend to have a huge impact on performance and should only be done when they are obvious or when the operation being simplified occurs repeatedly and is complex. The best

method of optimizing the use of GPU for compute purposes is to rely on the advantages of the hardware.

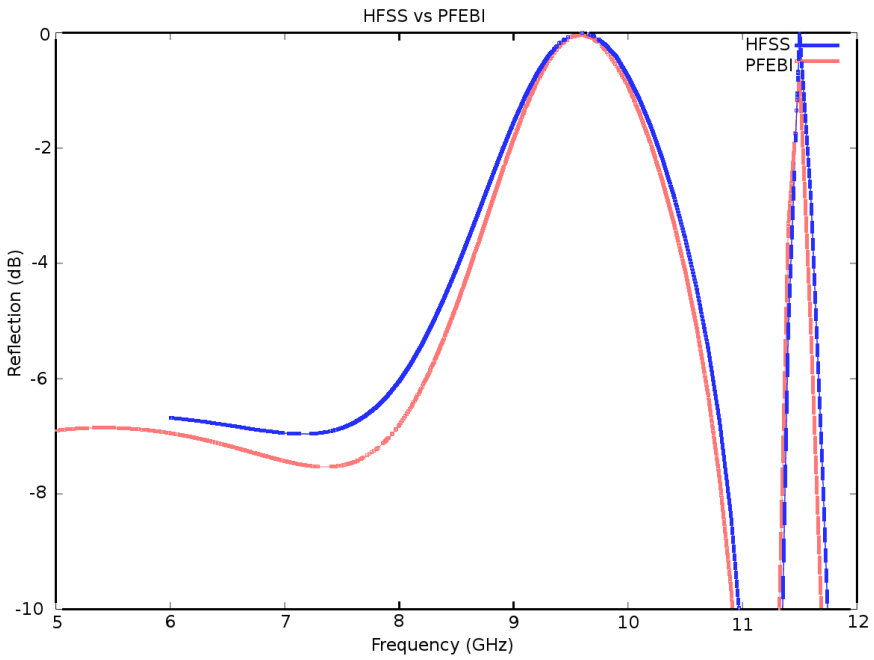
GPU FEBI Benchmark Tests

A common unit cell for FSS designs is a periodic ring structure. This is a good example to use for verification of the method. As can be seen in Figure 3-18, the response as derived from the FEBI method matches the commercial HFSS code, which uses an FEM based method. Another unit cell, seen in Figure 3-19, demonstrating an arbitrary design has also been generated using polynomial basis functions. It can be seen that in both of these examples the FEBI method accurately models the system.

Next, tests will verify that the GPU acceleration is working correctly and successfully decreases the computation time. The CPU used in this study was an Intel Xeon CPU at 2.4 GHz and the GPU was an NVidia Tesla M2090. In this case, using GPU accelerated code there is no significant difference in the frequency sweeps results. It has also been found that the time required to solve a frequency is improved. The simulation times required show a remarkable decrease when simulated on the GPU as shown in Table 3-2. With this, it becomes attractive to use a heuristic optimization technique such as Covariance Matrix Adaptation Evolution Strategy.

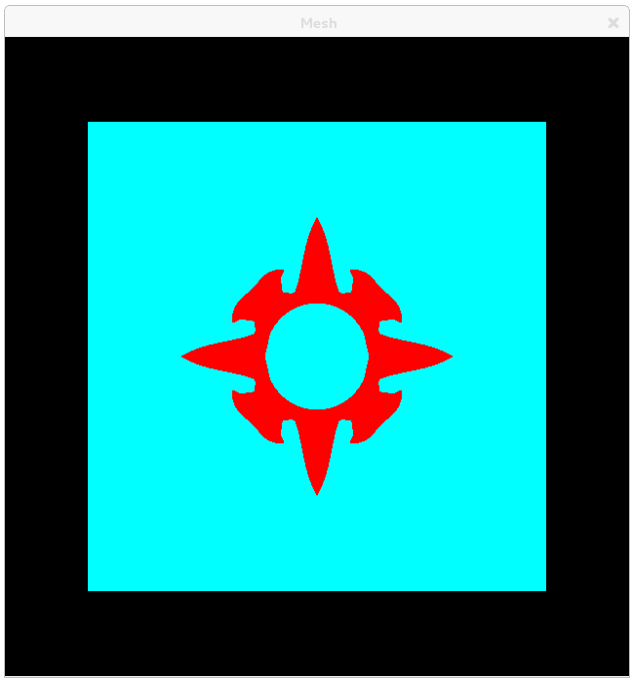


(A)



(B)

Figure 3-18: A periodic PEC ring (in red) on a dielectric substrate (A) has been evaluated (B) for comparison between the commercial HFSS and the PFEBI implementation.



(A)

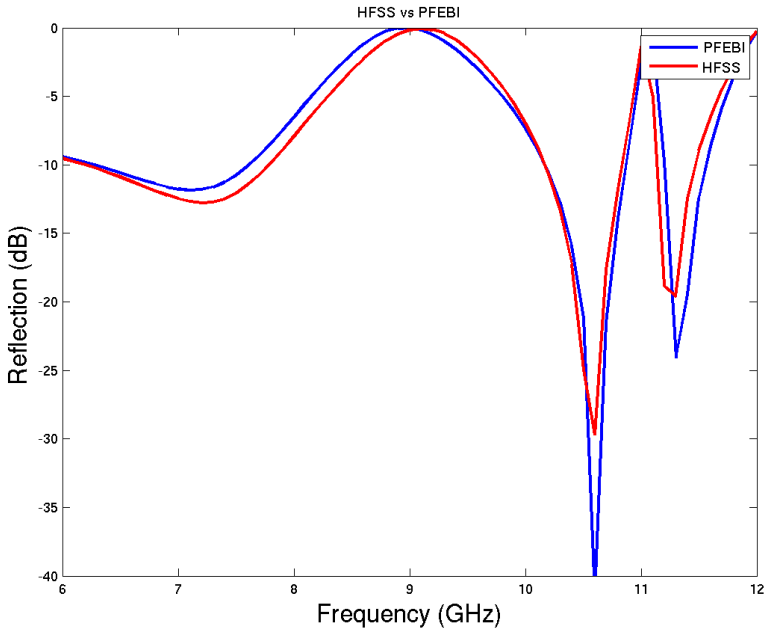


Figure 3-19: An arbitrary mesh (A) has been evaluated for comparison between the commercial HFSS and the PFEBI implementation.

	CPU (s)	GPU (s)
Ring	642	19
Arbitrary	6314	157

Table 3-2: Benchmark data for simple generic FSS designs between a CPU and GPU implementation of the PFEBI algorithm.

Iterative Methods to Solve a FEBI System

Given the finite-element boundary integral system defined by the linear system defined as

$$\left\{[Z_{FE}] + \begin{bmatrix} Z_{BI} & 0 \\ 0 & 0 \end{bmatrix}\right\} \cdot \begin{bmatrix} E^{BI} \\ E^{FE} \end{bmatrix} = \begin{bmatrix} V \\ 0 \end{bmatrix} \quad (3\text{-}14)$$

$$Z_{BI} \cdot \boldsymbol{E}^{BI} \approx [\boldsymbol{Z}_{BI(near)} + \boldsymbol{Z}^{AIM}_{BI(far)}] \cdot \boldsymbol{E}^{BI} \quad (3\text{-}15)$$

The ultimate goal is to solve for the field vector, $\begin{bmatrix} E^{BI} \\ E^{FE} \end{bmatrix}$, which contains the unknown electric field contributions for each edge element. If this system only consisted of a fully populated matrix such as what would be found from the boundary integral, a reasonable choice would be to use a direct solution technique like LU decomposition.

However, since the finite element matrix is both much larger than the boundary integral matrix and sparse, such a direct technique will oftentimes be a bad choice. This is because, in this situation, many of the eigenvalues are near zero making the system unstable.

An alternative method which is useful for large, mostly sparse systems is to use an iterative method which operates on the Krylov subspace, i.e. one in which the matrix system is successively multiplied with a vector iteratively in order to minimize the residual, or error between a guess of the solution and the true solution [35].

In this dissertation, the biconjugate gradient method has been primarily used due to its memory efficiency. However, several other more advanced methods exist as well which may

converge more quickly. These include the generalized minimum residual method [36], which requires that the residual at each iteration be saved, or the induced reduction method, which is somewhat of a compromise between biconjugate gradient and generalized minimum residual in that it requires the storage for a fixed number of residuals.

Material Parameters and Design Ideas

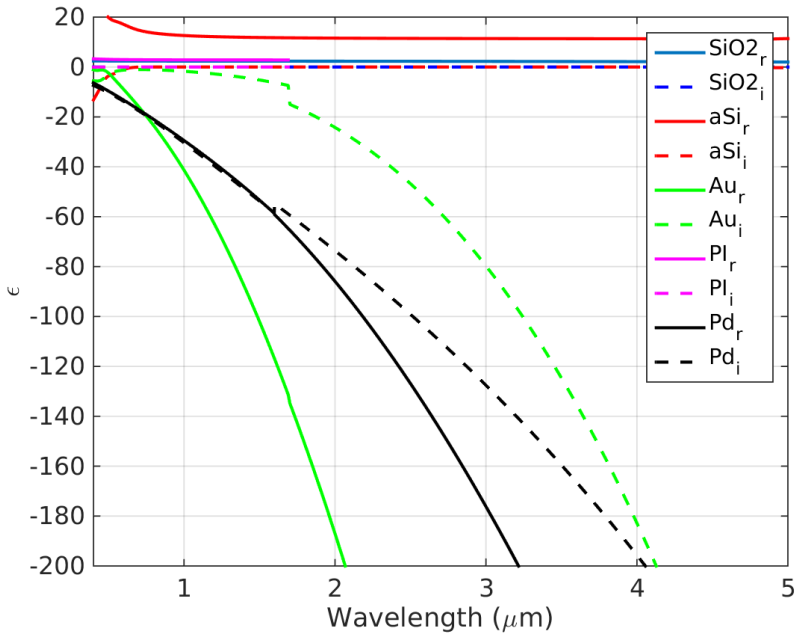


Figure 3-20: The complex permittivity for several of the materials that will be considered in this dissertation.

Finally, before designs are presented it is important to understand the properties of various materials. By knowing the typical properties for materials, one may simplify the optimization process by selecting materials that are likely to induce the properties being sought.

For example, it can be seen in Figure 3-20 that palladium is a metallic material that, using a dielectric model, has a real and imaginary permittivity that are close in optical frequency bands. In this case, palladium might be a good candidate for use in designs in which absorption is

desired. Palladium falls within the platinum group of metals and shares several characteristics with platinum. However, unlike platinum, palladium is readily available for commercial use.

On the other hand, gold is a material that has a very small real component and large imaginary component in its permittivity model. This suggests that gold might be useful for inducing changes in phase, polarization or simply for use in low-loss scattering problems. Silver and copper are other metals that share many properties with gold electromagnetically although, in general, gold tends to be an ideal candidate due to it being less lossy and not prone to oxidation like copper.

For dielectrics, the designs that will be discussed are primarily based on silicon-based glasses. Unlike the metals described above, except in rare cases, typically one would desire to use a dielectric that has a minimal imaginary component. This is because the imaginary component of a dielectric manifests itself electromagnetically typically in heat losses as opposed to metals where, in addition to losses, are primary contributors to unusual scattering patterns.

Chapter 4
Metamaterial Absorber for the Near-IR with Curvilinear Geometry Based on Bézier Surfaces

Introduction

Periodic metallodielectric nanostructures have recently been investigated by several groups for use as infrared (IR) absorbers [37, 38]. One type of IR absorber utilizes metamaterials with an effective magnetic resonance, such as negative index metamaterials. The intrinsic loss associated with their effective properties can be exploited to achieve high absorption over the resonant band [37]. Another absorber design approach is adapted from the RF and based on electromagnetic band-gap (EBG) metasurfaces. Such a device uses a lossy metallic screen that is backed by a thin dielectric layer and a ground plane [39]. An EBG metasurface was introduced in [38] with dual-band absorption in the mid-IR over a wide field of view. Many EBG designs have relied on the use of a pixelated grid to determine the placement of the metallic features. Here, a new approach that allows for a more general unit cell shape using prismatic elements whose triangle placement is determined using Bézier surfaces as opposed to one that uses brick elements may be considered. To determine an optimal surface, the powerful Covariance Matrix Adaptation Evolution Strategy (CMA-ES) algorithm has been employed to synthesize the metamaterial absorber.

Design Approach for Broadband Absorbers

The absorber structure considered here consists of a single patterned Au screen and an Au ground layer that are separated by a thin dielectric layer. The top Au screen is periodic in two

dimensions and is defined by a unit cell geometry. The screen unit cell geometry is determined using a rotated curvilinear prototype to attain four-fold symmetry. This structure can be fabricated by alternately depositing the Au ground and polyimide (PI) dielectric layers before using e-beam lithography to define the features of the top Au screen.

This single-screen absorber operates by coupling with the incident wave at one or more resonances, enhancing the fields in the structure. Energy from the incident wave is dissipated by the loss in the metallic features.

In order to synthesize the absorber structure, the CMA-ES [28, 29] optimizer linked to a full-wave periodic finite element-boundary integral electromagnetic solver has been employed. The optimization routine evolves candidate designs over multiple generations using a fitness or cost evaluation to determine the performance of candidate designs. In order to optimize the absorber structure considered in this chapter, the screen geometries are determined by thresholding a Bézier surface.

While the Bézier surface has been previously defined, a description will be reproduced here for convenience. Bézier surfaces are a two-dimensional extension of the Bézier curve. Mathematically, Bézier curves may be defined as a sum of series of Bernstein polynomials. The conventional ith Bernstein polynomial of order n is defined on the interval $0 \leq u \leq 1$ as

$$B_{i,n}(u) = \frac{n!}{i!\,(n-i)!} u^i (1-u)^{n-i} \qquad (4\text{-}1)$$

Given n+1 geometrical control points, $\boldsymbol{p}_i$, the Bézier curve $c(u)$ is defined in terms of the nth order Bernstein polynomials $B_{i,n}(u)$ by

$$c(u) = \sum_{i=0}^{n} B_{i,n}(u)\boldsymbol{p}_i \qquad (4\text{-}2)$$

Because the weighting factors are control points in the *x*-*y* plane, the Bézier curve is determined by the weighted average of the control points where the individual weighting varies with the parameter u. This allows for a geometrical interpretation of the Bézier curve as the weight of the control point is changed. As the position and weight of the control points is moved, the weights pull and tug at the overall geometrical representation, but the curve always remains smooth and inside the convex hull formed by the control point coordinates. Other polynomial formulations, such as the Taylor series, generally do not have such geometric interpretations or relationships with the resulting curve.

These Bézier curves can be easily extended to a two or higher dimensional surface [24, 25]. While a list of control points defines the Bézier curve, a grid of control points can do the same for a Bézier surface. The basis functions for this case formed by multiplying two independent Bézier curves in their respective dimensions as follows:

$$c(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} B_{i,m}(u)B_{j,n}(v)\boldsymbol{p}_{ij} \qquad (4\text{-}3)$$

where $\boldsymbol{p}_{ij}$ are the set of $(m+1)\times(n+1)$ control points. These two dimensional basis functions exert most of their influence in a region localized by the control points but are non-zero everywhere. Thus, to adapt the Bézier surface to the design of a metallic screen, a threshold on the resulting curve was applied so that every point above the threshold is determined to be Au and every point below it is determined to be empty.

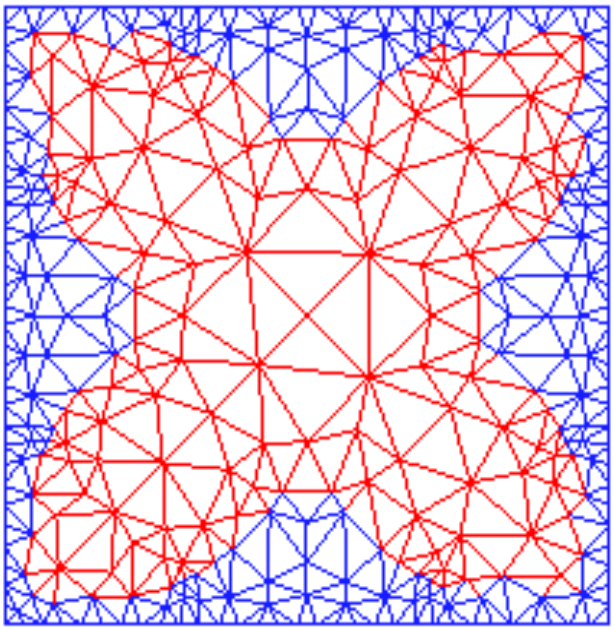


Figure 4-1: Mesh of the unit cell showing Au triangles in red and non-Au triangles in blue.

Eight-fold symmetry is also desired in the design of the Au screen, so that the absorber response at normal incidence will be polarization independent. This symmetry condition can be achieved by first forcing the control point matrix to be diagonal. This forces the Bézier surface to be symmetric along the diagonal, forming one quadrant of the overall unit cell. The Bézier surface is then rotated three times to fill in the remaining quadrants of the unit cell. The parameters to be optimized using CMA-ES include the diagonal control point values, the unit cell dimension, and the thicknesses for the screen and dielectric layers. Material properties for Au and PI based on experimental measurements [40] are incorporated into the fitness evaluation. The *Cost* function employed is given by

$$cost = (1 - A)^2 \qquad (4\text{-}4)$$

where A is the absorption magnitude at each specified wavelength λ_i.

In addition, a second absorber was designed. In this case, the optimization was more robust while scanning over a larger frequency and incident angle range. Like the previous design, a gold FSS screen was optimized above a palladium ground plane. However, unlike the previous design, the ground plane uses material parameters for a ground plane. This was chosen due to its properties as a p-type semiconductor. Furthermore, the Bézier surface control point matrix was determined without enforcing symmetry conditions. By neglecting symmetry conditions, the resulting surface is allowed more variability at the cost of a loss of polarization independence.

Design Results

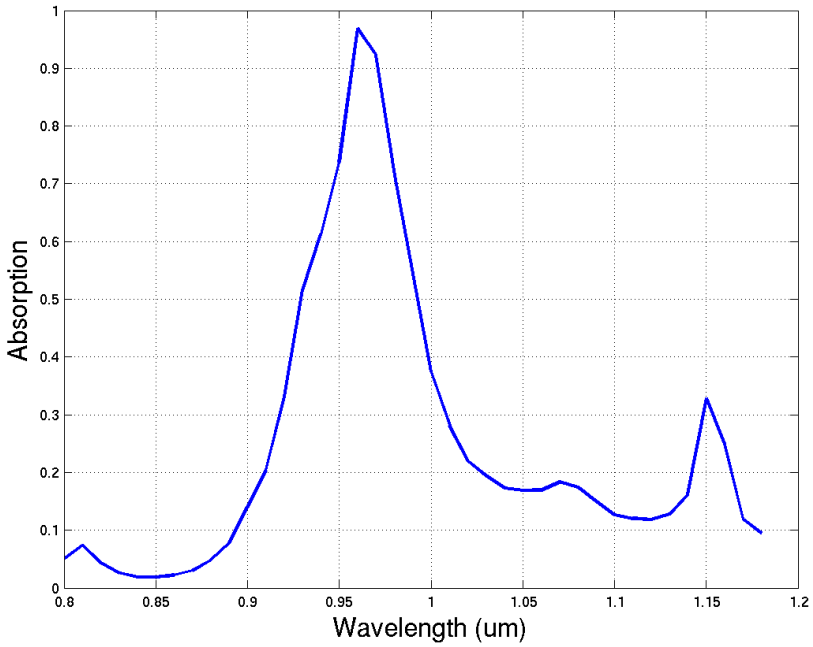


Figure 4-2: Absorptivity plot for design in Figure 4-1 showing a peak value of 96.9 % at 0.96 μm the design frequency.

The CMA-ES was employed to optimize an absorber with a target wavelength of 0.96 μm. A population of 12 members was optimized over 100 generations to evolve the geometry shown in Figure 4-2. The unit cell dimension (period) for the optimized geometry is 0.79 μm, and

the layer thicknesses are 20 nm, 318 nm, and 200 nm for the Au screen, PI substrate, and Au ground, respectively. The predicted absorption coefficient for the optimized design is plotted verses wavelength in Figure 4-2 for normal incidence. The absorption is high at the design wavelength, with a peak absorptivity of 96.9% and reflection suppressed to less than -15.2 dB.

The second absorber can be seen in Figure 4-3. This surface was meshed as a unit cell with a threshold applied to achieve a prismatic mesh for use in a finite element boundary integral solver. This unit cell was optimized to be a square with 256 nm sides. The screen layer was optimized between two 153 nm thick polyimide layers sitting atop a ground plane. Palladium silicide was chosen as the screen to encourage a high absorption response as well as for its properties as a semiconductor. Finally, the frequency response over the band of interest, $0.5 - 1.8\ \mu m$, is plotted in Figure 4-4 with a doped p-Silicon ground plane.

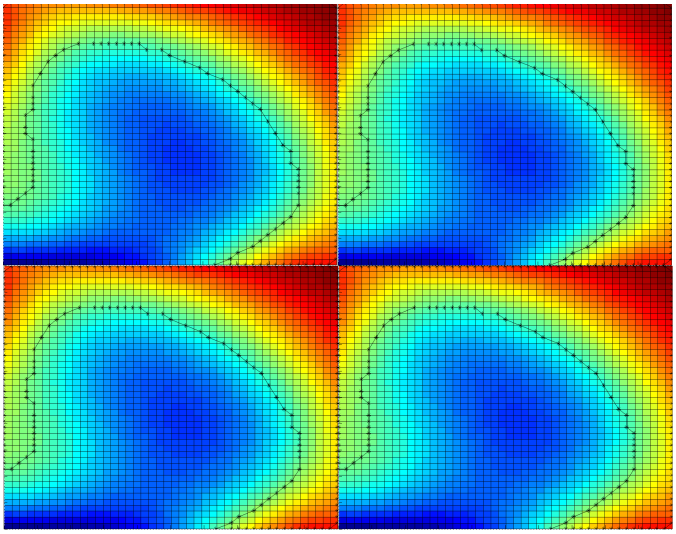


Figure 4-3: The Bézier surface used for mesh generation of the wide-band, wide-angle absorber. Surface values over a threshold, seen here in light green to red, are meshed as Pd2Si. Values below the threshold are meshed as polyimide.

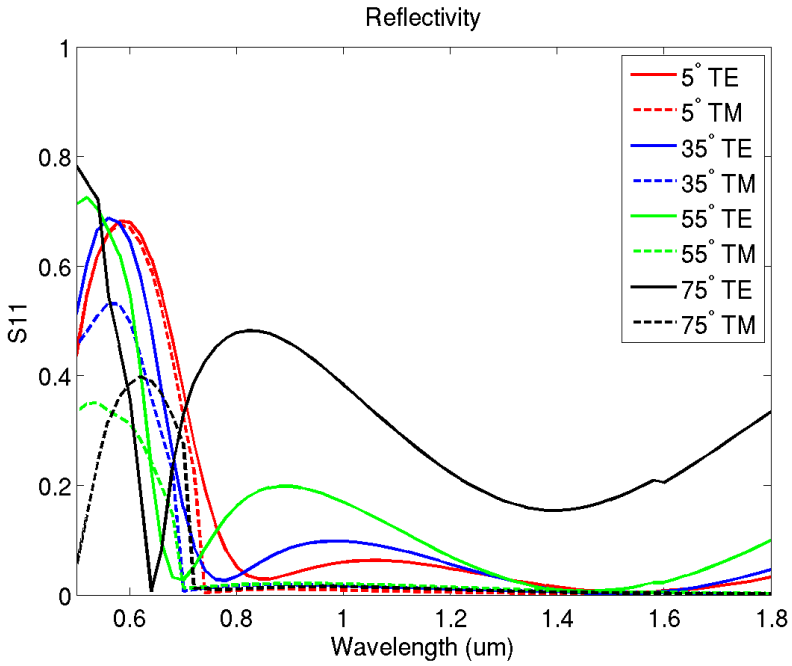


Figure 4-4: The frequency response for the optimized Bézier surface over the optimized frequency band.

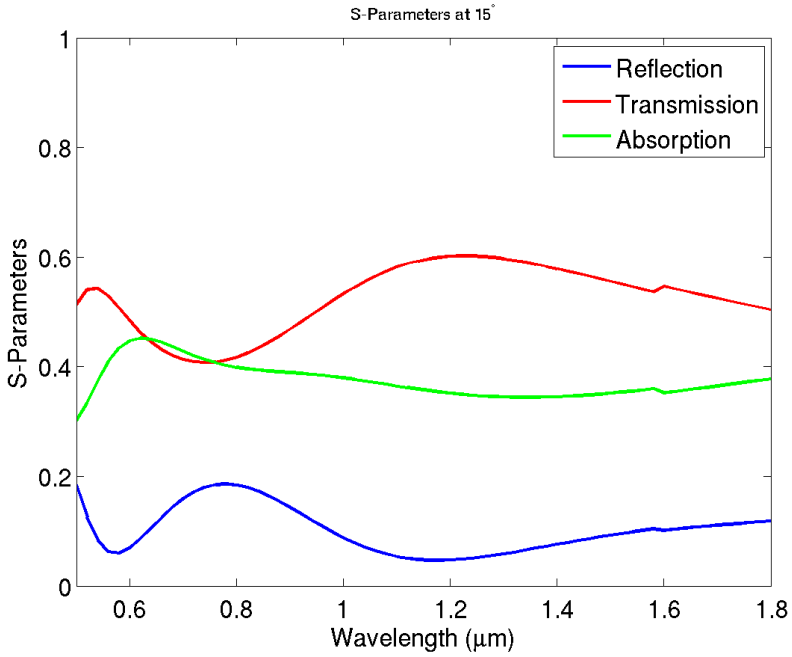
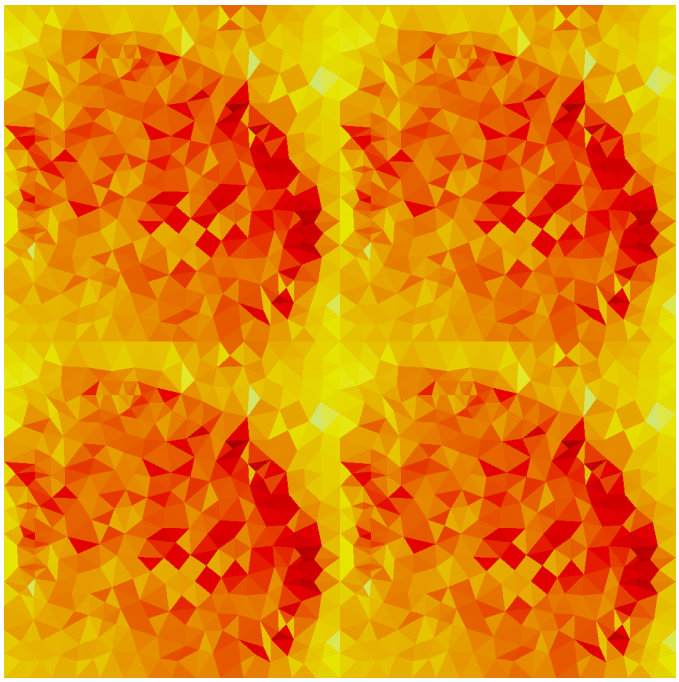


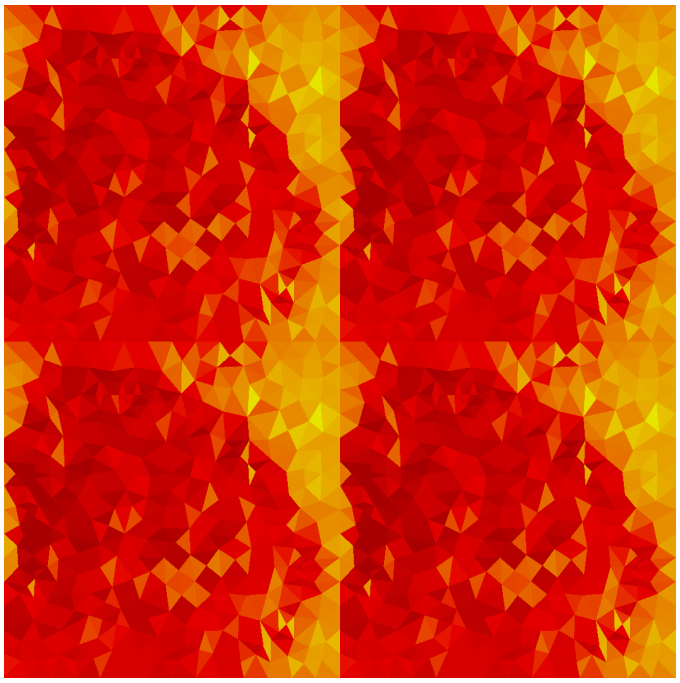
Figure 4-5: The frequency response of the optimized surface with the ground plane removed.

One concern that one may have with this design is what the source of the source of the absorption is. If the majority of the energy is being absorbed as heat into the palladium screen, for

instance, this design may have a limited usefulness. In order to test this, two things have been done. First, the finite element field strength can be sampled at each triangle edge immediately above and below the screen. If the losses primarily occur on the screen, one would expect that the fields below the screen would be significantly lower than those above. However, it can be seen in Figure 4-6 and Figure 4-7. However, in the case of the shorter wavelengths, the field is in fact stronger below than above suggesting that the fields are confined beneath the screen forcing the energy to be lost within the substrate. In the longer wavelength case, the fields interact more strongly with the palladium inducing some loss on the screen. However, it does not appear to be significantly large and the fields below remain strong.

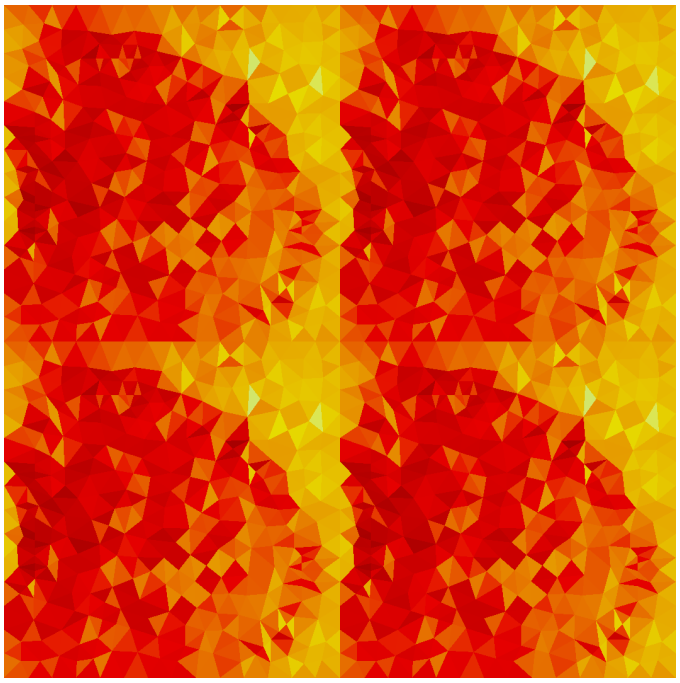


(A)

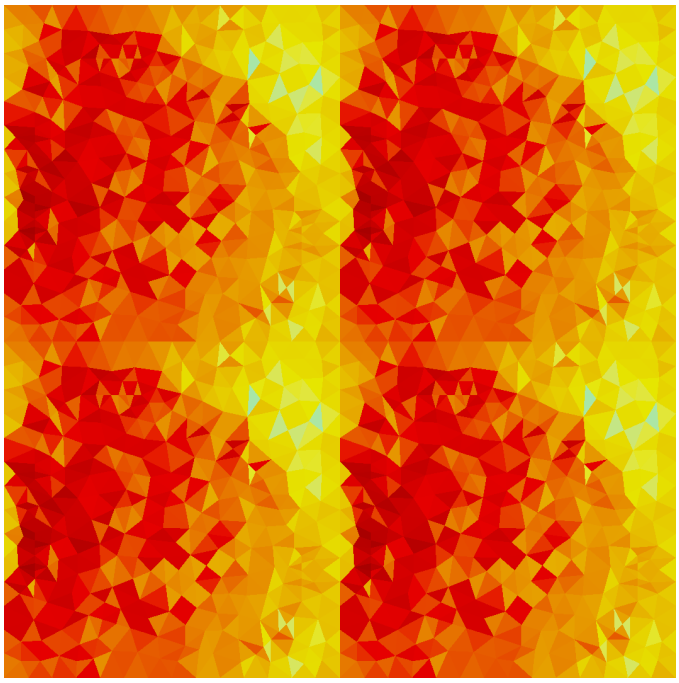


(B)

Figure 4-6: The finite element field plots above (A) and below (B) the screen at an incident frequency of $0.8\ \mu m$*. Values close to or above* $0.5\ V/m$ *the fields are redder. The fields along the edges are averaged to give each triangle a solid color value.*



(A)



(B)

Figure 4-7: The finite element field plots above (A) and below (B) the screen at an incident frequency of $1.2\ \mu m$*. Values close to or above* $0.5\ V/m$ *the fields are redder. The fields along the edges are averaged to give each triangle a solid color value.*

By replacing the substrate with a standard glass, there is further confirmation that this filter works primarily by suppressing the amount of energy that is able to reflect as can be seen in Figure 4-5. Still, because palladium was used as a screen, the absorption is somewhat high at around 40 % through the simulated band. This is to be expected since palladium is a highly lossy metal. To account for this, a less lossy metallic screen could be used. This approach, however, comes at the cost of a loss of semiconductor properties in the screen.

As an alternative screen, silver has been chosen as an example over gold because the permittivity of silver is further from the permittivity of semiconductor substrate. Figure 4-8 shows a frequency sweep of this filter from near normal-incidence to near grazing incidence. Because this filter was not optimized using parameters for silver, it reflects light more strongly than what

one would normally desire although there is strong confinement of the electromagnetic energy into the substrate. However, the energy in the visible spectrum is still strongly suppressed with an overall reflection remaining below 50 % throughout the band of interest, $0.6\ \mu m - 1.6\ \mu m$ except at extremely oblique angles.

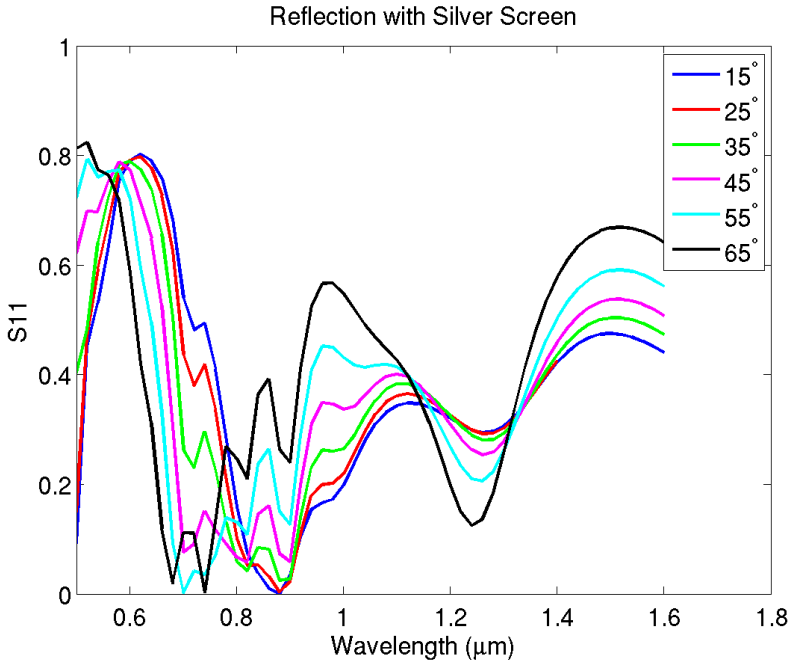


Figure 4-8: A frequency sweep of the previous FSS with a silver screen used.

This suggests that the light reacts strongly with the metals at high frequencies due to the plasma effects of the screen that are occurring due to the curved shape of the surface. At frequencies between $0.6\ \mu m$ and $1.4\ \mu m$, because the wavelength of the light is reduced due to the glass medium, the wavelength of the light is small enough that it may pass between the screens while large enough that it is not resonating. In this region, the light passes through the screen into semiconductor substrate where it is absorbed. Finally, at the lowest simulated frequencies, the wavelength of the light begins to become large enough that it must interact with the metals causing large amounts of energy to be reflected.

Conclusion

Bézier surfaces were presented as a possible way to represent arbitrary FSS unit cells in a prismatic two-dimensional periodic mesh. The CMA-ES optimization method was used to generate a Bézier surfaces for absorbing FSS problems. One design, optimized for a single frequency point using a single screen exhibited polarization independent absorption at $0.96\ \mu m$. A second design, which produced above 90 % absorption, polarization-independent response at oblique incidence over two octaves. A nature inspired spiral design was shown that was able to achieve a broadband absorption. Future research can be done into the use of different materials for use as a screen, the direct field response along the spirals as a function of frequency as a way to tune the absorption as well as the use of a unit cell with a deeper spiral in order to shift the anti-reflective properties to a higher frequency. The techniques described here can be further explored to design multi-band or broadband absorbers as well as other metamaterials for the near-IR.

Chapter 5

An Oblique-Angle Infrared Circular Polarization Filter Using a Bézier Surface Representation

Introduction

The polarization state of light has many applications such as optical communication [41] and imaging [42, 43]. Many devices have been presented which can be used to induce a circular polarization on light from fiber Bragg gratings and stacked subwavelength gratings [44, 45], or liquid crystal technology [46].

The atmosphere of the Earth is comprised of many gases. While the major components of the atmosphere are nitrogen, oxygen and argon, a relatively small percentage is comprised of what is known as the greenhouse gases. These greenhouse gases get their name due to having vibrational absorption bands that correspond with parts of the infrared that convert electromagnetic energy into heat. Of these, there is particular interest in the concentration of CO2, which has an absorption band at 4.3 μm. A challenge in measuring CO2 concentration lies in the nearby absorption band for N2O, which lies at 4.5 μm. Commonly, these can be measured using a cascade laser, which can finely sweep and measure the frequency response in this band [47].

In this chapter, a technique to synthesize an optical polarization filter using thin film frequency selective surface (FSS) techniques. This process is initially proposed for the use in the mid-infrared, which uses a frequency selective surface to isolate the circular polarization component precisely at the CO2 absorption band. Following this, a process will be presented that

will allow the design to be adapted for use at the important communication frequency of $1.55\ \mu m$ which has minimal dispersion and loss in silica fibers.

Mesh Design

Mesh Design Using a Bézier Surface

Production of optical frequency selective surfaces oftentimes is based on modeling of a pixelated model. While the calculation of electromagnetic responses of these models is straightforward, the pixelization lends itself to a design that relies on many sharp corners/edges as well as the potential for extremely sub-wavelength features. Production of these features on a nanoscale is quite a challenge, which may produce a discrepancy between the performance of the simulated and the manufactured designs. For this reason, a curvilinear frequency selective surface based on a Bézier surface representation [24, 25] and the periodic finite element boundary integral method has been used. The Bézier surface itself is a two-dimensional expansion upon the idea of the Bézier curve. Adapting the Bézier curve to a two-dimensional system as in (5-1) creates the Bézier surface where $\boldsymbol{k}_{i,j}$ is a list of control points which map onto the u, v plane. B_i^n is a Bernstein polynomial.

$$p(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u) B_j^m(v) \boldsymbol{k}_{i,j} \qquad (5\text{-}1)$$

This representation differs from the typical interpolation scheme in which the surface must pass through the control points. Instead, in a Bézier representation, the control points attract the surface. Typically, the surface acquired from a Bézier representation resembles its control point matrix. This provides an advantage in an optimization scheme in which a designer may

encourage certain symmetries or shapes by building them into the control points of the Bézier surface.

Optimization of Bézier Surfaces Using the Covariance Matrix Adaptation Evolutionary Strategy

The Bézier surface can be readily incorporated into an evolutionary optimization scheme. The Bézier surface is capable of representing continuous surfaces with a limited number of variables. Because of this, one may define an optimization in which the control points and other parameters useful in designing a frequency selective surface, such as layer thickness or unit cell size, into a global search scheme such as the Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES). CMA-ES [28, 29], like the well-known Genetic Algorithm (GA), is a derivative-free stochastic numerical optimization routine. In this design, a unit cell comprised of hundreds of triangles using a control point matrix with only a handful of variables is used. CMA-ES, which has an optimal population size of $pop = 4log10(N) + 3$, is able to converge quickly to an optimal design. In addition to the control points and size parameters, an additional variable has been incorporated into the CMA-ES optimization the threshold that is used to determine the cutoff for what regions in the Bézier surface are considered metal and what are considered a dielectric.

Optimization for Circular Polarization

In designing a circular polarization filter, one may take advantage of the fact that any incident wave can be considered as a sum of left- and right-hand circularly polarized waved. That is,

$$\boldsymbol{E_i = E_{lhcp} + E_{rhcp}} \tag{5-2}$$

Mathematically, any polarization may be described by the Stokes vector $\boldsymbol{S}$. However, in designing an electromagnetic system, it is often more convenient to consider instead the Mueller matrix, $\boldsymbol{M}$, which acts on the polarization vector [48]. That is,

$$\boldsymbol{S_{out} = M \cdot S_{in}} \tag{5-3}$$

These parameters can be determined by calculating the electromagnetic interaction of incident waves. The relevant Mueller parameters can be represented in terms of reflected fields as

$$\begin{pmatrix} \boldsymbol{m_{11}} \\ \boldsymbol{m_{12}} \\ \boldsymbol{m_{13}} \\ \boldsymbol{m_{14}} \end{pmatrix} = \begin{pmatrix} \boldsymbol{0.5(|E_{11}|^2 + |E_{12}|^2 + |E_{21}|^2 + |E_{22}|^2)} \\ \boldsymbol{0.5(|E_{11}|^2 + |E_{12}|^2 - |E_{21}|^2 - |E_{22}|^2} \\ \boldsymbol{real(E_{11} \cdot E_{21}^* + E_{12} \cdot E_{22}^*)} \\ \boldsymbol{imag(E_{11} \cdot E_{21}^* + E_{12} \cdot E_{22}^*)} \end{pmatrix} \tag{5-4}$$

Here, the term m_{14} represents the circular polarization of the reflected wave. Due to stability conditions [48], it is sufficient to consider only these terms in an optimization of Mueller parameters when a specific Stokes polarization response is desired.

A Circular Polarization Filter at $4.3\ \mu m$

A frequency selective surface was optimized to separate circularly polarized components of an incident wave at 4.3 μm. That is, a left-hand circularly polarized wave will be reflected and a right-hand circularly polarized wave will be transmitted. A few unit cells of the meshed Bézier surface can be seen in Figure 5-1. The frequency response of the circular polarization normalized to reflection intensity is shown plotted in Figure 5-2. It can be seen that the ratio of circularly polarized light to reflected intensity at 4.3 μm is 88 %. Furthermore, the light at 4.5 μm, which can be considered noise from the nearby N2O band, is completely reflected. One may calibrate the system by incorporating a 4.3 μm circular polarization detector a distance behind the filter.

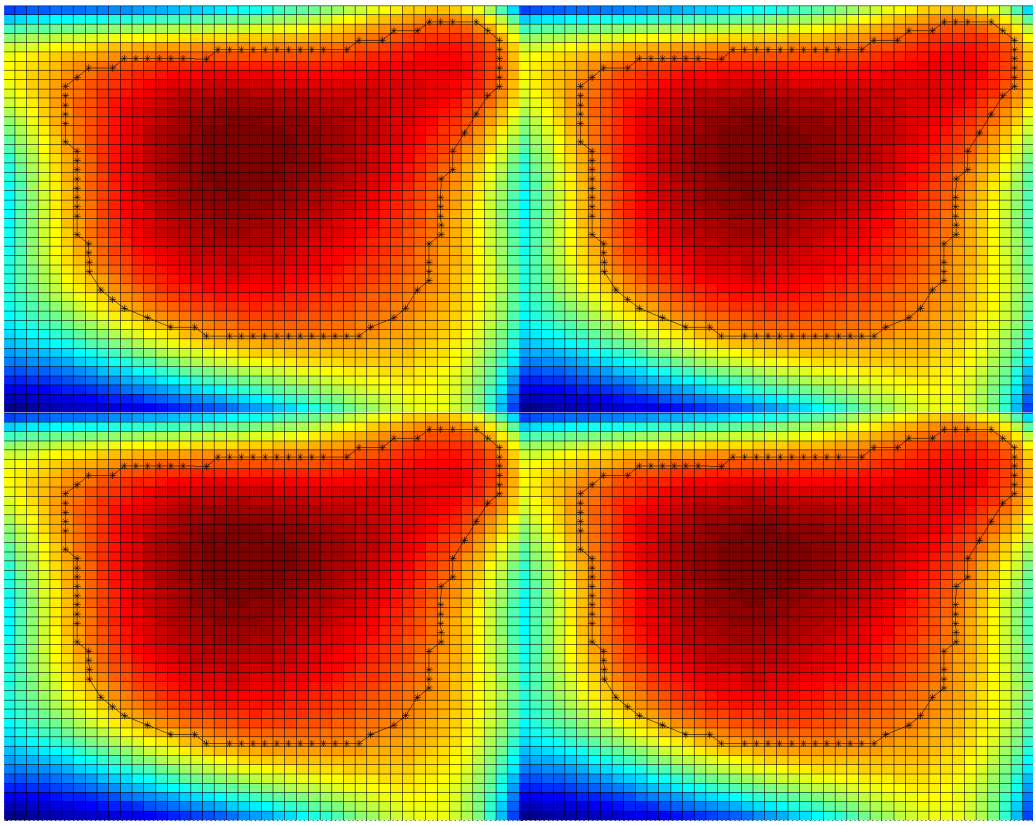


Figure 5-1: Model of the frequency selective surface showing a 2x2 array of unit cells. In this Bézier surface representation, a black outline represents the boundary between the parts of the surface above and below the threshold value with red being the screen.

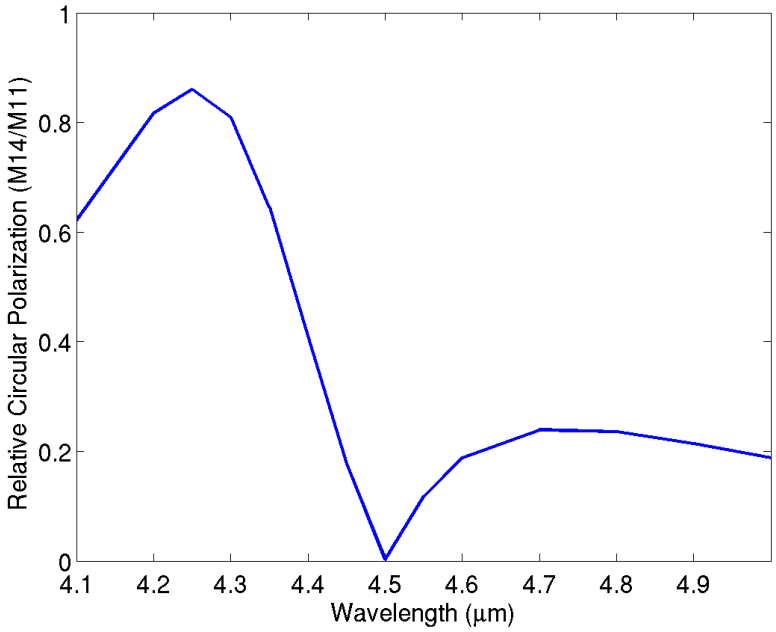


Figure 5-2: A plot of the normalized circular polarized light as a function of wavelength.

A Circular Polarization Filter Optimized for the Telecommunications Band

An advantage of this design is that it is simple and easy to scale to various frequency bands. As an example, using the Bézier surface shown in Figure 5-1 as a basis, a circular polarization filter targeting the telecommunications band surrounding $1.55\ \mu m$ was optimized. With this basis, the CMA-ES optimization was able to find a solution within six generations arriving at the mesh shown in Figure 5-3. In this case, the unit cell period is $653\ nm$ with a $92.6\ nm$ screen sandwiched between two $114\ nm$ glass layers.

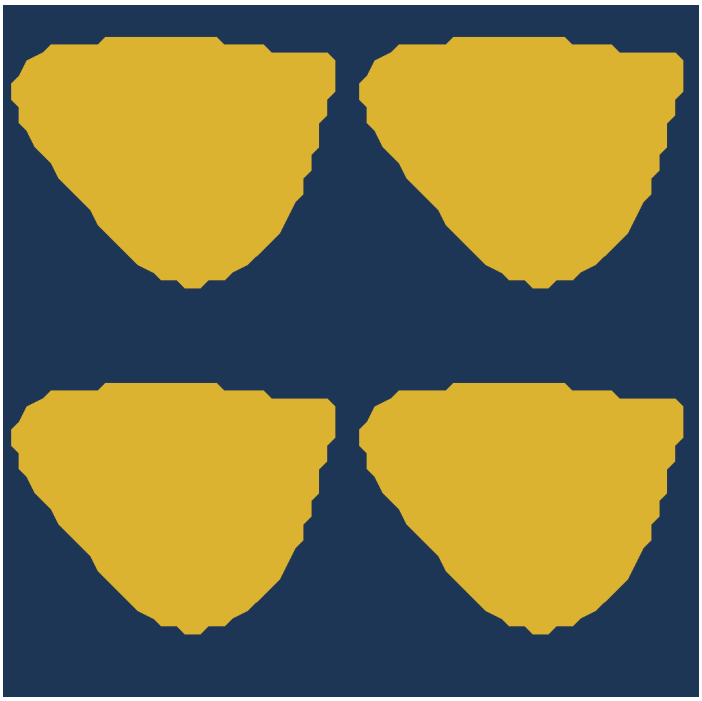


Figure 5-3: The circular polarization filter re-optimized for use in the telecommunications band. The metallic screen is shown in gold and the dielectric is dark blue.

In this band, the gold is less lossy. This allows the filter to reflect the circular polarization components more easily in a wider band. At the optimized angle of $35°$, nearly the entire short, conventional and long wavelength bands are above 50 % relative circular polarization yielding either a highly elliptical or a purely circular polarization. Furthermore, at $1.55\ \mu m$, the relative circular polarization remains above 50 % within $\pm 25°$ of the optimal angle. This would allow an operator plenty of leeway in the angle at which the filter is connected.

As in the previous filter, this design operates by reflecting one circular polarization and transmitting the other. In this case, it could see use in encoding an optical signal using a technique such as Dual Polarization Quadrature Phase Shift Keying [49].

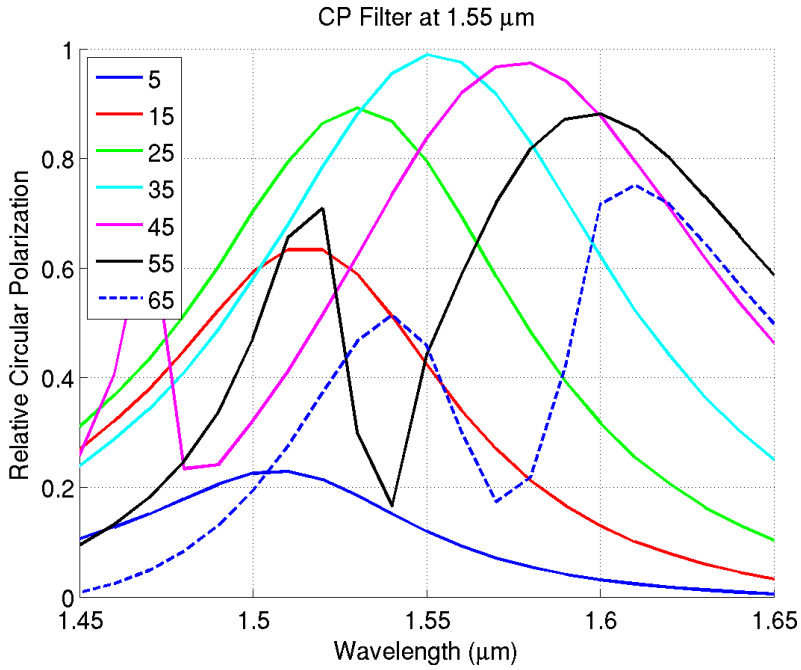


Figure 5-4: The relative circular polarization for several angles centered on the 1.55 µm band.

Conclusion

A simple frequency selective surface was optimized to isolate circularly polarized waves centered at the CO2 absorption band of 4.3 μm. This design successfully provides a sharp contrast between the intensities of lights at the CO2 and N2O bands allowing a simple optical system to observe greenhouse gases without overlapping bands.

The adaptability of this design was shown by quickly re-optimize the filter for use in the fibre-optic telecommunications band. This design showed strong lossless unity relative circular polarization at the center of the important conventional band. It was further suggested that this design might be useful for modulation purposes.

Chapter 6
A Highly Transparent Conductive Surface Design Based on an Optical Frequency Selective Surface

Introduction

With uses ranging from consumer applications like LCD and touch screens to EMI shielding, transparent antennas and photovoltaics, the market for transparent conductive materials has seen rapid growth. Meanwhile, concerns about the global supply and price of indium [50], used in indium tin oxide (ITO) a common transparent conductive material, have created a demand for alternatives, which provide high transparency, low resistivity and ease of manufacturing.

There has been considerable interest into ITO alternatives, which are chemically similar. That is, materials that exhibit semiconductor properties that are transparent within the visible spectrum. Popular materials include aluminum-doped zinc oxide (AZO) [51] and gallium doped zinc oxide (GZO) [52]. These alternatives, while cheaper, do not fully achieve the performance that one would expect in a commercially produced sample ITO failing to achieve the level of transparency or conductivity seen in ITO.

In addition to transparent conductive oxides, there has been additional research in using silver nanowires [53] and carbon nanotubes [54] for a similar effect. These techniques have shown promising results yielding low sheet resistance and high levels of transparency. Indeed, Madraria et al were able to achieve films with 85 % transparency at $550\ nm$ on a plastic substrate using a dry printing technique [55]. While transparent conductive films designed from nanowires is a promising approach, most research has looked into solutions of randomly arranged nanowires.

Another alternative approach that has not seen much research is in an ordered nanowire technique. In other words, applying frequency selective surface design techniques the problem. Frequency selective surfaces (FSS) are periodic structures that are engineered to achieve a particular electromagnetic response over a particular frequency band. With semiconductor etching techniques providing for smaller and more complex transistors and, more importantly, metallic interconnects this approach becomes more attractive. Rather than relying on rare materials or growing random meshes of nanowires, one may instead apply a mask and print a well-defined FSS.

As transistor-manufacturing processes allow for smaller transistors, technologies to produce complicated, metallic nanowire networks have matured. For example, the semiconductor industry, in moving to multi-gate architectures such as FinFET [56], may now mass-produce three-dimensional arrays of contacts. In addition to complex wire-networks, the metallic contacts may be produced at even smaller sizes. IBM, for instance, recently reported FinFETs, which had gate widths as low as 8 nm [57].

A successful transparent conductive surface must meet three general requirements. It must achieve a high degree of transparency, ideally with an average visible transparency of above 80 %, low dispersion and a wide field of view. It must have a low resistivity. It must have a well-defined method of production that could be applied on a massive scale. In this chapter, I will propose an optical FSS based design that achieves all three goals.

Designing an Optimized Nanowire

Optical Frequency Selective Surfaces from Nanoscale Interconnect Technology

FSS technologies are well-understood devices that are used extensively from radio to optical bands of the electromagnetic spectrum. These surfaces generally consist of periodic arrangement of materials in order to achieve some desired frequency response. Commonly, these materials are metallic although it is common in the optical regime to use only dielectric materials in an FSS [58].

In FSS design, it is common for the space between periodic features, otherwise known as the unit cell size, to be somewhere on the order of the wavelength which is being targeted by the unit cell. Thus, as the frequency of the light increases, the unit cell size must by extension decrease. For one to design an FSS that operates within the visible spectrum, the unit cell size must be on the order of a few hundred nanometers. At this size, one must take special care in a design to ensure that the features of the unit cell, which are necessarily smaller, are realizable.

Fortunately, with constant advances in transistor technologies, it is now possible to consider nanowires, which are only tens of nanometers in diameter. Major semiconductor manufacturing companies are now producing semiconductors with 14 nm half-pitches. By necessity, gates of this size require nanowires, which are of a similar size.

Recent MOSFET technology has moved towards three-dimensional transistors such as those seen in FinFET technologies. Immersion lithography [59], in particular, is an attractive technique for etching trenches and memory cells with feature sizes in the tens of nanometers. For example, 193 nm immersion lithography has been used to a size to produce 22 nm CMOS cells with fin channels with widths of 8 nm [57]. These transistors, in addition to exhibiting small

feature sizes demonstrate complex wire-networks requiring interconnects that are of a similar order.

The resistivity of nanoscale copper interconnects is well documented in the annual Interconnect Roadmap Report [60]. It is known that as the width decreases, the copper becomes less able to facilitate current flow thus increasing resistivity. For this reason, as a design objective, wire width has been encouraged to be above 40 nm because at interconnect line widths below this threshold the resistivity of the lines begins to grow somewhat rapidly. Keeping the line widths somewhat large will allow for a design that more easily competes with randomly dispersed nanowires, which have somewhat large bulk resistances on the order of 30 Ω/sq.

Mathematical Description of the Unit Cell

A further design requirement was in geometrical complexity. It is desirable to use a geometrical representation that allows for continuity and generality. The unit cells have been subdivided into triangles to allow some complexity and freedom in the definition of the shape but rather than optimizing each triangle independently, a Bézier surface representation has been used.

Bézier surfaces are mathematical surfaces, somewhat related to an interpolation technique, which are often seen in computer aided design or graphics programs. A Bézier surface representation serves this purpose well because it provides a method to describe a nearly infinite number of shapes with a limited number of variables.

While the description of a Bézier surface was discussed in an earlier chapter, I will reproduce it here for reference. Mathematically, a Bézier surface is an N-dimensional spline. In two-dimensional space, a Bézier surface is a sum of independent Bernstein polynomials. A general form of the ith polynomial of an nth order Bernstein polynomial can be written in the u-dimension as in (6-1).

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i} \quad (6\text{-}1)$$

Thus, a Bézier surface of degree (n+1) by (m+1) can be written as in (6-2).

$$p(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u) B_j^m(v) k_{i,j} \quad (6\text{-}2)$$

By selecting a high degree Bézier surface, one can describe a surface with a higher degree of complexity. A 5th order Bézier surface in both dimensions was chosen as a tradeoff between generality of the surface and manufacturability of the metallic structure. This choice yielded 25 control points to be optimized. At this level of complexity, the surface easily facilitates curvilinear wires without risking large rates of change in the surface.

By consolidating the design into a limited number of variables such as Bézier control points, unit cell size, and wire thickness, it becomes simple to use a stochastic optimization model such as covariance matrix adaptation evolution strategy (CMA-ES) to develop an effective unit cell [28, 29]. Finally, to map the Bézier surface to a two-dimensional unit cell, an additional optimization parameter was used as a surface threshold term. Using this threshold value, all points on the surface above the threshold are mapped in the mesh-generation code to copper and all values below the threshold are mapped to a glass.

As an evolutionary algorithm, CMA-ES updates a random population over several generations in an attempt to move towards a local minimum. Unlike the well-known genetic algorithm, CMA-ES moves towards its goal by determining an approximation of the covariance matrix of the solution space. Each generation, the algorithm approximates the inverse Hessian matrix allowing the search space to be updated similar to a quasi-classical optimization.

The search space was initialized using a Bézier surface, which approximated rounded-edge mesh unit cell. As a fitness function, a root mean square (RMS) of several per-wavelength

fitness values was used. At each wavelength, the fitness was a weighted RMS calculated as in (6-3).

$$fit_{\lambda} = \sqrt{0.4R^2 + 0.6(1-T)^2} \qquad (6\text{-}3)$$

Such that R is the reflected power and T is the transmitted power. This sort of fitness function will allow attempt to find an arrangement that will yield a high degree of transparency while forcing the lost energy to be absorbed rather than reflected through the inclusion of the reflection coefficient in the fitness function. Fitness values were calculated at wavelengths of 400 nm, 500 nm, 700 nm and 900 nm alternating between horizontal and vertical polarizations. Furthermore, to encourage a wide field of view, fitness values were calculated at both normal and 25° incidence.

Simulations were done using the periodic finite-element boundary integral algorithm described earlier. A prismatic mesh was used to simplify meshing for a layered two-dimensional periodic surface. The nanowire layer was simulated such along with two layers above and below of glass with a glass half-spaces on top and bottom as in Figure 6-1. This allowed the finite-element code to quantify field interactions near the interface while compressing the simulation domain with the boundary integral. Using half spaces above and below the FSS should be accurate to a manufactured structure if the half spaces are several wavelengths thick.

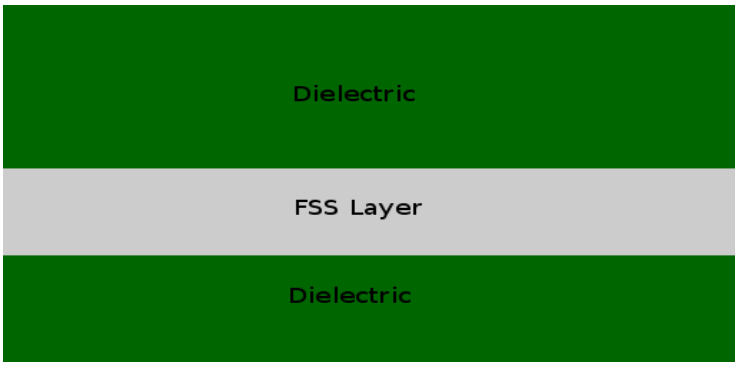


Figure 6-1: A side view of the FSS. The metallic FSS layer is between two thick dielectric half spaces

Simulation Results and Discussion

CMA-ES reached a minimum after 19 generations with the Bézier surface seen in Figure 6-2.

The optimized FSS has a metallic concentration of 24.8 % and a thickness, t, of 45.3 nm. The nanowire varies in width throughout the unit cell from a minimum of 47.0 to 124.8 nm and a total unit cell size of 285.3 nm. Using an estimated resistivity, ρ, of 4 μΩ/cm as a basis, which is slightly lower than the typical resistance of a 38 nm half-pitch interconnect, an estimated sheet resistance of 3.56 Ω/□ can be calculated using (6-4).

$$R_s = \frac{\rho/concentration}{t} \qquad (6\text{-}4)$$

This compares well with ITO thin-films, which have conductivities between 20 and 25 Ω/□ [62].

Frequency sweeps were performed using the optimized FSS with a variety of materials from 400 to 750 nm using the periodic finite-element boundary integral method with Figure 6-3 showing a $2x2$ grid of meshed unit cells. Frequency sweeps were performed using the optimized FSS with a variety of materials. Because the wire size is on the order of where the resistivity of copper begins to become similar to the resistivity aluminum, frequency responses were calculated using both materials. The metal was also simulated using the properties of silver due to its high conductivity. For dielectric, FSS designs were simulated using polyimide for its thermal and mechanical properties and a constant refractive index of 1.523 which is the refractive index of Corning 0211 flexible glass at 589.3 nm [61]. Sweeps were also performed with angles of incidence ranging from 0 to 65° to measure expected response over a wide field of view.

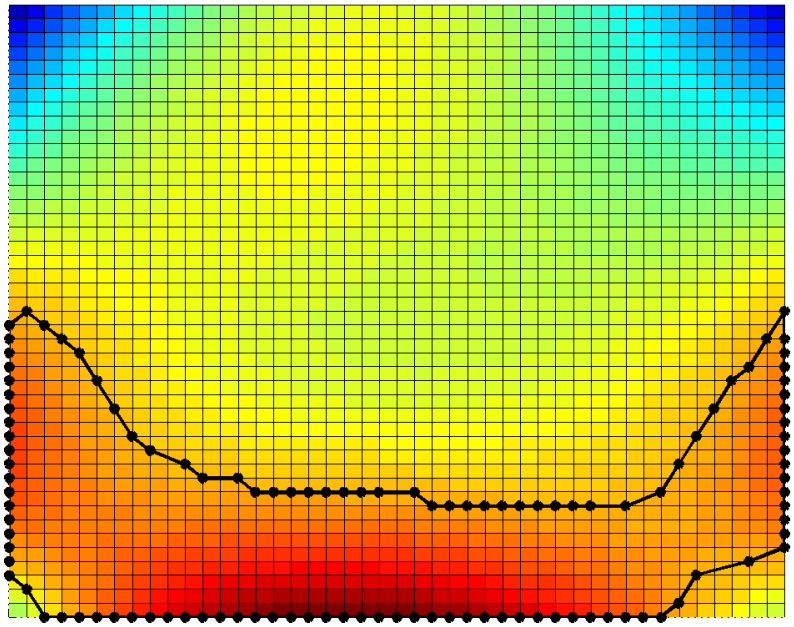


Figure 6-2: An optimized Bézier surface. The black outline shows the threshold points along the surface.

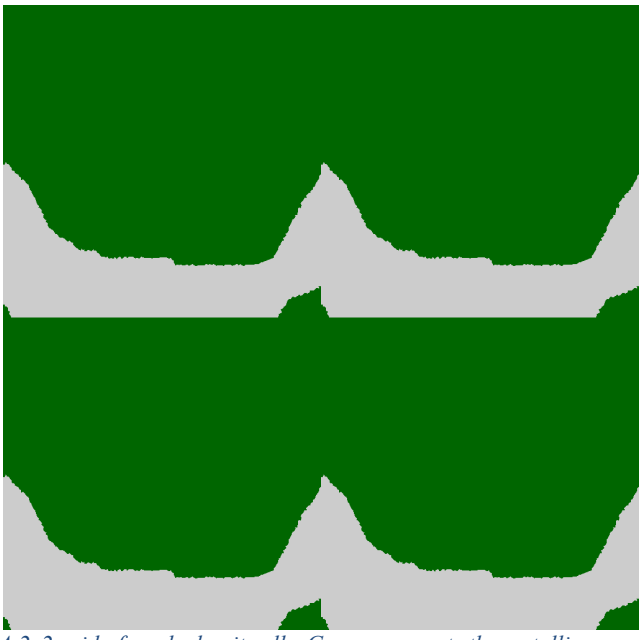


Figure 6-3: A 2x2 grid of meshed unit cells. Gray represents the metallic screen and green represents dielectric substrate.

Figure 6-4 shows the results of frequency sweeps at angles of incidence from 0 to 65°. As can be seen in the dashed lines, the filter shows strong transparency up to 65° where it has a mean transparency of 51 %. Meanwhile, the reflectivity of the surface, which goes up to 11 % at 65°, is relatively low over all angles. This is advantageous over a surface like ITO where the energy, which does not pass through the surface, is reflected. In this FSS design, the lost energy is absorbed as heat, which in a consumer device like a phone should provide a nicer viewing experience.

Another advantage of this design is that, aside from the violet and far-red portions of the spectrum, the response has relatively low dispersion. For example, at normal incidence, transmitted light energy varies from 81 to 85 % between 450 and 650 nm.

The results in Figure 6-4were taken by averaging the transverse electric (TE) and transverse magnetic (TM) polarizations. This can be done since unpolarized light can be thought of as being one or the other 50 % of the time. However, by looking at each polarization directly, as in Figure 6-5, one can get an idea of the physics that is occurring within the filter in the visible spectrum.

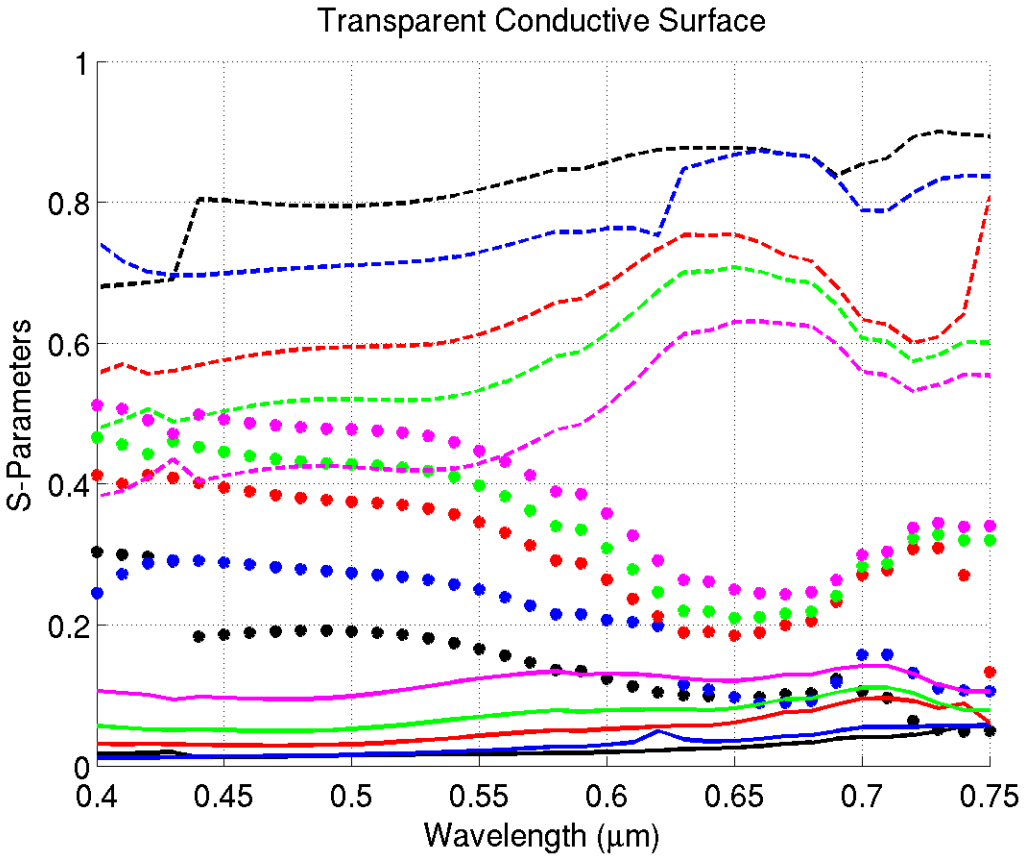
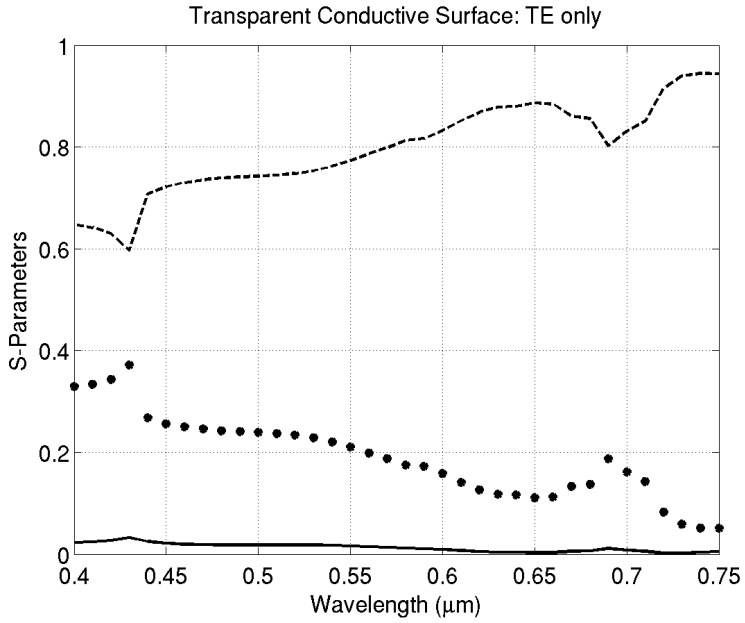
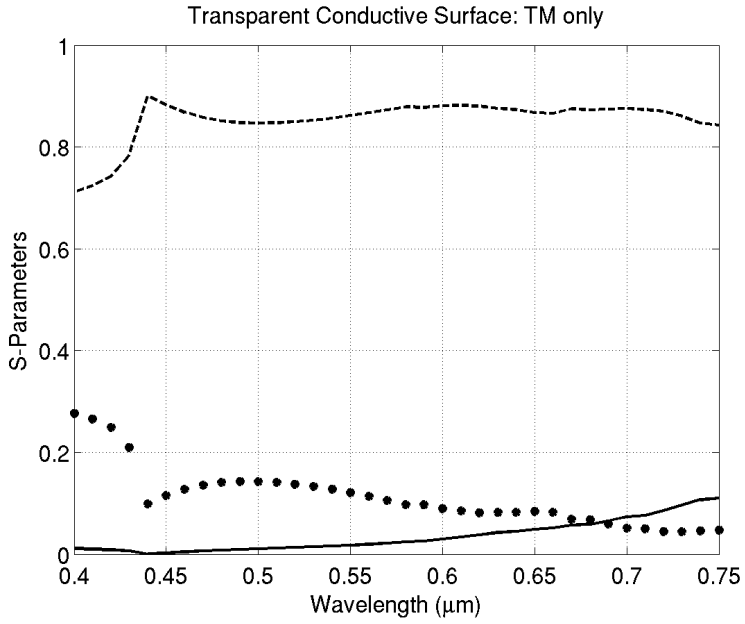


Figure 6-4: Frequency sweeps for unpolarized light from 0 to 65°. Dashed lines show the unit transparency. Solid lines show the reflectivity. Dots show the absorbed energy. Average transparency drops to 51 % in the visible spectrum at an angle of incidence.



(A)



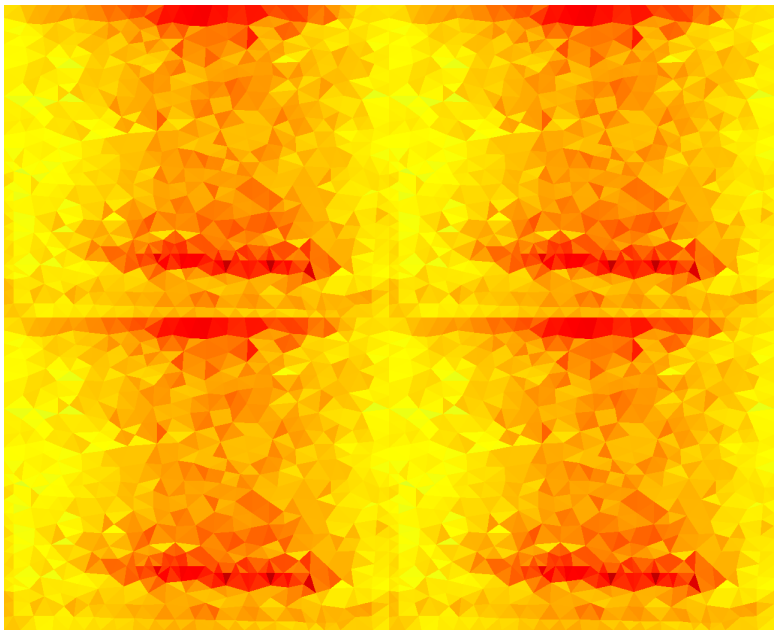
(B)

Figure 6-5: Frequency Sweeps at normal incidence for TE (A) and TM (B) polarizations only.

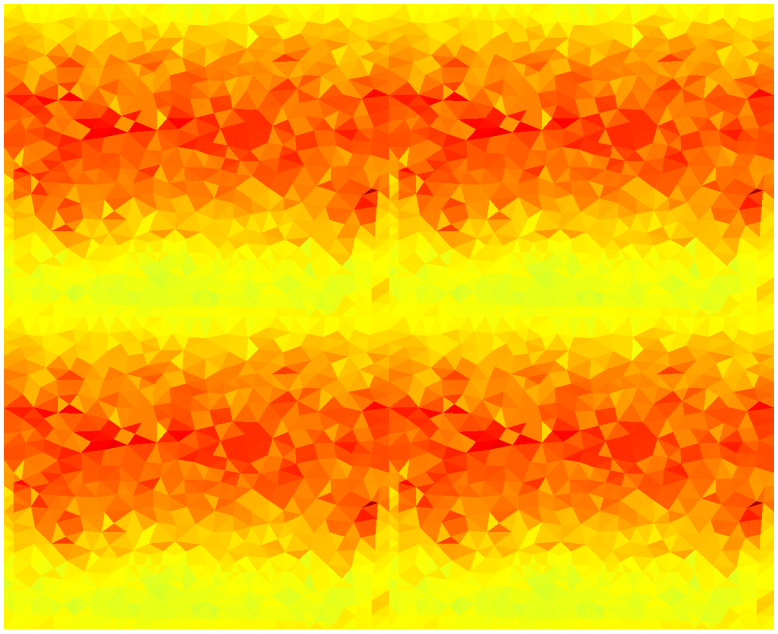
It can be seen that at higher frequencies, towards the ultra-violet, the grating like structure that exists throughout the unit cell starts to take hold and push higher absorption in the TE polarization. In the TM polarization, throughout the band, the frequency response is relatively high showing a response that is nearly agnostic of the metallic structure altogether. This causes absorption to rise in this polarization. At lower frequencies, towards 700 nm, the adjusted wire begins to exhibit a similar grating-like response in the TM polarization. Since this portion of the wire is smaller and does not extend across the unit cell, the TE response exceeds 90 % transmission but the TM response only begins to show an increase in reflection. Additionally, at about 430 nm, a resonance occurs in both polarizations due to the size of the wavelength approaching that of the unit cell.

The use of a finite-element boundary-integral tool for simulation provides us with an additional view at the physical nature of this structure. By examining the finite-element field strengths on the edges of the mesh, one can get a picture of the electromagnetic wave as it exists within the structure. Figure 6-6 and Figure 6-7 show two-dimensional cuts of the finite element field strength immediately above the metallic surface at 430 and 600 nm.

It can be seen in these images that at higher frequencies, towards violet, the waves are nicely confined within the metallic portion of the FSS. In TE, the wave is occluded by the nanowire resulting in some absorption, seen in the darker red color, which represents a stronger field. Meanwhile, at lower frequencies, the wavelength is larger than the unit cell and the interaction is more complex. It appears that the metallic nanowire is focusing the beam into the dielectric component as seen by the dark red field response along the edge of the wire as well as the more concentrated fields in the dielectric.

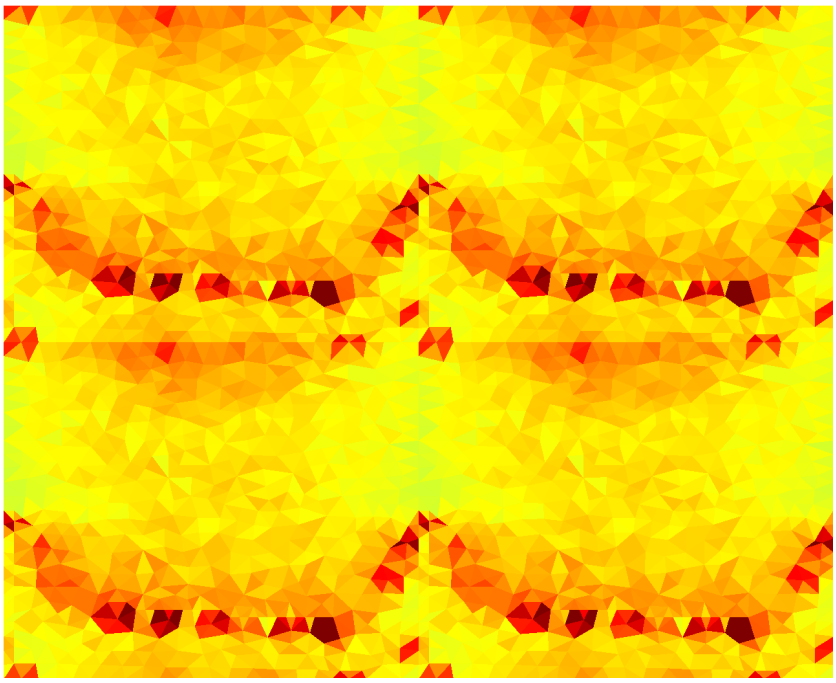


(A)



(B)

Figure 6-6: The finite element field strength at 460 nm. (A) shows the TE fields. It should be noted that there is little interaction between the fields and the metallic screen. (B) shows the TM fields. In this arrangement, the fields fit neatly between the screens yielding little perturbation.



(A)

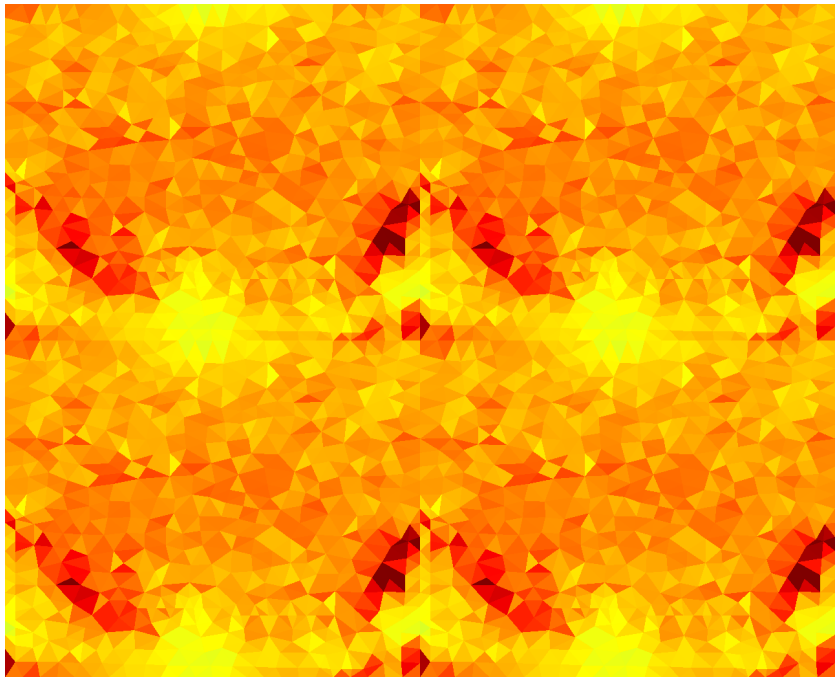


Figure 6-7: The finite element field strength at 600 nm. (A) shows the TE fields. At this higher wavelength, the fields begin to interact with the enlarged sections of the screen resulting in a slight increase in scattering. (B) shows the TM fields. In this arrangement, the fields are primarily concentrated within between the periodic screens but the interaction begins to become larger resulting in increased scattering.

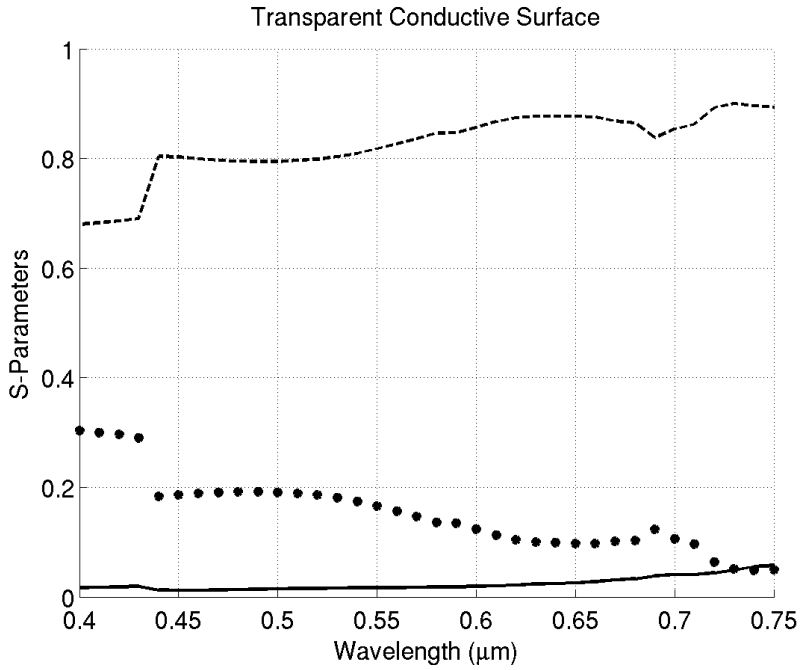


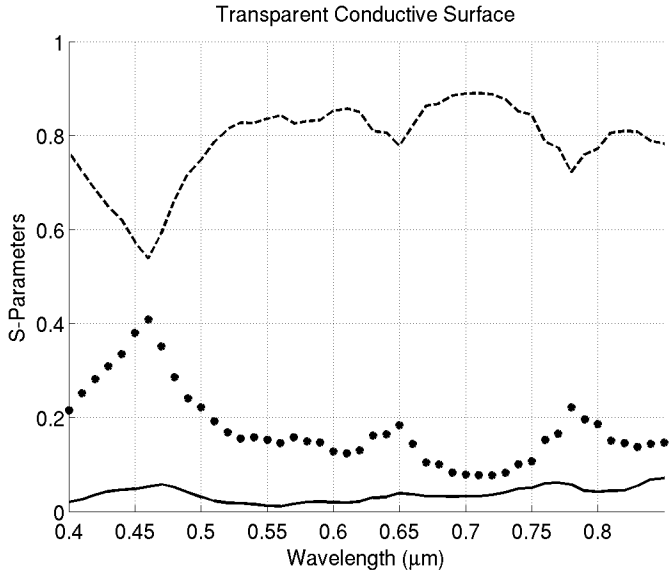
Figure 6-8: A plot of the S-parameters for the FSS using a polyimide substrate.

The TM response is much simpler. Towards the violet, a sinusoidal beam response can be seen across the unit cell. This is similar to what one would expect to see in a wave interaction with a pure dielectric made of one material. At the lower frequency of $600\ nm$, because the wave is larger than the unit cell, interaction with the metallic nanowire is unavoidable. This leads to some energy being reflected.

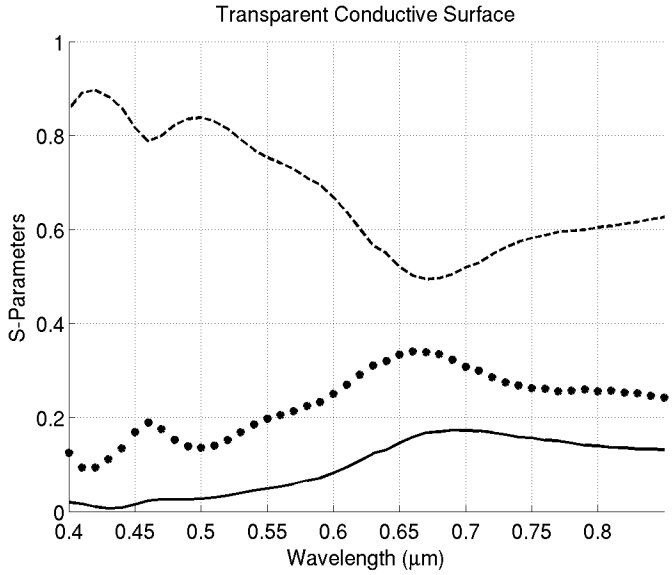
As can be seen in Figure 6-8, this design provides a similar response for different dielectric substrates as well. In addition to the Corning glass substrate, simulations were also performed using a polyimide substrate. Here, the mean transmissivity in the frequency band is 82 %.

Similarly, frequency sweeps were performed using nanowires made of silver and aluminum. Figure 6-9 shows the results of these two side-by-side. From this, it can be seen that the properties of the design are largely due to the relationship between the material parameters of

copper and the physical arrangement of the wire itself. While using a silver wire maintains a high degree of transmission, it adds dispersion to the response, behaving especially poorly at higher frequencies. Meanwhile, aluminum exhibits a poor response throughout most of the band with a high transmission at high frequencies.



(A)



(B)

Figure 6-9: Frequency sweeps at normal incidence for silver (A) and aluminum (B) nanowires.

Conclusion

A FSS based nanowire design has been described that exhibits a strong, low dispersion transparent response while maintaining a consistent path through which current can flow. This design behaves well with widely different dielectric substrates but lacks consistency with other metallic wires. However, due to the transparency that exists in FSS designs, which use a different metal, it should be possible to use a similar optimization process, taking into account the proper material parameters, to achieve a similar response using these other metals as well.

Due to a lack of proper facilities, testing of this design outside of commonly used computer aided design techniques was unable to be performed. However, because this design was optimized with current nanofabrication techniques in mind and it should be possible with the proper facilities to produce a manufactured sample for comparison.

Chapter 7

Enhanced Electro-Absorption Modulation Using an FSS

Introduction

As the demand for increased communication and transmission of data has grown in recent years, there has been a move to increase the use of optical bands which can contain bandwidths several gigahertz wide. The conventional fibre-optics band around $1.55\ \mu m$ contains a bandwidth of $3.7\ GHz$, for example. For these bands to be accessible, two key features are necessary: the ability to encode an electrical signal into an optical one and the ease of integration into a silicon circuit.

Since the discovery of the electro-optical effect in silicon [63], semiconductor electro-optical devices have become attractive. Direct integration into silicon would allow for fast communication within a device or allow for direct integration of a computer with a larger communications network [64]. Interconnects in a computer are increasingly becoming a significant contributor to the total energy consumption of components. SiGe electro-absorption modulators are an attractive alternative to traditional metallic interconnects due to their low power consumption. Energy consumption models for SiGe modulators have calculated a typical consumption of tens of femtojoules or less per bit [65].

In recent years, semiconductor based optical modulation has begun to mature to a point where semiconductor based photonics has become a reality. Using strained silicon or strained silicon-germanium (SiGe) stacked quantum wells [66], devices which demonstrate an electrically-influenced change in the refractive index or absorption coefficient have been developed.

This chapter will demonstrate simulations of devices that have been built around the known parameters of a SiGe stacked quantum well with an embedded metallic frequency selective surface that amplifies the differences in optical parameters of SiGe quantum wells boosting the absorption allowing the device to be smaller.

Properties of a SiGe Absorption Modulator

Silicon-Germanium Physical Properties

The use of stacked multiple quantum wells consisting of Si and Ge induces the quantum-confined Stark effect which characterized by electric-field dependent optical absorption [66]. These stacked SiGe layers typically have thicknesses that are only a fraction of the size of the wavelength. For this reason, it is appropriate to simplify the overall index of refraction to a weighted average of the two materials. In these simulations, the refractive index was chosen based on the properties of a theoretical device consisting of $Si_{0.1}Ge_{0.9}$ which has a weighted index of $n = 3.941$. With a voltage swing typical voltage swing between $V = 0\ V$ and $V = 5\ V$, typical values for the absorption coefficient range from on the order of $\alpha = 1000\ cm^{-1}$ to $8000\ cm^{-1}$ in the original fibre-optics band centered around 1.3 μm and from a reported $1000\ cm^{-1}$ to $5000\ cm^{-1}$ in the conventional band for a device held at high temperatures which one might expect on an active CPU [67].

The Electro-Absorption Effect

The electro-absorption effect occurs in a SiGe stacked quantum well due to external electric fields manipulating the band-gaps of the wells. Typically, these devices consist of

multiple layers of different, doped semiconductor materials stacked such that the electrons or holes may only occupy discrete energy states within each layer. By inducing an electric field on such a device, a shift in the band-gap energy is induced with a fast response rate with rates as high as $3.5\ GHz$ having been reported [68] This shift in band-gap energy can be engineered such that the absorption bands shift to the bands used in fibre-optic communication making them useful for photonic devices.

Germanium features an indirect band gap at $0.67\ eV$ and a small direct band gap at $0.8\ eV$ [65], which corresponds with the conventional communications band at $1.55\ \mu m$. Silicon, on the other hand, features an indirect band gap of $4.0\ eV$. When Si and Ge combine in a quantum well, strain induces a shift in the bandgap energy. By controlling the ratio between Si and Ge, a degree of control over the band gap is created which has an effect on the optical absorption of the material. Typically, individual quantum wells have a thickness that is near $20\ nm$.

Design and Simulation

For an electro-absorption modulator to be useful for communication, it must be able to show a difference in absorption between two applied voltages of at least $3\ dB$ in a reasonable length. Previous devices have shown a $3\ dB$ drop at the $1.55\ \mu m$ band after approximately $60\ \mu m$ [65]. An exceptional device incorporated a coupled ring-resonator reducing the device length to $12\ \mu m$ [66]. For this design, a large difference between the absorption of the system when an applied voltage increases the absorption as compared to one in which no FSS is included is desired. Optical properties of the SiGe multiple quantum wells were determined using data from Liu [65].

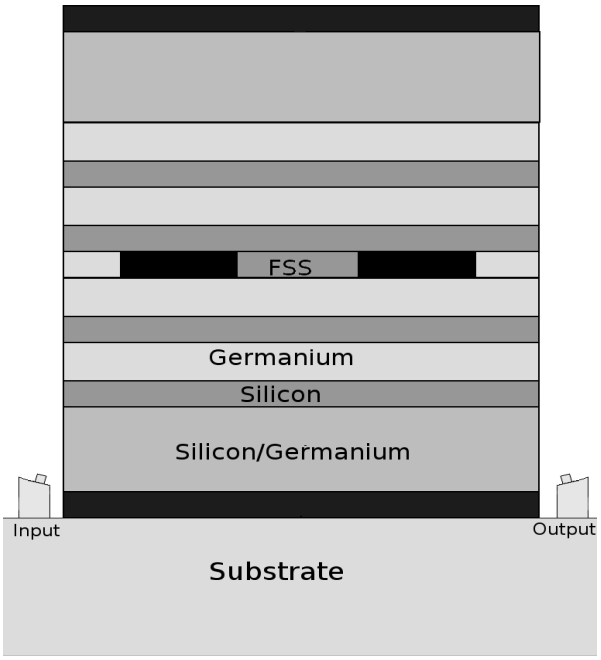


Figure 7-1: A simple schematic of the typical design of a simple silicon germanium electro-absorption modulator.

The Covariance Matrix Adaption Evolution Strategy, which attempts to approximate the covariance matrix of a random sample in an attempt to push the mean to a fitness minimum [29, 30], was integrated with a periodic finite element boundary integral solver to optimize designs centered at 1.3 μm and 1.55 μm. The fitness function used was the difference function in.

$$\boldsymbol{fit = 1 - A_{high} - A_{low}} \qquad (7\text{-}1)$$

The structure of the optimized device is similar to that of Figure 7-1 featuring an FSS embedded in SiGe quantum wells sandwiched between inactive SiGe buffers and electrodes. An optical source is supplies a signal at an angle and the light is assumed to reflect perfectly off the electrodes until it exits the device.

Figure 7-2: The unit cells arranged in 2x2 grids for the optimized frequency selective surfaces. The top design targets 1.3 *µm and the bottom targets* 1.5 *µm.*

The optimized designs can be seen in Figure 7-2. These FSS unit cells feature loops which are on the order of the target wavelength within the SiGe quantum wells in which the

surface is embedded. For this reason, a current is induced on the edges of the FSS. When the SiGe has stronger absorption, the current is stronger and the energy is confined to the surface. When the absorption is set low, the screen acts as a reflective surface and little energy is lost as current.

Frequency sweeps of the designs are shown in Figure 7-3. Using these results, the length required to achieve a 3 dB or 10 dB difference in absorbed optical power can be calculated using

$$\boldsymbol{L = \frac{\log(A_{target})}{\log(1 - A_{diff})} L_0} \qquad (7\text{-}2)$$

where A_{target} is the targeted absorption difference, A_{diff} is the difference in absorption between the two states and L_0 is the distance traveled per pass. From this, it is found that the length required to achieve a 3 dB difference in transmission is 1.03 μm for the FSS targeting the 1.3 μm band and 2.5 μm for the FSS targeting the 1.55 μm band.

Conclusion

Using the optical parameters of previously published SiGe devices, a simple FSS-based design that can be embedded between the quantum wells has been demonstrated. Using this approach, the computer models show an enhancement of the absorption properties in the electro-absorption modulator. This allows the device length to be reduced making it more attractive for use in computer applications where there is a desire to move away from traditional metallic wire-based interconnects to an optical communications system.

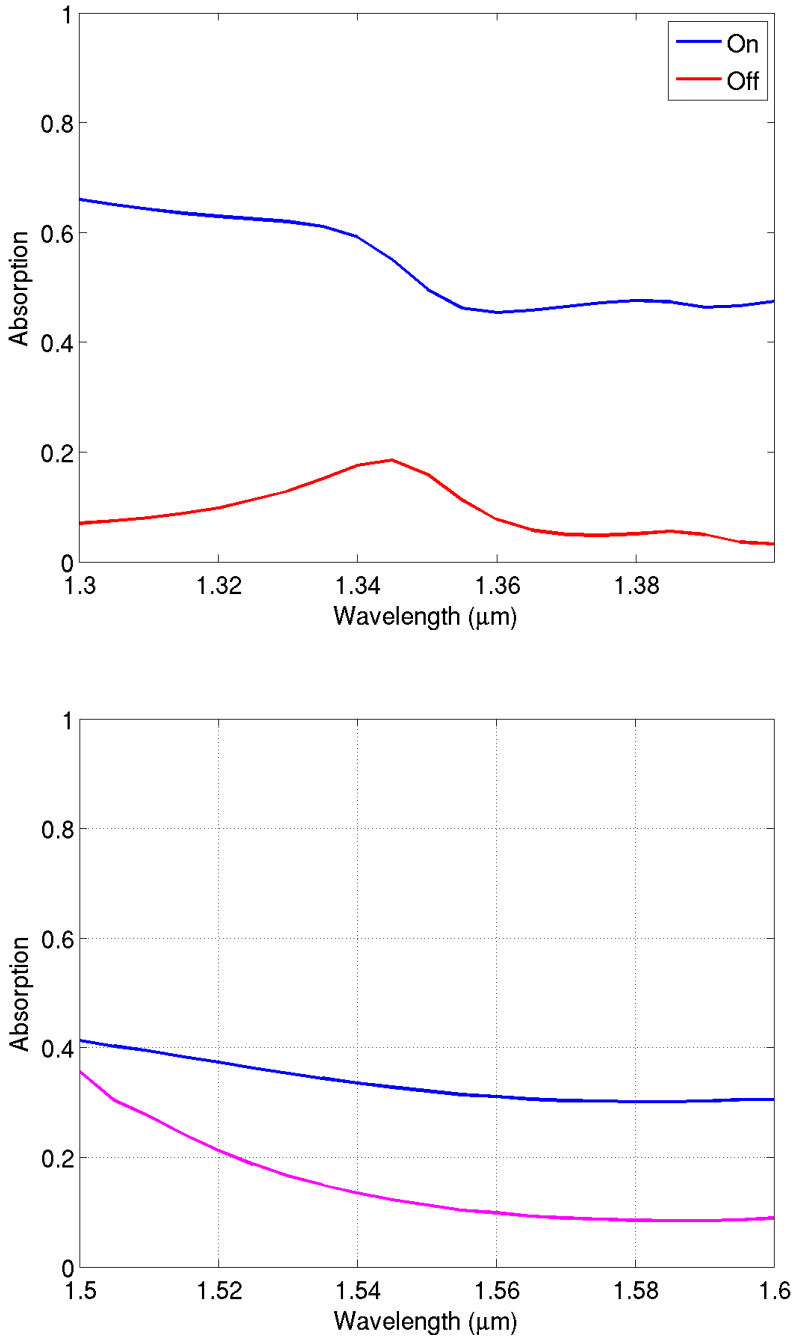


Figure 7-3: Plots of simulated absorption versus wavelengths for the designs shown in Figure 7-2.

Chapter 8 Conclusions and Future Work

Conclusions

The research in this dissertation has been divided into two primary areas of focus. The first section discussed an implementation of the finite element method using a boundary integral method to truncate the top and bottom of the simulation domain to reduce the amount of unknowns and parameters to store in memory.

However, the calculation of the boundary integral, particularly with the use of triangular basis functions requires several repeated, linearly independent numerical integrations, which incorporate the contributions of many elements near and far from each element. Since this sort of problem is well suited for a GPGPU system, the boundary integral code was implemented in the GPU to provide a strong increase in computational speed for these systems balancing the tradeoffs between a boundary integral and an approximate boundary such as the perfectly matched layer.

Using this system, a covariance matrix adaptation evolutionary strategy was employed in conjunction with a Bézier surface based meshing algorithm to produce several proposed designs in the infrared and optical spectra. These designs provided novel approaches to existing electromagnetics problems.

These include problems such as manipulating the polarization of the scattered light to induce a circular polarization, confining optical energy into a doped semiconductor substrate, transparent films whose periodicity is taken advantage of to maintain a constant path for current flow, and enhanced optical absorption modulation which will allow the physical footprint of such systems to be reduced allowing them to be more easily integrated into computer networks.

Future Work

In this dissertation, an accelerated numerical electromagnetic solver and several designs has been discussed. These concepts lay the groundwork for future topics that I would be pleased to see further examination and development. In particular, I would like to see physical realizations of the designs that have been proposed such that they might see further development and possible commercial or research applications. Additionally, a few aspects of the solver could be enhanced. One area, which was not discussed in this work up to this point, was the inclusion of support for lumped elements, such as in Figure 8-1, and analysis thereof.

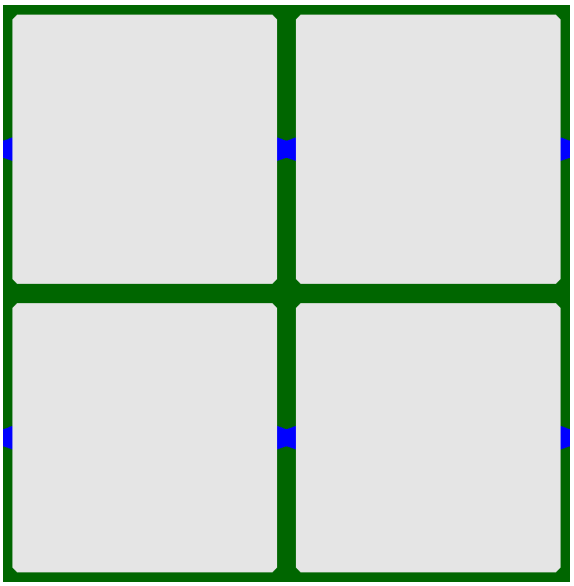


Figure 8-1: A lumped element mesh. The silver portion is a metallic patch and the blue portion between the patches is a lumped resistive element.

Rudimentary work has been done to include support for lumped elements in the triangular finite element boundary integral code. However, since most of the work in this dissertation has been in the optical spectrum, it proved difficult to find designs that are both realizable and an improvement on standard FSS techniques. Recent advances in tunable optical nano-antennas may be easier to simulate using a lumped element approach rather than modeling

the load that may be more easily analyzed using such a model [68]. The current implementation will only work within the finite element region and only supports vertical and horizontal connections to metallic elements.

Based on the presented work, the following is a brief summary of many areas that I believe may be considered for future research:

- Implementation of the generalized minimum residual method with a prismatic basis, hybrid finite-element boundary integral method
- Further study into optical lumped elements as an approximation for nano-scale loads.
- Development of physical optical Bézier surface based Frequency Selective Surfaces using modern etching techniques.
- Study into the physical causes of coupling between doped semiconductors and frequency selective surfaces.

References

1. Y. Avitzour, Y. A. Urzhumov, and G. Shvets, "Wide-angle infrared absorber based on a negative-index plasmonic metamaterial," Phys. Rev. B, vol. 79, pp. 045131/1-5, January 2009.
2. J. H. Jiang, S. Yun, F. Toor, D. H. Werner, and T. S. Mayer, "Conformal dual-band near-perfectly absorbing mid-infrared metamaterial coating," ACS Nano, vol. 5, no. 6, pp. 4641-4647, April 2011.
3. V.G. Veselago, "The electrodynamics of substances with simultaneously negative values of ε and μ," *Sov. Phys. USPEKHI*, Vol. 10, No. 4, 1968, pp. 509-514.
4. J.B. Pendry, "Negative refraction makes a perfect lens," *Phys. Rev. Lett*., Vol. 85, No. 18, pp. 3966-3969, 2000.
5. R. P. Drupp, J. A. Bossard, Y-H. Ye, D. H. Werner, and T. S. Mayer, "Dual-band infrared metallodielectric photonic crystals," *Applied Physics Letters*, Vol. 85, pp. 1835-1837, Sept. 2004.
6. D. J. Kern and D. H. Werner, "Magnetic loading of EBG AMC ground planes and ultrathin absorbers for improved bandwidth performance and reduced size," Microw. Opt. Technol. Lett., vol. 48, no. 12, pp. 2468-2471, December 2006
7. J. L. Volakis, A. Chatterjee, and L. C. Kempel, *Finite Element Method for Electromagnetics*, New York: Wiley-IEEE Press, 1998.
8. R. L. Courant, "Variational methods for the solution of problems of equilibrium and vibration," *Bulletin of the American Mathematical Society*, 49, 1943, pp. 1-23.
9. J. H. Argyris, H. Balmer, J. St. Doltsinis, P.C. Dunne, M. Haase, M. Kleiber, G.A. Malejannakis, H.-P. Mlejnek, M. Muller, and D.W. Scharpf, "Finite element method – the natural approach," *Computer Methods in Applied Mechanics and Engineering*, vol. 17-18, No. 1, pp. 1-106, Jan. 1979.
10. E. Hinton, and J. S. Campbell, "Local and global smoothing of discontinuous finite element functions using a least squares method," *International Journal for Numerical Methods in Engineering*, vol. 8, no. 3, pp. 461-480, Mar. 1974.
11. J.-Y. Wu, and R. Lee, "The advantages of triangular and tetrahedral edge elements for electromagnetic modeling with the finite element method," *IEEE Transactions on Antennas and Propagation*, Vol. 45, No. 9, pp. 1431-1437, Sep. 1997.
12. B. Engquist, and A. Majda, "Absorbing boundary conditions for the numerical simulation of waves," *Mathematics of Computation*, Vol. 31, No. 139, pp. 629-651, Jul. 1977.
13. A. Bayliss, M. Gunzburger, and E. Turkel, "Boundary conditions for the numerical solution of elliptic equations in exterior regions," *SIAM Journal of Applied Math*, Vol. 42, pp. 430-451, 1982.
14. J.-P. Berenger, "A perfectly matched layer for the absorption of electromagnetic waves," *Journal of Computational Physics*, Vol. 114, pp. 185-200, Oct. 1994.
15. C. A. Brebbia, *Boundary element method for engineers*, London: Pentech, 1978.
16. H. Ling, "RCS of waveguide cavities: a hybrid boundary-integral/modal approach," *IEEE Transactions on Antennas and Propagation*, Vol. 38, No. 9, Sep. 1990.

17. T. F. Eibert, J. L. Volakis, D. R. Wilton, and D. R. Jackson, “Hybrid FE/BI modeling of 3-d periodic doubly periodic strucures utilizing triangular prismatic elements and an MPIE formulation accelerated by the Ewald transformation”, IEEE Trans. Antennas Propag., vol. 47, no. 5, pp. 843-850, May 1999.
18. R. E. Jorgenson, and R. Mittra, “Efficient calculation of the free-space periodic Green’s function,” *IEEE Transactions on Antennas and Propagation*, Vol. 38, No. 5, pp. 633-642, May 1990.
19. J. B. Kruskal, “On the shortest spanning subtree of a graph and the traveling salesman problem,” *Proceedings of the American Mathematical Society*, Vol. 7, No. 2, pp. 48-50, 1956.
20. M. L. Fredman, and R. E. Tarjan, “Fibonnaci heaps and their uses in improved network optimization algorithms,” *25th Annual Symposium on Foundations of Computer Science*, pp. 338-346, 24-26 Oct. 1984.
21. X. Wang, and D. H. Werner, "Fast Analysis of 3-D Doubly Periodic Structures with Complex Geometry and Anisotropic Materials using the Adaptive Integral Method," *2010 IEEE Antennas and Propagation Society International Symposium (APSURSI)*, pp. 1-4, July 11-17 2010.
22. B. Delaunay, “Sur la *sphère vide". Bulletin de l’Académie des Sciences de l'URSS, Classe des sciences mathématiques et naturelles, Vol. 6, pp. 793–800, 1934.*
23. The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.7 edition, 2015.
24. R. H. Bartels, *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*, 1987: Morgan Kaufmann.
25. H. Edelsbrunner, D. G. Kirkpatrick, and R. Seidel, “On the shape of a set of points in the plane,” *IEEE Transactions on Information theory*, Vol. 29, No. 4, pp. 551-559, Jul. 1983.
26. H. Edelsbrunner and E. P. Mücke. “Three-dimensional alpha shapes,” ACM Trans. Graph*., 13(1): pp43–72, Jan. 1994.*
27. R. L. Haupt and D. H. Werner, *Genetic Algorithms in Electromagnetics*, New York: Wiley-IEEE Press, 2007.
28. P. L. Werner, R. Mittra, and D. H. Werner, "Extraction of Equivalent Circuits for Microstrip Components and Discontinuities Using the Genetic Algorithm," *IEEE Microwave and Guided Wave Letters*, Vol. 8, No. 10, pp. 333-335, 1998.
29. N. Hansen, S.D. Muller and P. Koumoutsakos, “Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES),” *Evolutionary Computing*, Vol. 11, No. 1, pp. 1-18, Mar. 2003.
30. M. D. Gregory, Z. Bayraktar, and D. H. Werner, “Fast optimization of electromagnetic design problems using the covariance matrix adaptation evolution strategy,” IEEE Trans. Antennas Propag., vol. 59, no. 4, pp. 1275-1285, April 2011**.**
31. A. Auger and N. Hansen, “Performance evaluation of an advanced local search algorithm,” *Proceedings of the IEEE Congress on Evolutionary Computation*, Vol. 2, pp. 1777-1784, 2-5 Sep. 2005.
32. A. Auger and N. Hansen, “A restart evolutionary strategy with increasing population size,” *Proceedings of the IEEE Congress on Evolutionary Computation*, Vol. 2, pp. 1769-1776, 2-5 Sep. 2005.
33. The Khronos Group, 2015 Nov. 11, *The OpenCL Specification* (version 2.1), [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf
34. G. P. M. Poppe, and C. M. J. Wijers, “More efficient calculation of the complex error function,” *ACM Transactions on Mathematical Software*, Vol. 16, No. 1, pp. 38-46, Mar. 1990.

35. Y. Saad, *Iterative Methods for Sparse Linear Systems*, Philadelphia: SIAM, 2003.
36. Y. Saad and M. H. Schultz, "GMRES: a generalized minimum residual algorithm for solving nonsymmetric linear systems", *SIAM Journal of Scientific Computing*, Vol. 7, No. 3, Jul. 1986.
37. Y. Avitzour, Y. A. Urzhumov, and G. Shvets, "Wide-angle infrared absorber based on a negative-index plasmonic metamaterial," *Phys. Rev. B*, vol. 79, pp. 045131/1-5, January 2009.
38. J. H. Jiang, S. Yun, F. Toor, D. H. Werner, and T. S. Mayer, "Conformal dual-band near-perfectly absorbing mid-infrared metamaterial coating," *ACS Nano*, vol. 5, no. 6, pp. 4641-4647, April 2011.
39. D. J. Kern, and D. H. Werner, "Magnetic loading of EBG AMC ground planes and ultrathin absorbers for improved bandwidth performance and reduced size," *Microw. Opt. Technol. Lett.*, vol. 48, no. 12, pp. 2468-2471, December 2006.
40. A. D. Rakić, A. B. Djurišić, J. M. Elazar, and M. L. Majewski, "Optical properties of metallic films for vertical-cavity optoelectronic devices," Appl. Opt., vol. 37, no. 22, pp. 5271-5283, August 1998.
41. Martinelli, M., Martelli, P., and Pietralunga, S. M., "Polarization Stabilization in Optical Communications Systems," *Journal of Lightwave Technology*, Vol. 24, No. 11, 4172-4183, Nov. 2006.
42. G. Nordin, J. Meier, P. Deguzman, and M. Jones, "Micropolarizer array for infrared imaging polarimetry," *J. Opt. Soc. Am. A*, Vol. 16, pp. 1168-1174 (1999).
43. J. Tyo, D. Goldstein, D. Chenault, and J. Shaw, "Review of passive imaging polarimetry for remote sensing applications," *Applied Optics*, Vol. 45, pp. 5453-5469 (2006).
44. J. Arce-Diego, R. López-Ruisánchez, J. López-Higuera, and M. Muriel, "Fiber Bragg grating as an optical filter tuned by a magnetic field," *Optics Letters*, Vol. 22, pp. 603-605 (1997).
45. P. Deguzman and G. Nordin, "Stacked Subwavelength Gratings as Circular Polarization Filters," *Applied Optics*, Vol. 40, pp. 5731-5737 (2001).
46. Q. Hong, T. Wu, X. Zhu, R. Lu, and S. Wu, "Designs of wide-view and broadband circular polarizers," *Optics Express*, Vol. 13, pp. 8318-8331 (2005).
47. G. Wysocki, R. Lewicki, R. F. Curl, F. K. Tittel, L. Diehl, F. Capasso, M. Troccoli, G. Hofler, D. Bour, S. Corzine, R. Maulini, M. Giovannini, and J. Faist, "Widely tunable mode-hop free external cavity quantum cascade lasers for high resolution spectroscopy and chemical sensing," *Appl. Phys. B, Vol.* 92, No. 3, pp. 305–311, 2008.
48. A. B. Kostinski, C. R. Givens, J. M. Kwiatkowski, "Constraints on Mueller Matrices of Polarization Optics," *Applied Optics*, Vol. 32, No. 9, March 1993.
49. C. Laperle, B. Villeneuve, Z. Zhang, D. McGhan, H. Sun, and M. O'Sullivan, "Wavelength Division Multiplexing (WDM) and Polarization Mode Dispersion (PMD) Performance of a Coherent 40Gbit/s Dual-Polarization Quadrature Phase Shift Keying (DP-QPSK) Transceiver," in *Optical Fiber Communication Conference and Exposition and The National Fiber Optic Engineers Conference*, March 2007.
50. US Geological Survey, 2014 Nov., *U.S. Mineral Commodity Summaries*, [Online]. Available: http://minerals.usgs.gov/minerals/pubs/commodity/indium/mcs-2015-indiu.pdf
51. T. Minami, H. Nanto, and S. Takata, "Highly Conductive and Transparent Aluminum Doped Zinc Oxide Thin Films Prepared by RF Magnetron Sputtering," *Japanese Journal of Applied Physics*, Vol. 23, No. 5, 1984.
52. E. Fortunato, L. Raniero, L. Silva, A. Gonçalves, A. Pimentel, P. Barquinha, H. Águas, L. Pereira, G. Gonçalves, I. Ferreira, E. Elangovan, R. Martins, "Highly stable transparent and

conducting gallium-doped zinc oxide thin films for photovoltaic applications", *Solar Energy Materials and Solar Cells*, Vol. 92, No. 12, pp. 1605-1610, December 2008.
53. M. Oh, W.-Y. Jin, H. J. Jun, M. S. Jeong, J.-W. Kang, and H. Kim, "*Silver nanowire transparent conductive electrodes for high-efficiency III-nitride light-emitting diodes,"Scientific Reports*, Vol. 5, pp. 13483-, Sep. 2015.
54. E. M. Doherty, S. De, P. E. Lyons, A. Shmeliov, et al, "The spatial uniformity and electromechanical stability of transparent, conductive films of single walled nanotubes," *Carbon*, Vol. 47, pp. 2466-2473, 2009.
55. A. R. Madraria, A. Kumar, and C. Zhou, "Large scale, highly conductive and patterned transparent films of silver nanowires on arbitrary substrates and their application in touch screens," *Nanotechnology*, Vol. 22 (24), 245201, 2011.
56. Huang, X. et al., "Sub 50-nm FinFET: PMOS" *International Electron Devices Meeting Technical Digest*, pp. 67-70, December 5–8, 1999.
57. L. Pasini, et al., "High performance low temperature activated devices and optimization guidelines for 3D VLSI integration of FD, TriGate, FinFET on insulator," in *2015 Symposium on VLSI Technology (VLSI Technology),* pp. T50-T51, 16-18 June 2015.
58. J. A. Ashbach, P. L. Werner, D. H. Werner, and F. Namin, "Single material alternative to a multilayer optical window," Proceedings of the *2010 IEEE International Symposium on Antennas and Propagation and USNC/URSI National Radio Science Meeting*, Toronto, Canada, July 11-17, 2010.
59. M. Switkes, and M. Rothschild, "Immersion lithography at 157 nm," *Journal of Vacuum Science and Technology B*, Vol. 19, pp. 2353-, 2001.
60. Interconnect Roadmap 2011
61. Corning 0211 Microsheet.
62. H. Kim et al., "Electrical, optical, and structural properties of indium‐tin‐oxide thin films for organic light-emitting devices," *Journal of Applied Physics*, Vol. 86, No. 11, pp 6451-6461, Dec. 1999.
63. R. A. Soref and B. R. Bennett, "Electrooptical effects in silicon," , IEEE Journal of Quantum Electronics, Vol. 23, No. 1, pp. 123-129, Jan. 1987.
64. D. A. B. Miller, "Physical reasons for optical interconnection", *International Journal of Optoelectronics, Special Issue on Smart Pixels*, Vol. 11, No. 3, pp. 155-168, May 1997.
65. R. K. Schaevitz, E. H. Edwards, et. al, "Simple electroabsorption calculator for designing 1310 nm and 1550 nm modulators using germanium quantum wells" *IEEE Journal of Quantum Electronics*, Vol. 48, No. 2, pp. 187–197, Feb 2012.
66. Q. Xu, B. Schmidt, et al., "Micrometre-scale silicon electro-optic modulator," *Nature*, Vol. 425, No. 7040, pp. 325-327, Mar 2005.
67. P. Chaisakul, D. Marris-Morini, M. Rouifed, G. Isella, D. Chrastina, J. Frigerio, X. Le Roux, S. Edmond, J. Coudevylle, and L. Vivien, "23 GHz Ge/SiGe multiple quantum well electro-absorption modulator," *Optics Express*, Vol. 20, No. 3, pp. 3219-3224, Jan 2012.
68. A. H. Panaretos, Y. A. Yuwen, D. H. Werner, and T. S. Mayer, "Tuning the optical response of a dimer nanoantenna using plasmonic nanoring loads," *Scientific Reports*, Vol. 5, 9813, May 2015.

Appendices

Appendix A

Mesh Generation Code for the CGAL Library

```
#include <stdlib.h>
#include <fstream>
#include <string.h>
#include <sstream>
#include <vector>
#include <iostream>
#include <cmath>
#include <map>
#include <time.h>
#include <png.h>
#include "make_mesh.h"

// OpenGL
void handleKeypress(unsigned char key, int x, int y) {
    switch (key) {
            // Escape Key
        case 27:
            exit(0);
        case 'm': case 'M':
            showMesh = !showMesh;
            break;
        case '+':
            if (displayScreen+1 < nlayers) {
                displayScreen++;
            }
            break;
        case '-':
            if (displayScreen > 0) {
                displayScreen--;
            }
            break;
        default:
            break;
    }
}

void mouseMove(int x, int y)
{
    if (xOrigin >= 0) {
        // update deltaAngle
        deltaAngle = (x - xOrigin) * 0.005f;

        // update camera's direction
        lx = sin(glAngle + deltaAngle);
        lz = -cos(glAngle + deltaAngle);
    }
}

void handleMouse(int button, int state, int x, int y) {
    // only start motion if the left button is pressed
    if (button == GLUT_LEFT_BUTTON) {
```

```
        int modifiers = glutGetModifiers();
        if ( modifiers != GLUT_ACTIVE_CTRL) {
            if (state == GLUT_UP) {
                glAngle += deltaAngle;
                xOrigin = -1;
            } else {
                xOrigin = x;
            }
        } else {
            if (state == GLUT_UP) {
                zPos += deltaZ;
            } else {
                xOrigin = x;
            }
        }
    }
}

void computePos(float deltaMove) {
    x += deltaMove * lx * 0.1f;
    z += deltaMove * lz * 0.1f;
}

void changeSize(int w, int h) {

    // Prevent a divide by zero, when window is too short
    // (you can't make a window of zero width).
    if (h == 0)
        h = 1;

    float ratio =  w * 1.0 / h;

    // Use the Projection Matrix
    glMatrixMode(GL_PROJECTION);

    // Reset Matrix
    glLoadIdentity();

    // Set the viewport to be the entire window
    glViewport(0, 0, w, h);

    // Set the correct perspective.
    gluPerspective(45.0f, ratio, 0.1f, 100.0f);

    // Get Back to the Modelview
    glMatrixMode(GL_MODELVIEW);
}

void renderBitmapString(
                        float x,
                        float y,
                        void *font,
                        char *string) {
    int i, len;
    char *c;
    len = (int) strlen(string);
    glRasterPos2f(x, y);
    glColor4f(0.0f, 0.0f, 0.0f, 0.0f);
    //  glutBitmapString(GLUT_BITMAP_HELVETICA_18, string);
    //  for (i = 0; i < len; i++)
    for (c=string; *c != '\0'; c++) {
        glutBitmapCharacter(font, *c);
    }
}

void renderScene(void)
{
```

```
//  glutSetWindow(subWindow1);
char number[12];
int i, j;

double x, y, z;

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
if (showMesh)
    glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );
else
    glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
glColor3f(0.0f, 0.0f, 1.0f);

glLoadIdentity();
//  glRotatef(glAngle, 0.0f, 0.0f, 1.0f);

gluLookAt( x, 1.0f, 0.5*z,
          x+lx, 1.0f,  z+lz,
          0.0f, 1.0f,  0.0f);

i = 0;
glBegin(GL_TRIANGLES);
for (i = 0; i < vertices.size() / 4; i++) {
    if (dim == 2)
        glVertex3f(nodes[vertices[i]*dim], nodes[vertices[i]*dim+1], 0.0);
    else if (dim == 3) {
        if (vertices[i] >= 0) {
            for (j = 0; j < 3; j++) {
                x = nodes[vertices[i*4 + j]*dim]/(sizeX);
                y = nodes[vertices[i*4 + j]*dim+1]/(sizeY)+1.0;
                z = nodes[vertices[i*4 + j]*dim+2]/(sizeX);
                if (pecMesh) {
                    if (is_patch[i]) {
                        glColor3f(0.863f, 0.698f, 0.192f);
                    } else {
                        glColor3f(0.1137f, 0.21176f, 0.3333f);
                    }
                }
                else {
                    if (which_dielectric[displayScreen * nFaces + i] == 1) {
                        glColor3f(0.1137f, 0.21176f, 0.3333f);
                    } else {
                        glColor3f(0.863f, 0.698f, 0.192f);
                    }
                    glBindTexture(GL_TEXTURE_2D, textureIDs[1]);
                }
                glVertex3d(x, y, z);
            }
        }
    }
}
glEnd();
// Write text
glColor4f(0.0f,0.0f,0.0f, 1.0f);
sprintf(number, "Layer: %d", displayScreen);
renderBitmapString(-0.95, 0.05, (void *)font, number);
// textBox
glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
glColor4f(0.8f, 0.8f, 0.8f, 0.8f);
glBegin(GL_TRIANGLES);
// triangle1
glVertex3d(-1.0, 0.0, 0.0);
glVertex3d(-0.5, 0.0, 0.0);
glVertex3d(-1.0, 0.2, 0.0);
// triangle2
glVertex3d(-1.0, 0.2, 0.0);
glVertex3d(-0.5, 0.2, 0.0);
```

```
    glVertex3d(-0.5, 0.0, 0.0);
    glEnd();
    glutSwapBuffers();
}

double factorial(int n)
{
    return (n <= 1 ? 1.0 : static_cast<double>(n)*factorial(n-1));
}

bool beztest(double x, double y)
{
    int n = bezsize;
    double c = 0; // result from Bézier calculation
    double Bi, Bj;
    for (int i = 0; i < bezsize; i++) {
        Bi = factorial(n-1) * pow(x, i) * pow(1.0 - x,n-1-i) / (factorial(i) *
factorial(n-1-i));
        for (int j = 0; j < bezsize; j++) {
            Bj = factorial(n-1) * pow(y, j) * pow(1.0 - y,n-1-j) / (factorial(j) *
factorial(n-1-j));
            c += Bi * Bj * Bézier_coeffs[i*n + j];
        }
    }
    if (fabs(c/cmax - bezthresh) < 0.001)
        printf("beztest bez(%f, %f) =  %f, thresh %f\n", x,y,c/cmax,bezthresh);
    return (c/cmax > bezthresh);
}

void read_off_file(std::string filename)
{
    std::ifstream fid;
    fid.open(filename.c_str());
    std::string line;
    std::stringstream ss;

    bool thislayer, was_screen, was_sub;
    bool test[3];
    bool facesDefined = false;
    bool nodesDefined = false;
    bool isOff = false;
    int i, j;
    int patch = 0;
    int screeni, subi;
    int n, attributes, boundary_marker;
    int nVert, vert_boundary_marker;

    int nEdge; // nEdge isn't actually used
    int nPolyVertices; // number of vertices for the ith polygon
    int maxPolyVertices = 4;
    int ithVertices[4];
    int idx;
    double triVertices[6];

    Point p;

    dim = 3; // OFF is defined in 3-Dimensions

    while(getline(fid, line)) {
        //the following line trims white space from the beginning of the string
        line.erase(line.begin(), find_if(line.begin(), line.end(),
std::not1(std::ptr_fun<int, int>(isspace))));
        if (line[0] == '#')
            continue;
        if (line.empty())
            continue;
        if (line.compare("OFF") == 0) {
```

```
            isOff = true;
            // Next line should be numVertices numFaces numEdges
            getline(fid, line);
            std::stringstream(line) >> nVert >> nFaces >> nEdge;
            continue;
        }

        if (!nodesDefined && isOff) {
            nodes.resize(nVert*dim);
            for (i = 0; i < nVert; i++) {
                while (line.empty())
                    getline(fid,line);
                if (dim == 2) {
                    // this doesn't happen
                    std::stringstream(line) >> nodes[i*dim] >> nodes[i*dim + 1];
                } else if (dim == 3) {
                    std::stringstream(line) >> nodes[i*dim] >> nodes[i*dim + 1] >>
nodes[i*dim + 2];
                }
                getline(fid,line);
            }

            nodesDefined = true;
        } else if (!facesDefined && isOff) {
            vertices.resize(nFaces * maxPolyVertices);
            for (i = 0; i < nFaces; i++) {
                while(line.empty())
                    getline(fid, line);

                //    ss << line;
                //    ss >> nPolyVertices;
                // For now, let's just assume that we have triangular meshes
                std::stringstream(line) >> nPolyVertices >>
vertices[i*maxPolyVertices] >> vertices[i*maxPolyVertices+1] >>
vertices[i*maxPolyVertices+2];
                //    std::cout << line << " " << nPolyVertices << "\n";
                //    for (int j = 0; j < nPolyVertices; j++) {
                //      ss >> vertices[i*maxPolyVertices + j];
                //      std::cout << vertices[i*maxPolyVertices + j] << " ";
                //    }
                getline(fid, line);
            }
            facesDefined = true;
        }
    }
    fid.close();
    for (int l = 0; l < nlayers; l++) {
        if (l < ndiel1+nscreens && l >= ndiel1) {
            for (j = 0; j < nFaces; j++) {
                // try to move the point to the upper right quadrant of a square of size
2
                // i.e. upper right quadrant's square is size 1 with bottom left at
origin
                test[0] = false;
                test[1] = false;
                test[2] = false;
                for (int k = 0; k < 3; k++) {
                    double x;
                    double y;
                    x = (nodes[vertices[j*4+k]*dim] + sizeX / 2.0) / sizeX;
                    y = (nodes[vertices[j*4+k]*dim+1] + sizeY / 2.0) / sizeY;
                    Point curPt = Point(nodes[vertices[j*4+k]*dim],
nodes[vertices[j*4+k]*dim+1]);
                    int holeNum = 0;
                    for (std::vector<Polygon_2>::iterator it = pgn.begin();
                         it != pgn.end(); it++) {
                        if (!isHole[holeNum]) { // Do not fill in holes.
```

```
                            if (it->bounded_side(curPt) != CGAL::ON_UNBOUNDED_SIDE)
                                test[k] = true;
                        } else {
                            // be a little more strict on holes to prevent them from
eating
                            // boundary triangles
                            if (it->bounded_side(curPt) == CGAL::ON_BOUNDED_SIDE)
                                test[k] = false;
                        }
                        holeNum++;
                    }
                }
                int count = 0;
                for (int iTest = 0; iTest < 3; iTest++) {
                    if (test[iTest])
                        count++;
                }
                if (count > 2) { //(test[0] == true && test[1] == true && test[2] ==
true) {
                    if (pecMesh && patch < nPatchLayers) {
                        which_dielectric.push_back(1);
                        is_patch.push_back(true);
                    } else {
                        which_dielectric.push_back(2);
                    }
                } else {
                    if (pecMesh && patch < nPatchLayers) {
                        is_patch.push_back(false);
                    }
                    which_dielectric.push_back(1);
                } // if (test
            } // for (j
            patch++;
        } else if (l < ndiel1 + nscreens + ndiel2 || l < ndiel1) { // if (l < nscreens
            std::cout << "l " << l << "\n";
            for (j = 0; j < nFaces; j++) {
                which_dielectric.push_back(1);
            }
        } else { // l >= nscreens+ndiel aka ground layers
            if (pecMesh) {
                for (j = 0; j < nFaces; j++) {
                    which_dielectric.push_back(1);
                    is_patch.push_back(true);
                }
            } else {
                for (j = 0; j < nFaces; j++) {
                    which_dielectric.push_back(3);
                }
            }
        }
    } // for (l
}

void read_poly_file(std::string filename)
{
    std::ifstream fid;
    fid.open(filename.c_str());
    std::string line;
    std::stringstream ss;

    bool nodesDefined = false;
    bool verticesDefined = false;
    bool holesDefined = false;
    int i;
    int n, attributes, boundary_marker;
    int nVert, vert_boundary_marker;
    int nHoles;
```

```
    int idx;

    while(getline(fid, line)) {
        //the following line trims white space from the beginning of the string
        line.erase(line.begin(), find_if(line.begin(), line.end(),
std::not1(std::ptr_fun<int, int>(isspace))));
        if (line[0] == '#')
            continue;
        if (!nodesDefined) {
            std::stringstream(line) >> n >> dim >> attributes >> boundary_marker;
            nodes.resize(n*dim);
            for (i = 0; i < n; i++) {
                getline(fid,line);
                while (line.empty())
                    getline(fid,line);
                if (dim == 2) {
                    std::stringstream(line) >> idx >> nodes[i*dim] >> nodes[i*dim + 1];
                } else if (dim == 2) {
                    std::stringstream(line) >> idx >> nodes[i*dim] >> nodes[i*dim + 1] >>
nodes[i*dim + 2];
                }
            }

            nodesDefined = true;
        } else if (!verticesDefined) {
            while(line.empty())
                getline(fid, line);
            std::stringstream(line) >> nVert >> vert_boundary_marker;
            vertices.resize(nVert * dim);
            for (i = 0; i < nVert; i++) {
                getline(fid, line);
                while(line.empty())
                    getline(fid, line);
                if (dim == 2) {
                    std::stringstream(line) >> idx >> vertices[i*dim] >> vertices[i*dim +
1];
                }
                else if (dim == 3) {
                    std::stringstream(line) >> idx >> vertices[i*dim] >> vertices[i*dim +
1] >> vertices[i*dim + 2];
                }
            }
            verticesDefined = true;
        }
        else if (!holesDefined) {
            while (line.empty())
                getline(fid, line);
            std::stringstream(line) >> nHoles;
            holesDefined = true;
        }
    }
    for (i = 0; i < vertices.size(); i++)
        vertices[i]--;
    fid.close();
}

void write_triangle_OFF_file(CDT &cdt, std::string outfile)
{
    bool binary = false;
    bool noc = false;
    bool verbose = false;

    std::ofstream out (outfile.c_str());
    std::map<const Vertex*, std::size_t, std::less<const Vertex*> > mapping;
    std::size_t vn = 0;
    Vertex_iterator vi;
    for ( vi = cdt.vertices_begin(); vi != cdt.vertices_end(); vi++ ) {
```

```
        CGAL_assertion( ! cdt.is_infinite( vi ) );
        mapping[ &*vi] = vn;
        vn++;
    }
    CGAL_assertion( vn == std::size_t( cdt.number_of_vertices() ) );

    std::size_t fn = 0;
    Face_iterator fi;
    for ( fi = cdt.faces_begin(); fi != cdt.faces_end(); fi++) {
        CGAL_assertion( !cdt.is_infinite( fi ) );
        fn++;
    }
    std::size_t fin = cdt.number_of_faces() - fn;

    File_header_OFF header( binary, noc, false, verbose);
    File_writer_OFF writer( header );
    writer.write_header( out, vn, 3 * fn + fin, fn );

    for (vi = cdt.vertices_begin(); vi != cdt.vertices_end(); vi++) {
        CGAL_assertion( !cdt.is_infinite( vi ) );
        writer.write_vertex( to_double(vi->point().x()),
                             to_double(vi->point().y()),
                             0.0);
    }
    writer.write_facet_header();

    fi = cdt.faces_begin();
    while (fn--) {
        writer.write_facet_begin( 3 );
        CGAL_assertion( mapping.find(&*(fi->vertex(0))) != mapping.end());
        CGAL_assertion( mapping.find(&*(fi->vertex(1))) != mapping.end());
        CGAL_assertion( mapping.find(&*(fi->vertex(2))) != mapping.end());
        writer.write_facet_vertex_index( mapping[ &*(fi->vertex(0))]);
        writer.write_facet_vertex_index( mapping[ &*(fi->vertex(1))]);
        writer.write_facet_vertex_index( mapping[ &*(fi->vertex(2))]);
        writer.write_facet_end();
        fi++;
    }
    CGAL_assertion( fi == cdt.faces_end() );
    writer.write_footer();
}

void init()
{
    bool hasAlpha;
    bool success;
    int height, width;

    glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
    glShadeModel(GL_SMOOTH);

    glEnable(GL_BLEND);
    glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    //   glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);
    GLfloat pos_light[4] = {2.0, 2.0, 2.0, 0.0};
    GLfloat amb_light[4] = {1.0, 1.0, 1.0, 0.0};
    glLightfv(GL_LIGHT0, GL_POSITION, pos_light);
    glLightfv(GL_LIGHT0, GL_AMBIENT, amb_light);

    // Bind image data to textures
    glGenTextures(2, textureIDs);
    // Texture 1
    const char* metalPng = "/home/jason/Textures/metal.png";
```

```
    success = loadPngImage(metalPng, width, height, hasAlpha, &metalTextureImage);
    if (!success) {
        std::cout << "Error loading png file\n";
        return;
    }
    glBindTexture(GL_TEXTURE_2D, textureIDs[0]);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, hasAlpha ? 4 : 3, width, height, 0, hasAlpha ?
GL_RGBA : GL_RGB,
                 GL_UNSIGNED_BYTE, metalTextureImage);
    // Texture 2
    const char* dielPng = "/home/jason/Textures/danube.png";
    success = loadPngImage(dielPng, width, height, hasAlpha, &dieleTextureImage);
    if (!success) {
        std::cout << "Error loading png file\n";
        return;
    }
    glBindTexture(GL_TEXTURE_2D, textureIDs[1]);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, hasAlpha ? 4 : 3, width, height, 0, hasAlpha ?
GL_RGBA : GL_RGB,
                 GL_UNSIGNED_BYTE, dieleTextureImage);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
}

void writePFEBI(int nScreen)
{
    int i, j, nShed;
    int nNodes = nodes.size()/3;
    int nTri;
    int vertexNumber[3];
    double x, y, z;
    double ix1, ix2, ix3, iy1, iy2, iy3, c1, c2;
    double area;
    std::ofstream meshFile ("mesh.txt");
    std::ofstream patchFile;
    if (pecMesh) {
        std::cout << "Generating screen information.\n";
        patchFile.open("screen.txt");
    }
    std::ofstream materialFile ("material.txt");

    // Start mesh information
    meshFile << "0\n";
    meshFile << nNodes << "\n";

    for (i = 0; i < nNodes; i++) {
        x = nodes[i*dim];
        y = nodes[i*dim+1];
        z = nodes[i*dim+2];
        meshFile << x << " " << y << " " << z << "\n";
    }
    nTri = vertices.size()/4;
    meshFile << "1\n";

    nShed = 0;
    meshFile << nTri - shed.size() << "\n";
    meshFile.close();
    // if this is a PEC mesh, write screen info
    std::cout << is_patch.size() << std::endl;
    if (pecMesh) {
```

```
        for (int l = 0; l < nPatchLayers; l++) {
            for (i = 0; i < nFaces; i++) {
                patchFile << is_patch[l*nFaces+i] << std::endl;
            }
            patchFile << std::endl;
        }
        if (useGround) {
            for (i = 0; i < nFaces; i++) {
                patchFile << "1\n";
            }
        }
        patchFile.close();
    }
    // Mesh file closed. Write material file information
    materialFile << (nFaces-shed.size()) * nlayers << std::endl;
    for (int l = 0; l < nlayers; l++) {
        nShed = 0;
        for (i = 0; i < nFaces; i++) {
            if (shed.size() == 0 || i != shed[nShed]) {

                materialFile << which_dielectric[l*nFaces+i] << std::endl;
            } else
                nShed++;
        }
        materialFile << std::endl;
    }
    materialFile.close();
}

// Read PFEBI input file
// format is:
//  AIM Method
//  unit cell size in x and y directions and slant angle
//  mesh filename
//  number of AIM grids
//  whether there are any screens -- In this case, we are using the relative permattivity
of the metal so this should be 0.
//  number of layers in the z direction
//  whether z layers have the same thickness
//  thickness of the z layers (repeated number of layers times if there is different
thicknesses)
//  Relative permittivities of medium above and below the structure
//  name of material filename
//  number of dielectric materials
//  the rest isn't imported for this part of the code.
void read_inpfem()
{
    int i;
    int zsame;

    std::string temp;
    std::ifstream inpfem ("inpfem.txt");

    getline(inpfem, temp); // AIM Method
    inpfem >> sizeX >> sizeY >> slant;
    getline(inpfem, temp);
    slant = slant * PI / 180.0;

    getline(inpfem, temp); // mesh filename
    getline(inpfem, temp); // AIM gridding
    getline(inpfem, temp); // PEC?
    if (atoi(temp.c_str())) {
        getline(inpfem, temp); // number of patches
        int nscreens = atoi(temp.c_str());
        for (i = 0; i < nscreens; i++) {
            getline(inpfem, temp); // patch layer
        }
```

```
        getline(inpfem, temp); // patch filename 'screen.txt'
        getline(inpfem, temp); // impedance
    }
    inpfem >> nlayers;
    getline(inpfem, temp);
    inpfem >> zsame;
    getline(inpfem, temp);
    if (zsame == 0) {
        for (i = 0; i < nlayers; i++)
            getline(inpfem, temp); // i don't care right now what size the layers are
    }
    getline(inpfem, temp); // half-spaces
    // we don't need anything else from inpfem
    inpfem.close();
}

bool is_digits(const std::string &s)
{
    return std::all_of(s.begin(), s.end(), ::isdigit);
}

int main(int argc, char* argv[])
{
    bool runGL = false;

    int count;
    int n;
    int fineness, meshDepth;
    int nBoundary;
    int nInnerPts;
    double coordinates[3];
    double boundary_coords[2];
    std::list<double> rightX;
    std::list<double> rightY;
    std::list<double> botX;
    std::list<double> botY;
    double right_shift;

    std::string temp;
    lambda = 1.2;
    sizeX = 2.2;
    sizeY = 2.2;
    slant = 90.0*PI/180.0;
    fineness = 15;
    meshDepth = 15;
    pecMesh = false;
    useGround = false;
    std::vector<Vertex_handle> boundary_vertices;
    std::vector<Vertex_handle> vertices;
    std::cout << "This version of the meshing code will work with any type of
surface.\n";
    std::ifstream fid ("input.bez");
    std::ofstream outfile ("output.poly");
    CDTplus cdt;

    read_inpfem();

    if (argc > 1) {
        for (count = 1; count < argc; count++) {
            if (strcmp(argv[count], "-d") == 0)
                runGL = true;
            if (strcmp(argv[count], "-f") == 0)
                fineness = atoi(argv[count+1]);
            if (strcmp(argv[count], "-i") == 0)
                meshDepth = atoi(argv[count+1]);
            if (strcmp(argv[count], "-p") == 0) {
                pecMesh = true;
```

```
                if (!is_digits(argv[count+1])) {
                    std::cerr << "Error: please enter the number of screens to mesh.\n";
                    return 1;
                }
                nPatchLayers = atoi(argv[count+1]);
            }
            if (strcmp(argv[count], "-g") == 0) {
                useGround = true;
            }
            if (strcmp(argv[count], "-h") == 0) {
                std::cout << "PFEBI Mesh Code Using the CGAL Library Help:\n";
                std::cout << "-d:         Display mesh using OpenGL\n";
                std::cout << "-f (arg):   Set Boundary Fineness Value (default 15)\n";
                std::cout << "-i (arg):   Set CGAL Mesh Fineness. Larger values yield
more triangles. (default 15)\n";
                std::cout << "-p (arg):   Use a PEC Instead of a Lossy Metal. Argument is
the number of screens.\n";
                return 0;
            }
        }
    }
    std::cout << "Display mesh: " << runGL << std::endl;

    coordinates[2] = 0.0;
    // Begin reading the input file: input.bez
    fid >> boundary_coords[0];
    fid >> boundary_coords[1];

    fid >> sizeX >> sizeY >> slant;
    slant = slant * PI / 180.0;
    // Now let's determine total number of objects defined
    //   Format: nscreen nsub ndiel
    fid >> nscreens >> ndiel1>> ndiel2 >> ngnd;
    fid >> bezsize >> bezthresh >> cmax;
    Bézier_coeffs.resize(bezsize*bezsize);
    for (int i = 0; i < bezsize; i++) {
        for (int j = 0; j < bezsize; j++) {
            fid >> Bézier_coeffs[i*bezsize+j];
        }
    }
    fid >> nSurface;
    pgn.resize(nSurface);
    isHole.resize(nSurface);
    std::vector<Point> points;
    std::cout << "Boundary " << boundary_coords[0] << " " << boundary_coords[1] << "\n";

    // before we finish reading everything, let's define the mesh boundary.
    right_shift = sizeY / tan(slant);
    if (right_shift < 0.001)
        right_shift = 0.0;
    if (fineness <= 3) {
        boundary_vertices.push_back(cdt.insert(Point(boundary_coords[0],
                                                     boundary_coords[1])));

boundary_vertices.push_back(cdt.insert(Point(boundary_coords[0]+right_shift+sizeX,
                                                     boundary_coords[1])));

boundary_vertices.push_back(cdt.insert(Point(boundary_coords[0]+right_shift+sizeX,
                                                     boundary_coords[1]+sizeY)));
        boundary_vertices.push_back(cdt.insert(Point(boundary_coords[0],
                                                     boundary_coords[1]+sizeY)));
    } else {
        for (int i = 0; i < 4; i++) {
            if (i == 2) {

boundary_vertices.push_back(cdt.insert(Point(boundary_coords[0]+right_shift+sizeX,
                                                               boundary_coords[1]+sizeY)));
```

```
            }
            else if (i == 3) {

boundary_vertices.push_back(cdt.insert(Point(boundary_coords[0]+right_shift,
                                                        boundary_coords[1]+sizeY)));
            }
            for (int j = 0; j < fineness-1; j++) {
                switch (i) {
                    case 0:
                        coordinates[0] = boundary_coords[0] + j*sizeX/((double)fineness-
1);
                        coordinates[1] = boundary_coords[1];
                        botX.push_back(coordinates[0]);
                        botY.push_back(coordinates[1]);
                        break;
                    case 1: // right side
                        if (slant * 180.0 / PI < 89.9) {
                            coordinates[0] = (boundary_coords[0] + sizeX) +
j*sizeY/tan(slant)/((double)fineness-1);
                        } else
                            coordinates[0] = boundary_coords[0]+sizeX;
                        if (slant * 180.0 / PI < 89.9)
                            coordinates[1] = boundary_coords[1]+(coordinates[0]-
boundary_coords[0]-sizeX)*tan(slant);
                        else
                            coordinates[1] =
boundary_coords[1]+j*sizeY/((double)fineness-1);
                        rightX.push_back(coordinates[0]);
                        rightY.push_back(coordinates[1]);
                        break;
                    case 2: // top
                        coordinates[0] = botX.back() + right_shift;
                        coordinates[1] = botY.back() + sizeY;
                        botX.pop_back();
                        botY.pop_back();
                        break;
                    case 3: // left
                        coordinates[0] = rightX.back() - sizeX;
                        coordinates[1] = rightY.back();
                        rightX.pop_back();
                        rightY.pop_back();
                        break;
                }
                boundary_vertices.push_back(cdt.insert(Point(coordinates[0],
coordinates[1])));
            }
        }
    }
    Polygon_2 boundary_pgn;
    double boundaryMult = 1.0;
    boundary_pgn.push_back(Point(boundaryMult*boundary_coords[0],
                                 boundaryMult*boundary_coords[1]));
    boundary_pgn.push_back(Point(boundaryMult*(boundary_coords[0]+sizeX),
                                 boundaryMult*boundary_coords[1]));
    boundary_pgn.push_back(Point(boundaryMult*(boundary_coords[0]+sizeX+right_shift),
                                 boundaryMult*(boundary_coords[1]+sizeY)));
    boundary_pgn.push_back(Point(boundaryMult*(boundary_coords[0]+right_shift),
                                 boundaryMult*(boundary_coords[1]+sizeY)));
    nBoundary = boundary_vertices.size();
    for (int i = 1; i < nBoundary; i++) {
        cdt.insert_constraint(boundary_vertices[i-1], boundary_vertices[i]);
    }
    if (nBoundary > 1) // should always occur
        cdt.insert_constraint(boundary_vertices[0], boundary_vertices[nBoundary-1]);
    clock_t begin=clock();
    Mesher mesher(cdt);
    for (int curSurface = 0; curSurface < nSurface; curSurface++) {
```

```
    bool hole;
    fid >> nInnerPts >> hole;
    isHole[curSurface] = hole;
    // load
    for (int i = 0; i < nInnerPts; i++) {
        fid >> coordinates[0] >> coordinates[1];
        double pgnCoords[2];
        double pgnMult = 1.0;
        pgnCoords[0] = coordinates[0];
        pgnCoords[1] = coordinates[1];
        Point curPt = Point(coordinates[0], coordinates[1]);
        points.push_back(curPt);
        pgn[curSurface].push_back(Point(pgnCoords[0]*pgnMult,pgnCoords[1]*pgnMult));
        vertices.push_back(cdt.insert(points.back()));
    }
    for (int i = 1; i < vertices.size(); i++) {
        cdt.insert_constraint(vertices[i-1], vertices[i]);
    }
    cdt.insert_constraint(vertices.back(),vertices[0]);
    vertices.clear();
    points.clear();
}
mesher.set_criteria(Criteria(0.125, sizeX/meshDepth));
mesher.refine_mesh();
std::cout << "Number of vertices: " << cdt.number_of_vertices() << std::endl;
std::cout << "Number of finite faces: " << cdt.number_of_faces() << std::endl;
int mesh_faces_counter = 0;
for(CDT::Finite_faces_iterator fit = cdt.finite_faces_begin();
    fit != cdt.finite_faces_end(); ++fit)
{
    if(fit->is_in_domain()) {
        ++mesh_faces_counter;

    }
}
std::cout << mesh_faces_counter << std::endl;
clock_t end = clock();
std::cout << "Time: " << (double)(end-begin)/CLOCKS_PER_SEC << std::endl;
CGAL::write_triangle_poly_file(cdt, outfile);
// output to OFF
std::string offFileName = "output.off";
write_triangle_OFF_file(cdt, offFileName);

fid.close();
outfile.close();

std::string filename = "output.off";
read_off_file(filename);

writePFEBI(nscreens);

if (runGL) {
    displayScreen = 0;
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition(100, 100);
    glutInitWindowSize(1024, 1024);
    mainWindow = glutCreateWindow("Mesh");
    glutDisplayFunc(renderScene);
    glutIdleFunc(renderScene);
    init();
    // Handle key input
    glutKeyboardFunc(handleKeypress);
    // Mouse functions
    glutMouseFunc(handleMouse);
    glutMotionFunc(mouseMove);
    glutMainLoop();
```

```
    }
    return 0;
}
```

Appendix B
Bézier Surface and Alpha Shape Code in Matlab

```
function fitness = prepare_mesh_cma(ucsize, deltaZ, deltaZdiel, deltaZgnd, ngnd, ...
          ndiel1, ndiel2, npatch, metalEr, dielEr, subEr, freq, bezvec, pol, theta, thr)
fitness = 0;
close all;
% Definitions
% First define the bottom-left corner of the mesh
corner = [-ucsize/2 -ucsize/2]; % (x, y) position
aim_type = 2;
% input.bez shortens the ucsize to six digits which causes problems if the last digit is
odd
if mod(round(ucsize*1e6),2)
  ucsize = ucsize - 1e-6;
end
sizeX = ucsize;
sizeY = ucsize;
dimap = fopen('./dielectric_map.txt','w');
for i=1:ndiel1
  fprintf(dimap, '1 0 0 0\n');
end
for i=ndiel1+1:ndiel1+npatch
  fprintf(dimap, '1 2 0 0\n');
end
for i=npatch+1:npatch+ndiel2
  fprintf(dimap, '1 0 0 0\n');
end
for i=npatch+ndiel2+1:npatch+ndiel2+ngnd
  fprintf(dimap, '0 0 0 0\n');
end
fclose(dimap);
beta = atan(sizeY/(2*abs(corner(1))-sizeX))*180/pi; %85.875; % beta angle in degrees
mesh_filename = 'mesh.txt';
gridding = 35;%floor(1.2*15/sizeX+0.5);%19;
is_pec = 0;
nScreen = npatch;
screen_pos = 1;
patchfile = 'metal.txt';
impedance_r = 0.0;
impedance_i = 0.0;
nLayers = ndiel1+npatch+ndiel2+ngnd;
const_thick = 0;
%deltaZ = 0.2;
top = abs(dielEr);
bottom = abs(dielEr); %2.2;
%ndiel = 1;
eps_r = [real(dielEr), real(metalEr)];
eps_i = [imag(dielEr), imag(metalEr)];
material_file = 'material.txt';
bicg = 2;
cgtol = 0.00001;
minIt = 2;
maxIt = 5000;
dispCG = '.FALSE.';
use_bestguess = '.TRUE.';
sim_type = 1;
phi = 0.0;
%theta = 25.0;
sweepphi = [0.0 0.0 1.0];
sweeptheta = [0.0 0.0 1.0];
polarization = pol;
% Fill initial inpfem.txt for mesher
inpfem = fopen('./inpfem.txt','w');
```

```
fprintf(inpfem, '%d\n', aim_type);
fprintf(inpfem, '%-11.5f %-11.6f %f\n', sizeX, sizeY, beta);
fprintf(inpfem, '%s\n', mesh_filename);
fprintf(inpfem, '%d %d\n', gridding, gridding);
fprintf(inpfem, '%d\n', is_pec);
if is_pec
    fprintf(inpfem, '2\n');
    fprintf(inpfem, '0\n'); % patch on top layer
    fprintf(inpfem, '%d\n', nLayers); % patch on bottom layer
    fprintf(inpfem,'screen.txt\n');
    fprintf(inpfem,'(0.0, 0.0)\n');
end
fprintf(inpfem, '%d\n', nLayers);
fprintf(inpfem, '%d\n', const_thick);
for i=1:ndiel1
    fprintf(inpfem, '%f\n', deltaZdiel);
end
for i=1:npatch
    fprintf(inpfem, '%f\n',deltaZ);
end
for i=1:ndiel2
    fprintf(inpfem, '%f\n',deltaZdiel);
end
for i=1:ngnd
    fprintf(inpfem, '%f\n',deltaZgnd);
end
fprintf(inpfem, '%f %f\n', top, bottom);
fprintf(inpfem, '%s\n', material_file);
fprintf(inpfem, '%d\n', 2);
for i=1:2
  fprintf(inpfem, '(%f, %f) (1.0, 0.0)\n', eps_r(i), eps_i(i));
end
fprintf(inpfem, '%d\n', bicg);
fprintf(inpfem, '%f %d %d\n', cgtol, minIt, maxIt);
fprintf(inpfem, '%s\n', dispCG);
fprintf(inpfem, '%s\n', use_bestguess);
fprintf(inpfem, '%d\n', sim_type);
fprintf(inpfem, '%f %f\n', phi, theta);
fprintf(inpfem, '%f %f %f\n', sweepphi(1), sweepphi(2), sweepphi(3));
fprintf(inpfem, '%f %f %f\n', sweeptheta(1), sweeptheta(2), sweeptheta(3));
fprintf(inpfem, '%d\n', polarization);
fprintf(inpfem, '%f %f %f\n', freq(1), freq(1), 0.1);
fclose(inpfem);

% Now prepare the mesher
% generate a control point matrix and rotate it for symmetry
bezsize = 18;%length(bezvec);
fineness = 75;
[u v] =  meshgrid(linspace(0,1,fineness), linspace(0,1,fineness));
p = zeros(bezsize/2,bezsize/2);
% Set up boundaries
p(9,1:9) = bezvec(1:9);
p(8,1:8) = bezvec(10:17);
p(7,1:7) = bezvec(18:24);
p(6,1:6) = bezvec(25:30);
p(5,1:5) = bezvec(31:35);
p(4,1:4) = bezvec(36:39);
p(3,1:3) = bezvec(40:42);
p(2,1:2) = bezvec(43:44);
p(1,1) = bezvec(45);
p2 = flipud(rot90(p,1));
for i=1:bezsize/2
  p2(i,i) = 0;
end
p = p + p2;
p = [p ; rot90(p,1)]; p = [p rot90(p,2)];
n = bezsize;
```

```
c = zeros(size(u));

for i = 0:n-1
    Bi = factorial(n-1)*u.^i.*(1-u).^(n-1-i)/(factorial(i)*factorial(n-1-i));
    for j = 0:n-1
        Bj = factorial(n-1)*v.^j.*(1-v).^(n-1-j)/(factorial(j)*factorial(n-1-j));
        c = c + Bi.*Bj*p(i+1,j+1);
%        disp(sprintf('maxes %f %f %f %f',max(max(c)),max(max(Bi)), max(max(Bj)),
p(i+1,j+1)))
    end
end

cmax = max(max(c));
c = c / cmax;
thresh = thr*mean(mean(c));%0.65;
if thresh > 1
  return;
end
threshHi = 0.1;
% contour(u,v,c, 'LineWidth', 2)%uint32(c > thresh));
figure;surf(u,v,c)
%axis([0 1 0 1])
mesh = fopen('./input.bez','w');
fprintf(mesh, '%1.5f %1.5f\n', corner(1), corner(2));
fprintf(mesh, '%1.5f %1.5f %f\n', sizeX, sizeY, beta);
fprintf(mesh, '%d %d %d\n', npatch, ndiel1, ndiel2, ngnd);
fprintf(mesh, '%d %f %f\n', bezsize, thresh, cmax);

for i=1:bezsize
    for j=1:bezsize
        fprintf(mesh,'%f ',p(i,j));
    end
    fprintf(mesh,'\n');
end

pts = find(abs(c - thresh) < 0.002);
ptsHi = find(abs(c - threshHi) < 0.005);
n = length(pts);
cPadded =zeros(size(c)+2);
cPadded(2:end-1,2:end-1) = c;
dx = 1/(fineness-1);
[upad vpad] = meshgrid(-dx:dx:1+dx,-dx:dx:1+dx);
beta = beta*pi/180;
xv = linspace(corner(1)+0.001,corner(1)+sizeX,25);yv = corner(2)*ones(1,25)+0.001;
if (abs(beta-pi/2) < 0.001)
  xv = [xv (corner(1)+sizeX+(1:24)*sizeY/25*0)]; yv = [yv (corner(2)+(1:24)*sizeY/25)];
  topright=corner(1)+sizeX;%+sizeY/tan(beta);
  toprighty=corner(2)+sizeY;
  xv = [xv (topright-(0:24)*sizeX/25)]; yv = [yv toprighty*ones(1,25)];
  xv = [xv xv(50:-1:26)-sizeX]; yv = [yv yv(50:-1:26)];
end
% figure;imshow(c);
%rescale and move points to be in upper right quadrant
cgray = mat2gray(c);
% im = im2bw(cgray, thresh);
im = im2bw(mat2gray(cPadded),thresh);
[L, Nim] = bwlabel(im);
L2 = bwlabel(imcomplement(im));
%figure(imshow(c))
%figure;imshow(L)
L = L(2:end-1,2:end-1);
L2 = L2(2:end-1,2:end-1);
% New Alpha Shapes code for MATLAB 2014b and above
surfaces = [];
surfIdx = 1;
isHole = [];
for ii=1:max(max(L))
```

```
    uFull = u(L == ii);
    vFull = v(L == ii);
    shp = alphaShape(uFull,vFull);
    shpNoHole = shp; shpNoHole.HoleThreshold = 5;
    numReg = numRegions(shp);
%    figure;plot(shp);figure;plot(shpNoHole)
    for i=1%:numReg
      % ignore exceedingly small components
      if (numReg < 1 || i > numReg)
        disp('No Alpha Shape to mesh: skipping');
        return;
      end
      if (area(shp,i) > 0.044)
        [tri xyz] = boundaryFacets(shp,i);
        [tri xyzNH] = boundaryFacets(shpNoHole,i);
        surfaces(surfIdx).u = xyzNH(1:1:end,1);
        surfaces(surfIdx).v = xyzNH(1:1:end,2);
        isHole(surfIdx) = false;
        holeCoords = xyz(size(xyzNH,1)+1:size(xyz,1),:);
        if numel(holeCoords) && ~isempty(holeCoords)
            if (abs(area(shp)-area(shpNoHole)) > 0.01)
              surfIdx = surfIdx + 1;
              surfaces(surfIdx).u = holeCoords(1:1:end,1);
              surfaces(surfIdx).v = holeCoords(1:1:end,2);
              isHole(surfIdx) = true;
            end
        end
        surfIdx = surfIdx + 1;
      end
    end
    figure(21);hold on
    numSurf = length(surfaces);
    for i=1:numSurf
        plot(surfaces(i).u,surfaces(i).v);
    end
end
% There are issues when the first or last coordinate of a surface is a corner
% so let's jumble them a bit
for ii=1:length(surfaces)
  doPivot = false;
  if ((surfaces(ii).u(1) == 0 || surfaces(ii).u(1) == 1) && ...
      (surfaces(ii).v(1) == 0 || surfaces(ii).v(1) == 1))
    doPivot = true;
  end
  if ((surfaces(ii).u(end) == 0 || surfaces(ii).u(end) == 1) && ...
      (surfaces(ii).v(end) == 0 || surfaces(ii).v(end) == 1))
    doPivot = true;
  end
  if doPivot
    pivotPt = floor(length(surfaces(ii).u)/2);
    surfaces(ii).u = [surfaces(ii).u(pivotPt:end); surfaces(ii).u(1:pivotPt-1)];
    surfaces(ii).v = [surfaces(ii).v(pivotPt:end); surfaces(ii).v(1:pivotPt-1)];
  end
end
ind = [];
% thin out the edges of the unit cell
adjust = 0.0;
figure(22);hold on
numSurf = length(surfaces);
for i=1:numSurf
    plot(surfaces(i).u,surfaces(i).v,'*-');
end

% surfaces are still in terms of 0 to 1 so scale
for ii=1:length(surfaces)
    surfaces(ii).u = surfaces(ii).u * sizeX - sizeX / 2;
    surfaces(ii).v = surfaces(ii).v * sizeY - sizeY / 2;
```

```
end
for ii=1:length(surfaces)
    if length(surfaces(ii).u) < 5
        ind = [ind ii];
    end
end
surfaces(ind) = [];
isHole(ind) = [];
% axis([0 1 0 1]);

fitness = 1.0;
sizeX = sizeX;
sizeY = sizeY;

nInnerPoints = 0;
nSurfaces = 0;
for ii=1:length(surfaces)
    if (length(surfaces(ii).u) > 4)
        nInnerPoints = nInnerPoints + length(surfaces(ii).u);
        nSurfaces = nSurfaces + 1;
    end
end
% fprintf(mesh, '%d\n', nInnerPoints);
fprintf(mesh, '%d\n', nSurfaces);
NS = 0;
for ii=1:length(surfaces)
    if (length(surfaces(ii).u) > 4)
        fprintf(mesh, '%d %d\n', length(surfaces(ii).u), isHole(ii));
        for j=1:length(surfaces(ii).u)
            fprintf(mesh, '%1.5f %1.5f\n', surfaces(ii).u(j), surfaces(ii).v(j));
        end
    else
        NS = NS + 1;
    end
end
fclose(mesh);
if NS == length(surfaces)
    disp('No Surfaces: returning\n');
    return;
end
inpaim = fopen('./input_aim.txt','w');
wavelength = 30 / freq;
gridsize = wavelength / sqrt(abs(dielEr)) / 10;
nearfield = 3 * gridsize;
fprintf(inpaim, '3\n');
fprintf(inpaim, '%f\n', nearfield);
fprintf(inpaim, '%f %f\n', gridsize, gridsize);
fclose(inpaim);
% figure out the proper number of boundary nodes
if is_pec
  unix('./make_mesh -f 35 -i 15 -p 1 -g &> /dev/null');
  if ans
      unix('./make_mesh -f 45 -i 25 -p 1 -g &> /dev/null');
  end
else
  unix('./make_mesh -f 2 -i 8 &>/dev/null');
end
% ensure that this mesh was valid
if (~exist('mesh.txt', 'file'))
    disp ('returning')
    return
end
fid = fopen('mesh.txt','r');
dummy = fscanf(fid, '%d', 1);
numnode = fscanf(fid, '%d', 1);
x = zeros(1,numnode); y = zeros(1,numnode);
for i=1:numnode
```

```
  xyz = fscanf(fid, '%f %f %f', [1,3]);
  x(i) = xyz(1);y(i) = xyz(2);
end
fclose(fid);
sizeY = (max(y) - min(y)) / sin(beta);
gridx = size(find(abs(x - corner(1)) < 0.001),2);
gridy = size(find(abs(y - corner(2)) < 0.001),2); % only works when there is no slant

% generate a proper inpfem
make_inpfem(false, aim_type, sizeY, beta, mesh_filename, gridx, gridy, is_pec, ...
    nLayers, const_thick, ndiel1, npatch, ndiel2, ngnd, deltaZdiel, deltaZ, ...
    deltaZgnd, top, bottom, material_file, eps_r, eps_i, bicg, cgtol, minIt, ...
    maxIt, dispCG, use_bestguess, sim_type, phi, theta, sweepphi, sweeptheta, ...
    polarization, freq)
unix('rm -f RefTE.txt RefTM.txt TrnTE.txt TrnTM.txt');
unix('~/clFEBI.x > tmpFile');
if ~exist('RefTE.txt', 'file')
  % check for atan2 issue
  make_inpfem(true, aim_type, sizeY, beta, mesh_filename, gridx, gridy, is_pec, ...
    nLayers, const_thick, ndiel1, npatch, ndiel2, ngnd, deltaZdiel, deltaZ, ...
    deltaZgnd, top, bottom, material_file, eps_r, eps_i, bicg, cgtol, minIt, ...
    maxIt, dispCG, use_bestguess, sim_type, phi, theta, sweepphi, sweeptheta, ...
    polarization, freq)
  unix('~/clFEBI.x > tmpFile');
end
if ~exist('RefTE.txt', 'file')
  % try again with a new mesh
  unix('./make_mesh -f 35 -i 5 &>/dev/null');
  % ensure that this mesh was valid
  if (~exist('mesh.txt', 'file'))
      disp ('returning')
      return
  end
  fid = fopen('mesh.txt','r');
  dummy = fscanf(fid, '%d', 1);
  numnode = fscanf(fid, '%d', 1);
  x = zeros(1,numnode); y = zeros(1,numnode);
  for i=1:numnode
    xyz = fscanf(fid, '%f %f %f', [1,3]);
    x(i) = xyz(1);y(i) = xyz(2);
  end
  fclose(fid);
  sizeY = (max(y) - min(y)) / sin(beta);
  gridx = size(find(abs(x - corner(1)) < 0.001),2);
  gridy = size(find(abs(y - corner(2)) < 0.001),2); % only works when there is no slant
  make_inpfem(false, aim_type, sizeY, beta, mesh_filename, gridx, gridy, is_pec, ...
    nLayers, const_thick, ndiel1, npatch, ndiel2, ngnd, deltaZdiel, deltaZ, ...
    deltaZgnd, top, bottom, material_file, eps_r, eps_i, bicg, cgtol, minIt, ...
    maxIt, dispCG, use_bestguess, sim_type, phi, theta, sweepphi, sweeptheta, ...
    polarization, freq)

  unix('~/clFEBI.x > tmpFile');
  if ~exist('RefTE.txt', 'file')
    % check for atan2 issue
    make_inpfem(true, aim_type, sizeY, beta, mesh_filename, gridx, gridy, is_pec, ...
    nLayers, const_thick, ndiel1, npatch, ndiel2, ngnd, deltaZdiel, deltaZ, ...
    deltaZgnd, top, bottom, material_file, eps_r, eps_i, bicg, cgtol, minIt, ...
    maxIt, dispCG, use_bestguess, sim_type, phi, theta, sweepphi, sweeptheta, ...
    polarization, freq)

    unix('~/clFEBI.x > tmpFile');
  end
end

if (exist('RefTE.txt', 'file'))
   % fitness is actually calculated in the function that calls this
   fitness = 1;
```

```
end
```

Appendix C
GPU Boundary Integral Calculation

```
#define USEDOUBLE
#define PI 0x1.921fb6p+1
#define BLOCK_SIZE 9
//#pragma OPENCL EXTENSION cl_amd_printf : enable

#ifdef USEDOUBLE
//#pragma OPENCL EXTENSION cl_amd_fp64 : enable
#pragma OPENCL EXTENSION cl_khr_fp64: enable
#define FTYPE double
#define FTYPE2 double2
#define FTYPE4 double4
#define FTYPE8 double8
#define FTYPE16 double16
#else
#define FTYPE float
#define FTYPE2 float2
#define FTYPE4 float4
#define FTYPE8 float8
#define FTYPE16 float16
#endif

FTYPE dist(FTYPE4 a, FTYPE4 b)
{
  return sqrt(pow(a.x-b.x,2)+pow(a.y-b.y,2)+pow(a.z-b.z,2)+pow(a.w-b.w,2));
}

FTYPE2 conjugate(FTYPE2 s)
{
    return (FTYPE2) (s.x, -s.y);
}

// Heron's formula
FTYPE tri_area(FTYPE16 r)
{
    FTYPE4 l = (FTYPE4) (0.0, 0.0, 0.0, 0.0);
    FTYPE s;

    l.x = dist((double4)(r.s3, r.s4, r.s5, 0.0),(double4)(r.s7, r.s8, r.s9, 0.0));
    l.y = dist((double4)(r.s0, r.s1, r.s2, 0.0),(double4)(r.s7, r.s8, r.s9, 0.0));
    l.z = dist((double4)(r.s0, r.s1, r.s2, 0.0),(double4)(r.s3, r.s4, r.s5, 0.0));

    s = (l.x + l.y + l.z)/2.0;
    s = sqrt(s*(s-l.x)*(s-l.y)*(s-l.z));

    return s;
}

// Heron's formula
FTYPE tri_area3n(FTYPE4 r1, FTYPE4 r2, FTYPE4 r3)
{
    FTYPE4 l = (FTYPE4) (0.0, 0.0, 0.0, 0.0);
    FTYPE s;
```

```
    l.x = dist(r2, r3);
    l.y = dist(r1, r3);
    l.z = dist(r1, r2);
    s = (l.x + l.y + l.z)/2.0;
    s = sqrt(s*(s-l.x)*(s-l.y)*(s-l.z));

    return s;
}

FTYPE2 comp_exp(FTYPE alpha)
{
  FTYPE c,s;
  c = native_cos(alpha); s = native_sin(alpha);
//  c = native_cos((float)alpha); s = native_sin((float)alpha);

  return (FTYPE2)(c,s);
}

FTYPE cabs(FTYPE2 a)
{
    //return sqrt(cpown(a.x,2) + cpown(a.y,2));
    return sqrt(mad(a.x,a.x,mad(a.y,a.y,0.0) ) );
//    return sqrt(mad(a.x,a.x,a.y*a.y) );
}

FTYPE4 vector_mult(FTYPE4 a, FTYPE4 b)
{
    FTYPE4 zero = (FTYPE4) (0.0, 0.0, 0.0, 0.0);
//    return (FTYPE4) (a.x*b.x, a.y*b.y, a.z*b.z, 0.0);
    return mad(a, b, zero);
}

float4 vector_multf(float4 a, float4 b)
{
    return (float4) (a.x*b.x, a.y*b.y, a.z*b.z, 0.0);
}

FTYPE2 rcmult(FTYPE a, FTYPE2 b)
{
  return (FTYPE2) (a * b.x, a * b.y);
}

FTYPE2 cmult(FTYPE2 a, FTYPE2 b)
{
    FTYPE2 out;

    // This is a direct implementation. MAD is faster
    //out = (FTYPE2)(a.x*b.x - a.y*b.y, a.x*b.y+a.y*b.x);
    out = (FTYPE2)(mad(a.x,b.x, -a.y*b.y), mad(a.x,b.y,a.y*b.x));

    return out;
}

FTYPE2 cdiv(FTYPE2 a, FTYPE2 b)
{
    FTYPE2 out;
    FTYPE div;

    div = b.x*b.x + b.y*b.y;
    out = (FTYPE2)((a.x*b.x + a.y*b.y)/div, (a.y*b.x - a.x*b.y)/div);

    return out;
}

FTYPE2 cpow(FTYPE2 z, int n)
{
```

```
  FTYPE r, theta;
  r = cabs(z);
  theta = atan2(z.y,z.x);

  z = comp_exp(theta*n);
  z *= pown(r,n);
  return z;
}

FTYPE2 cpowd(FTYPE2 z, FTYPE n)
{
  FTYPE r, theta;
  r = cabs(z);
  theta = atan2(z.y,z.x);

  z = comp_exp(theta*n);
  z *= pow(r,n);
  return z;
}

/*
// Algorithm 680 Collected Algorithms from ACM.
// TRANSACTIONS ON MATHEMATICAL SOFTWARE,      VOL. 16, NO. 1, PP. 47.
// REFERENCE - GPM POPPE, CMJ WIJERS; MORE EFFICIENT COMPUTATION OF
// THE COMPLEX ERROR-FUNCTION, ACM TRANS. MATH. SOFTWARE.
FTYPE2 wofz(FTYPE2 zp)
{
  FTYPE xabs, yabs, x, y, qrho, xabsq, xquad, yquad, xsum, ysum, xaux;
  FTYPE u1, v1, daux, u2, v2, u, v, h, h2, tx, ty, c, w1, qlambda, rx, ry, sx, sy;
  int i, n, j, nu, np1, kapn;
  FTYPE factor = 1.12837916709551257388;
  bool A, B;

  xabs = fabs(zp.x);
  yabs = fabs(zp.y);
  x = xabs / 6.3;
  y = yabs / 4.4;

  xabsq = xabs*xabs;
  xquad = xabsq - yabs*yabs;
  yquad = 2*xabs * yabs;

  qrho = x*x + y*y;
  A = false;
  B = false;
  if ( qrho < 0.085264 ) {
    A = true;
    qrho = (1.0 - 0.85*y) * sqrt(qrho);
    n = round(6 + 72 * qrho);
    j = 2*n + 1;
    xsum = 1.0/j;
    ysum = 0.0;
    for (i = n ; i > 0; i--) {
      j = j - 2;
      xaux = (xsum * xquad - ysum * yquad) / i;
      ysum = (xsum * yquad + ysum * xquad) / i;
      xsum = xaux + 1.0/j;
    }
    u1 = -factor * (xsum * yabs + ysum * xabs) + 1.0;
    v1 = factor * (xsum * xabs - ysum * yabs);
    daux = exp(-xquad);
    u2 = daux * cos(yquad);
    v2 = -daux * sin(yquad);

    u = u1 * u2 - v1 * v2;
    v = u1 * v2 + v1 * u2;
  }
```

```
else {
  if (qrho > 1.0 ) {
    h = 0.0;
    kapn = 0;
    qrho = sqrt(qrho);
    nu = round(3 + (1442/ (26*qrho + 77 ) ) );
  } else {
    qrho = (1-y)*sqrt(1-qrho);
    h = 1.88 * qrho;
    h2 = 2*h;
    kapn = round(7 + 34 * qrho);
    nu = round(16 + 26 * qrho);
  }

  if (h > 0.0) {
    qlambda = pow(h2, kapn);
    B = true;
  }
  rx = 0.0;
  ry = 0.0;
  sx = 0.0;
  sy = 0.0;

  for (n = nu; n >= 0; n--) {
    np1 = n+1;
    tx = yabs + h + np1*rx;
    ty = xabs - np1*ry;
    c = 0.5 / (tx*tx + ty * ty);
    rx = c*tx;
    ry = c*ty;
    if ( (h > 0.0) && (n < kapn) ) {
   tx = qlambda + sx;
   sx = rx * tx - ry * sy;
   sy = ry * tx + rx * sy;
   qlambda = qlambda / h2;
    }
  }

  if ( h == 0.0 ) {
    u = factor * rx;
    v = factor * ry;
  } else {
    u = factor * sx;
    v = factor * sy;
  }

  if (yabs == 0.0 )
    u = exp(-(xabs*xabs));
}

// evaluate w(z) in other quadrants
if (zp.y < 0.0) {
  if ( A ) {
    u2 = 2*u2;
    v2 = 2*v2;
  } else {
    xquad = -xquad;
    w1 = 2*exp(xquad);
    u2 = w1*cos(yquad);
    v2 = -w1*sin(yquad);
  }
  u = u2 - u;
  v = v2 - v;
  if (zp.x != 0.0)
    v = -v;
}
return (FTYPE2) (u, v);
```

```
}

FTYPE2 cerf2(FTYPE2 z)
{
  int i;
  int n = 10;
//   FTYPE twopi = 2 * PI;
  FTYPE2 f = (FTYPE2) (0.0, 0.0);
  FTYPE2 cj = (FTYPE2) (0.0, 1.0);
  FTYPE2 one = (FTYPE2) (1.0, 0.0);
  FTYPE2 cjz = cmult(-z, cj);
  FTYPE2 z2 = cpow(z,2);
//   FTYPE powarg;
  FTYPE2 cjzpow = cpow(cjz,2);

// Since cjzpow is always large (because that was a requirement for
// entering this function), just use the asymptotic approximation for
// besseli
// Actually, that wouldn't work since this ignores nu and would always
// yield f = 0;


  for (i = 0; i < n; i++) {
    if (i % 2 == 0)
      f += besseli(2*i+0.5,cjzpow) - besseli(2*i+1.5,cjzpow);
    else
      f -= besseli(2*i+0.5,cjzpow) - besseli(2*i+1.5,cjzpow);
  }
  printf("cerf2 z = (%f, %f) f = (%f, %f) ", z.x, z.y, f.x, f.y);
  f *= sqrt(2);

  f = one - f;

  f = cmult( f, exp(-z2.x)*comp_exp(-z2.y)); //cpow(-z,2) );
  FTYPE2 temp = exp(-z2.x)*comp_exp(-z2.y);
  printf("cerf2 = %f %f %f %f %f %f\n", z2.x, z2.y, f.x, f.y, temp.x, temp.y);

  return f;
}


FTYPE2 cerf(FTYPE2 z)
{
//   z = (2/sqrt(PI))*(z - cpow(z,3)/3 +
//     cpow(z,5)/10 - cpow(z,7)/42 + cpow(z,9)/216);
   FTYPE2 cj = (FTYPE2) (0.0, 1.0);
   FTYPE2 zp = cmult(cj, z);
   FTYPE2 u;
   FTYPE2 exp_arg;
   FTYPE2 out;

   u = wofz(zp);

   exp_arg = cmult(z,z);
printf("exp_arg: %f %f %f %f %f %f\n",z.x, z.y,zp.x, zp.y, u.x, u.y);
   out = cmult(cmult(-exp(exp_arg.x),comp_exp(-exp_arg.y)),u);
   return out;
}
*/

// This cerf replacement function is based on
// Abramowitz & Stegun section 7.1
//
// For large z, continued fraction 7.1.14
FTYPE2 cerf_continued_fraction(FTYPE2 zj, FTYPE sgn)
{
  bool test;
```

```
  int n = 0;

  FTYPE2 one = (FTYPE2) (1.0, 0.0);
  FTYPE2 cj = (FTYPE2) (0.0, 1.0);

  FTYPE2 z;
  FTYPE2 zsq;
  FTYPE2 y; // output variable
  FTYPE2 a, b, f, C, D, delta;
  FTYPE2 eps = (FTYPE2) (1.0e-10, 0.0);

  z = cmult(-zj, zj);

  b = one+2.0*z;
  f = b;
  C = f;
  D = (FTYPE2) (0.0, 0.0);

  while (n < 50) {
    n++;
    a = (FTYPE)n * (-4.0 * z);
    b += 2.0*one;

    D = b + cmult(a,D);
    if (cabs(D) <=eps.x)
      D = pow(eps,2);
    D = cdiv(one , D);
    C = b + cdiv(a, C);
    if (cabs(C) <= eps.x)
      C = pow(eps,2);
    delta = cmult(C, D);

    f = cmult(f, delta);

    test = (cabs(delta-one) <= eps.x);
    if (test) {
      y = f;
//      printf("Done after %d\n:", n);
      break;
    }
  } // while

  zsq = cpow(zj,2);
  y = 2.0 / sqrt(PI) * cmult(exp(-zsq.x)*comp_exp(-zsq.y), cdiv(zj, y));

  y = sgn*y;

  return y;
}

/*FTYPE2 cerf2(FTYPE2 z)
{
  int kn;
  int n;
  int np1;
  int nu;
  FTYPE factor = 1.12837916709551257388;
  FTYPE H = 0.0;
  FTYPE H2;
  FTYPE qr;
  FTYPE ql;
  FTYPE C;
  FTYPE w1;
  FTYPE rmaxreal = 0.5e150;
  FTYPE rmaxgoni = 3.53711887601422e15;
  FTYPE rmaxexp = 708.503061461606;
  FTYPE2 r;
```

```
  FTYPE2 s;
  FTYPE2 zquad = (FTYPE2)(pow(fabs(z.x),2) - pow(fabs(z.y),2), 2.0*fabs(z.x)*fabs(z.y));
  FTYPE2 one = (FTYPE2) (1.0, 0.0);
  FTYPE2 q = (FTYPE2)(fabs(z.x) / 6.3, fabs(z.y)/4.4);
  FTYPE2 u2;
  FTYPE2 exp_arg;
  FTYPE2 out;
  if ((fabs(z.x) > rmaxreal) || (fabs(z.y) > rmaxreal))
    return (FTYPE2) (1.0, 0.0);
  qr = pow(q.x,2) + pow(q.y,2);
  if (qr > 1.0) {
    kn = 0;
    qr = sqrt(qr);
    nu = round(3.0+(1442.0/(26.0*qr + 77.0)));
  } else {
    qr = (1.0-fabs(z.y)/4.4)*sqrt(1.0-qr);
    H = 1.88 * qr;
    H2 = 2*H;
    kn = round(7.0+34.0*qr);
    nu = round(16.0+26.0*qr);
  }

  if ( H > 0.0 )
    ql = pow(H2,kn);

  r = (FTYPE2) (0.0, 0.0);
  s = (FTYPE2) (0.0, 0.0);

  for (n = nu; n >= 0; n--) {
    np1 = n + 1;
    FTYPE2 T = (FTYPE2)(fabs(z.y) + H + np1*r.x, fabs(z.x) - np1*r.y);
    C = 0.5 / (pow(T.x,2) + pow(T.y,2));
    r = (FTYPE2)(C * T.x, C * T.y);
    if ((H > 0.0) && (n <= kn)) {
      T = (FTYPE2)(ql + s.x, T.y);
printf("test %f %f %f %f s %f %f %f\n", T.x, T.y, r.x, r.y, s.x, s.y, ql);
      s = (FTYPE2)(r.x*T.x - r.y*s.y, r.y*T.x + r.x*s.y);
      ql = ql / H2;
    }
  }

  printf("s = (%f, %f), r = (%f, %f) %f %d %d\n", s.x, s.y, r.x, r.y, H, kn, nu);
  if (H == 0.0)
    out = (FTYPE2)(factor * r.x, factor*r.y);
  else
    out = (FTYPE2)(factor * s.x, factor*s.y);

  if (fabs(z.y) == 0)
    out.x = exp(-pow(fabs(z.x),2));
  if (z.y < 0.0) {
    zquad = (FTYPE2)(-zquad.x, zquad.y);

    if ((zquad.y > rmaxgoni) || (zquad.x > rmaxexp))
      return (FTYPE2)(1.0, 0.0);

    w1 = 2*exp(zquad.x);
    u2 = (w1*cos(zquad.y), -w1 * sin(zquad.y));

    out = u2 - out;

    if (z.x != 0.0)
      out = (FTYPE2)(out.x, -out.y);
  }

//  exp_arg = cmult(z,z);

//  out = cmult(cmult(-exp(exp_arg.x),comp_exp(-exp_arg.y)),u);
```

```
//  printf("cerf2(%f, %f) = (%f, %f)\n", z.x, z.y, out.x, out.y);
  return out;
}*/

FTYPE2 cerf(FTYPE2 z)
{
    FTYPE2 f = (FTYPE2) (0.0, 0.0);
    FTYPE2 cj = (FTYPE2) (0.0, 1.0);
    FTYPE2 one = (FTYPE2) (1.0, 0.0);
    FTYPE2 zp = (FTYPE2)(-z.y, z.x);//cmult(cj, z);
    FTYPE2 zsq;
    FTYPE2 tmp;

//    FTYPE sgn = 1.0;
//    FTYPE2 cc;
    zsq = cpow(z,2);
//    int k;
    int m = 64;
//    int m2 = 2*m;
    FTYPE L = sqrt((FTYPE)m /2.0/sqrt(2.0));
    FTYPE2 L2 = (FTYPE2) (L, 0.0);

//    FTYPE theta;
//    FTYPE t;
//    FTYPE ff[64];
// Use pre-calculated fourier coefficients since these only depend on accuracy and not
the value for z in wofz(z)
    FTYPE a[32] = {  -1.3032e-12,3.7409e-12, 8.0303e-12,-2.1544e-11,-5.5442e-11, 1.1658e-
10, 4.1537e-10, -5.2310e-10,-3.2080e-09, 8.1249e-10, 2.3798e-08, 2.2930e-08,-1.4813e-07,
-4.1841e-07,4.2558e-07, 4.4015e-06, 6.8210e-06, -2.1410e-05, -1.3075e-04,-2.4533e-04,
3.9259e-04, 4.5195e-03, 1.9006e-02, 5.7304e-02, 1.4061e-01, 2.9544e-01, 5.4601e-01,
9.0193e-01, 1.3455e+00, 1.8257e+00, 2.2635e+00, 2.5723e+00};

    FTYPE2 zval;
    FTYPE2 p = (FTYPE2)(0.0, 0.0);
    FTYPE2 w;
//    ff[0] = 0.0;

//    FTYPE data[128];
//    for (int i = 1; i < m2; i++) {
//      k = i - m+1;
//      theta = k * PI / m;
//      t = L*tan(theta/2.0);
//      ff[i] = exp(-t*t)*(L*L + t*t);
//    }

//    for(int i = 0; i < m2; i++) {
//      data[2*i] = ff[i];
//      data[2*i+1] = 0.0;
//    }

//    cdft(128, 1, data);
//    for (int i = 0; i < m/2; i++) {
//      a[i] = pown(-1.0,i) * data[(m/2-i)*2]/m2; // a = real(fft(f))/m2
//    }
      tmp = (FTYPE2)(L-zp.y, zp.x);
      FTYPE2 tmp2 = (FTYPE2)(L+zp.y, -zp.x);
      zval = cdiv(tmp, tmp2);
    for (int i = 0; i < m/2; i++) {
      p += a[m/2-1-i]*cpow(zval,i);
    }
    w = cdiv(2*p,cpow(tmp2,2));
    w += cdiv(one/sqrt(PI),tmp2);

    w = cmult(w,exp(-zsq.x)*comp_exp(-zsq.y));
    return w;
```

```
}

int2 incr_indices2(int2 mn)
{
    int order = max(abs(mn.x),abs(mn.y));

    if (order == 0)
        mn.x = 1;
    else if (mn.x == order && mn.y > -order && mn.y < order)
        mn.y = mn.y + 1;
    else if (mn.y == order && mn.x > -order && mn.x <= order)
        mn.x = mn.x - 1;
    else if (mn.x == -order && mn.y > -order && mn.y <= order)
        mn.y = mn.y - 1;
    else if (mn.y == -order && mn.x >= -order && mn.x <= order)
        mn.x = mn.x + 1;

    return mn;
}

FTYPE2 GreenFun_SpectralDomain_Sing(FTYPE4 s, FTYPE kpm, FTYPE4 kvec,
                                FTYPE Rho_a, FTYPE Rho_b, FTYPE Sin_Gamma,
                                FTYPE Cotan_Gamma)
{
    bool loop = 1;
    int2 mn = (int2) (0, 0);
    int order;

    FTYPE threshold = 1.0e-3;

    FTYPE2 out = (FTYPE2) (0.0, 0.0);
    FTYPE2 contrib = (FTYPE2)(0.0, 0.0);
    FTYPE2 contrib_ind = (FTYPE2)(0.0, 0.0);
    FTYPE2 ktmn;
    FTYPE2 kzmn;
    FTYPE kztemp;
    FTYPE2 cj = (FTYPE2)(0.0,1.0);

    while ( loop ) {
        kzmn = (FTYPE2)(0.0, 0.0);
        ktmn.x = kvec.x + 2*PI*mn.x/Rho_a;
        ktmn.y = kvec.y + 2*PI*mn.y/(Rho_b*Sin_Gamma) - 2*PI*mn.x/Rho_a*Cotan_Gamma;
        kztemp = pown(kpm,2) - pown(ktmn.x,2) - pown(ktmn.y,2);
        if (kztemp > 0.0) {
            kzmn.x = sqrt(kztemp);
            kzmn.y = 0.0;
        } else {
            kzmn.x = 0.0;
            kzmn.y = -sqrt(fabs(kztemp));
        }
        if (cabs(kzmn) < 1e-10)
            kzmn = (FTYPE2) (1e-10, 0.0);
        contrib_ind = comp_exp(-(ktmn.x*s.x + ktmn.y*s.y));
       if (kztemp > 0.0) {
         contrib_ind = cdiv(contrib_ind, (FTYPE2) (0.0,
2.0*Rho_a*Rho_b*Sin_Gamma*kzmn.x));
         contrib_ind = cmult(contrib_ind, cerf((FTYPE2) (0.0, kzmn.x/(2*kpm))));
       } else {
         contrib_ind = cdiv(contrib_ind, (FTYPE2) (-2.0*Rho_a * Rho_b * Sin_Gamma *
kzmn.y, 0.0));
         contrib_ind = cmult(contrib_ind, cerf((FTYPE2) (-kzmn.y/(2*kpm), 0.0)));
       }

        contrib += contrib_ind;

        order = max(abs(mn.x), abs(mn.y));
        if (mn.x == order && mn.y == -order) {
```

```
            out += contrib;
            if (cabs(cdiv(contrib,out)) < threshold)
                loop = 0;
          contrib = (FTYPE2)(0.0, 0.0);
        }
        mn = incr_indices2(mn);
    }
    return out;
}

FTYPE2 GreenFun_SpectralDomain(FTYPE4 s, FTYPE kpm, FTYPE4 kvec,
                               FTYPE Rho_a, FTYPE Rho_b, FTYPE Sin_Gamma,
                               FTYPE Cotan_Gamma)
{
    bool loop = 1;
    int2 mn = (int2) (0, 0);
    int order;
    // int count_index = 0;
    FTYPE threshold = 1.0e-3;

//    FTYPE ktmnX, ktmnY;

    FTYPE2 out = (FTYPE2) (0.0, 0.0);
    FTYPE2 contrib = (FTYPE2)(0.0, 0.0);
    FTYPE2 contrib_ind;
    FTYPE2 ktmn;
    FTYPE2 kzmn;
    FTYPE kztemp;

    while ( loop ) {
        kzmn = (FTYPE2)(0.0, 0.0);
        ktmn.x = kvec.x + 2*PI*mn.x/Rho_a;
        ktmn.y = kvec.y + 2*PI*mn.y/(Rho_b*Sin_Gamma) -
       2*PI*mn.x/Rho_a*Cotan_Gamma;

        kztemp = pown(kpm,2) - pown(ktmn.x,2) - pown(ktmn.y,2);
        if (kztemp > 0.0) {
            kzmn.x = sqrt(kztemp);
            kzmn.y = 0.0;
        } else {
            kzmn.x = 0.0;
            kzmn.y = -sqrt(fabs(kztemp));
        }
//printf("kzmn %e %e\n", kzmn.x, kzmn.y);
        if (cabs(kzmn) < 1e-10)
            kzmn = (FTYPE2) (1e-10, 0.0);
//        contrib_ind = cdiv(comp_exp(-(ktmn.x*s.x + ktmn.y*s.y)),
//      cmult((FTYPE2)(-2.0*Rho_a*Rho_b*Sin_Gamma*kzmn.y,2.0*
//      Rho_a*Rho_b*Sin_Gamma*kzmn. x),
//      cerf((FTYPE2)(-kzmn.y/(2*kpm), kzmn.x/(2*kpm)))));

        contrib_ind = comp_exp(-(ktmn.x*s.x+ktmn.y*s.y));
        if (kztemp > 0.0) {
          contrib_ind = cdiv(contrib_ind, (FTYPE2)(0.0,
            2.0*Rho_a*Rho_b*Sin_Gamma*kzmn.x));
          contrib_ind = cmult(contrib_ind, cerf((FTYPE2)(0.0, kzmn.x/(2*kpm))));
        }
        else {
          contrib_ind = cdiv(contrib_ind, (FTYPE2)
            (-2.0*Rho_a*Rho_b*Sin_Gamma*kzmn.y, 0.0));
          contrib_ind = cmult(contrib_ind, cerf((FTYPE2)(-kzmn.y/(2*kpm), 0.0)));
      }
//printf("contrib_ind %e %e\n", contrib_ind.x, contrib_ind.y);
        contrib += contrib_ind;
        order = max(abs(mn.x), abs(mn.y));
        if (mn.x == order && mn.y == -order) {
            out += contrib;
```

```
            if (cabs(cdiv(contrib,out)) < threshold)
                loop = 0;
            contrib = (FTYPE2)(0.0, 0.0);
        }
        mn = incr_indices2(mn);
    }
    return out;
}

FTYPE2 GreenFun2_Normal(FTYPE s, FTYPE kpm)
{
    FTYPE Efact = kpm;

    FTYPE2 gf;// = cmult(comp_exp(-kpm*s)/s,cerf((FTYPE2)(s*Efact, -0.5))) +
              //   cmult(comp_exp(kpm*s)/s,cerf((FTYPE2)(s*Efact, 0.5)))*0.5;
 FTYPE2 temp, temp1, temp2, temp3, temp4, temp5;
temp = comp_exp(-kpm*s)/s;
//temp1 = cerf((FTYPE2)(s*Efact, -0.5));
//temp2 = cerf((FTYPE2)(s*Efact, 0.5));
temp3 = cmult(temp, cerf((FTYPE2)(s*Efact, -0.5)));
temp = comp_exp(kpm*s)/s;
temp4 = cmult(temp, cerf((FTYPE2)(s*Efact, 0.5)));
temp5 = 0.5*(temp3+temp4);
//printf("GF2 %f %f %f %f\nexp(-iks)/s = %f %f cerf = %f %f cerf_pos = %f %f
cmult %f %f %f %f\n", gf.x, gf.y, s, kpm, temp.x, temp.y, temp1.x, temp1.y, temp2.x,
temp2.y, temp3.x, temp3.y, temp5.x, temp5.y);
    gf = temp5;
    return gf;
}
FTYPE2 GreenFun2_Singularity(FTYPE s, FTYPE kpm )

{
     FTYPE factorial_temp[22] = {1.0,1.0,2.0,6.0,24.0,120.0,720.0,5040.0,40320.0,
                362880.0,3628800.0,39916800.0,4.790016e8,6.2270208e9,
                8.71782912e10,1.307674368e12,2.0922789888e13,
                3.55687428096e14,6.402373705728e15,
                1.21645100408832e17,2.43290200817664e18,
                5.109094217170944e19 };
    int p, L;

    FTYPE negOneFactor;
    FTYPE Efact = kpm;
    FTYPE2 one = (FTYPE2) (1.0, 0.0);
    FTYPE2 cj = (FTYPE2)(0.0, 1.0);
    FTYPE2 out = (FTYPE2)(0.0, 0.0);
    FTYPE2 GreenFun2_Sing_add = (FTYPE2)(0.0, 0.0);
    FTYPE2 GF_temp = (FTYPE2)(0.0, 0.0);
    FTYPE2 GF_temp_1 = (FTYPE2)(0.0, 0.0);
    FTYPE2 cj_kpm_2_Efact = (FTYPE2)(0.0, 0.5);
    FTYPE2 tmp;

    if ( s <= 1.0e-6 ) {
        out = kpm*cmult(cj,-cerf((FTYPE2)( 0.0, -0.5) ) +
                    cerf((FTYPE2)( 0.0, 0.5))) * 0.5;
        for ( p = 1; p <= 6; ++p)
            GF_temp.x += pown(0.5,2*(p))/factorial_temp[p];
      GreenFun2_Sing_add = -2.0*Efact/sqrt(PI)*(one + GF_temp);
    }
    else {
        tmp = (FTYPE2)(-pown(kpm,2)*s/2.0, pown(kpm,3)*pown(s,2)/6 - kpm);
        out = cmult(tmp,cerf((FTYPE2)(s*Efact, -0.5))) +
              cmult(conjugate(tmp),cerf((FTYPE2)(s*Efact,0.5)));
        out = out/2.0;
        GF_temp = (FTYPE2) (0.0, 0.0);
        for ( p = 1; p <= 6; ++p ) {
            GF_temp_1 = (FTYPE2) (0.0, 0.0);
            for (L = 1; L <= 2*p+1; ++L) {
```

```
                negOneFactor = ( ( L % 2 ) == 1 ) ? -1.0 : 1.0;

                GF_temp_1 += cpow(cj_kpm_2_Efact,L)*(negOneFactor+1.0)*pown(s,2*p-L)*
                             pown(Efact,2*p+1-L)/factorial_temp[L]/
                             factorial_temp[2*p+1-L];
            }

            negOneFactor = ( ( p % 2 ) == 1 ) ? -1.0 : 1.0;

            GF_temp += negOneFactor*factorial_temp[2*p]/factorial_temp[p]*GF_temp_1;
        }
        GreenFun2_Sing_add.x = -(2.0*Efact + GF_temp.x)/sqrt(PI);
        GreenFun2_Sing_add.y = GF_temp.y/sqrt(PI);
    }

    out += GreenFun2_Sing_add;

    return out;
}

FTYPE4 getr_1(FTYPE4 r1, FTYPE4 r2, FTYPE4 r3 )
{
    FTYPE4 r;
    r = (r1 + r2 + r3) / 3.0;
    return r;
}

void getr_3(FTYPE4 *ri, FTYPE4 *r)
{
  int i;
  FTYPE4 vt[3];
  vt[0] = (FTYPE4)(0.0, 1.0/6.0, 1.0/6.0, 0.0);
  vt[1] = (FTYPE4)(0.0, 1.0/6.0, 2.0/3.0, 0.0);
  vt[2] = (FTYPE4)(0.0, 2.0/3.0, 1.0/6.0, 0.0);

  for (i = 0; i < 3; i++) {
    vt[i].x = 1.0 - vt[i].y - vt[i].z;
    r[i] = (FTYPE4)(0.0, 0.0, 0.0, 0.0);
  }

  for (i = 0; i < 3; i++) {
    r[i].x = vt[i].x*ri[0].x + vt[i].y*ri[1].x + vt[i].z*ri[2].x;
    r[i].y = vt[i].x*ri[0].y + vt[i].y*ri[1].y + vt[i].z*ri[2].y;
    r[i].z = vt[i].x*ri[0].z + vt[i].y*ri[1].z + vt[i].z*ri[2].z;
  }
}

void getr_7(FTYPE4 *ri, FTYPE4 *r)
{
    int i;
    FTYPE2 v[7];
    v[0].x = 1.0/3.0;
    v[1].x = 0.10128650732345633;
    v[2].x = 0.10128650732345633;
    v[3].x = 0.797426985353;
    v[4].x = 0.470142064105115;
    v[5].x = 0.470142064105115;
    v[6].x = 0.0597158717897;
    v[0].y = 1.0/3.0;
    v[1].y = 0.10128650732345633;
    v[2].y = 0.797426985353;
    v[3].y = 0.10128650732345633;
    v[4].y = 0.470142064105115;
    v[5].y = 0.0597158717897;
    v[6].y = 0.470142064105115;
  for (i = 0; i < 7; i++) {
    r[i] = (FTYPE4)( (1.0 - v[i].x - v[i].y)*ri[0].x + v[i].x*ri[1].x + v[i].y*ri[2].x,
```

```
                    (1.0 - v[i].x - v[i].y)*ri[0].y + v[i].x*ri[1].y + v[i].y*ri[2].y,
                    (1.0 - v[i].x - v[i].y)*ri[0].z + v[i].x*ri[1].z + v[i].y*ri[2].z,
                    0.0);
  }
}

// The following is based on the paper:
//      "On the numberical integration of the linear shape functions times
//       times the 3-D green's function or its gradient on plane triangle"
//      IEEE Trian. A.P. vol.41,No.10,1993,pp1448-1455
void getr_9(FTYPE4 *ri, FTYPE4 *r)
{
    int i;
    FTYPE2 v[9];
    FTYPE4 cOne = (FTYPE4)(1.0, 0.0, 0.0, 0.0);
    FTYPE4 one = (FTYPE4) (1.0, 1.0, 1.0, 0.0);
    v[0].x = 0.5/9.0;
    v[1].x = 2.0/9.0;
    v[2].x = 0.5/9.0;
    v[3].x = 0.7222222222;
    v[4].x = 2.0/9.0;
    v[5].x = 0.3888888888;
    v[6].x = 0.3888888888;
    v[7].x = 2.0/9.0;
    v[8].x = 0.7222222222;
    v[0].y = 2.0/9.0;
    v[1].y = 0.5/9.0;
    v[2].y = 0.7222222222;
    v[3].y = 0.5/9.0;
    v[4].y = 0.3888888888;
    v[5].y = 2.0/9.0;
    v[6].y = 0.3888888888;
    v[7].y = 0.7222222222;
    v[8].y = 2.0/9.0;
  for (i = 0; i < 9; i++) {
    r[i] = (FTYPE4)( (1.0 - v[i].x - v[i].y)*ri[0].x + v[i].x*ri[1].x + v[i].y*ri[2].x,
                    (1.0 - v[i].x - v[i].y)*ri[0].y + v[i].x*ri[1].y + v[i].y*ri[2].y,
                    (1.0 - v[i].x - v[i].y)*ri[0].z + v[i].x*ri[1].z + v[i].y*ri[2].z,
                    0.0);
  }
}

//   This subroutine be used to remove the singularity based the
//   following paper:
//     "On the numberical integration of the linear shape functions times
//      times the 3-D green's function or its gradient on plane triangle"
//     IEEE Trian. A.P. vol.41,No.10,1993,pp1448-1455
//FTYPE4 singular_inte(FTYPE4 rp, FTYPE16 r_n)
FTYPE4 singular_inte(FTYPE4 rp, FTYPE4 rin1, FTYPE4 rin2, FTYPE4 rin3, int debug, int
debug2)
{
  FTYPE4 NrecipR = (FTYPE4) (0.0, 0.0, 0.0, 0.0);

  // index variables
  int i;
  FTYPE4 p1 = rin1;//(FTYPE4)(r_n.s0, r_n.s1, r_n.s2, r_n.s3);
  FTYPE4 p2 = rin2;//(FTYPE4)(r_n.s4, r_n.s5, r_n.s6, r_n.s7);
  FTYPE4 p3 = rin3;//(FTYPE4)(r_n.s8, r_n.s9, r_n.sa, r_n.sb);
  FTYPE4 p2mp1 = p2-p1;
  FTYPE4 p3mp1 = p3-p1;
  FTYPE s_intes[3];
  FTYPE4 s1;
  FTYPE4 s2;
  FTYPE4 s3;
  FTYPE4 m1;
  FTYPE4 m2;
  FTYPE4 m3;
```

```
FTYPE4 u;
FTYPE4 v;
FTYPE u3;
FTYPE v3;
FTYPE u0;
FTYPE v0;
FTYPE f2i0[3];
FTYPE f3i0[3];
FTYPE4 n;
FTYPE4 length;
FTYPE4 rpmp1;
FTYPE ua_reciprocal_R;
FTYPE va_reciprocal_R;
FTYPE u_reciprocal_R;
FTYPE v_reciprocal_R;
FTYPE conversion[9];
FTYPE tmp;
conversion[0] = 1.0;
for (i = 1; i < 9; i++)
    conversion[i] = 0.0;

FTYPE2 s[3];
FTYPE4 t[3];

// Calculate the normal vector
n = cross(p2mp1,p3mp1);
FTYPE area = tri_area3n(p1, p2, p3);//tri_area(r_n);
n = n/(2.0*area);
// calculate length vector
length = (FTYPE4)(dist(p2,p3),dist(p3,p1),dist(p2,p1),0.0);
// calculate s vectors
s1 = (p3-p2)/length.x;
s2 = (p1-p3)/length.y;
s3 = (p2-p1)/length.z;
// calculate m vectors
m1 = cross(s1,n);
m2 = cross(s2,n);
m3 = cross(s3,n);

// calculate u vector
u = p2mp1/length.z;

// calculate v vector
v = cross(n,u);
u3 = dot(p3mp1,u);
v3 = 2.0*area/length.z;
rpmp1 = rp - p1;

u0 = dot(u,rpmp1);
v0 = dot(v,rpmp1);
// calculate s and t
// x - negative;
// y - positive;
// z - 0;
s[0].x = -((length.z-u0)*(length.z-u3)+v0*v3)/length.x;
s[0].y = ((u3-u0)*(u3-length.z)+v3*(v3-v0))/length.x;
s[1].x = -(u3*(u3-u0)+v3*(v3-v0))/length.y;
s[1].y = (u0*u3+v0*v3)/length.y;
s[2].x = -u0;
s[2].y = length.z-u0;

t[0].z = (v0*(u3-length.z)+v3*(length.z-u0))/length.x;
t[1].z = (u0*v3-v0*u3)/length.y;
t[2].z = v0;
t[0].x = sqrt(pow(length.z-u0,2)+v0*v0);
t[0].y = sqrt(pow(u3-u0,2)+pow(v3-v0,2));
t[1].y = sqrt(u0*u0+v0*v0);
```

```
  t[1].x = t[0].y;
  t[2].x = t[1].y;
  t[2].y = t[0].x;

  for (i = 0; i < 3; i++) {
      tmp = (t[i].y + s[i].y)/(t[i].x + s[i].x);
      f2i0[i] = log(tmp);//log((t[i].y + s[i].y)/(t[i].x + s[i].x));
      f3i0[i] = s[i].y*t[i].y - s[i].x*t[i].x + pow(t[i].z,2)*f2i0[i];
  }

  NrecipR.x = 0.0;
  ua_reciprocal_R = 0.0;
  va_reciprocal_R = 0.0;
  for (i = 0; i < 3; i++) {
      NrecipR.x += t[i].z*f2i0[i];
  }

  ua_reciprocal_R += 0.5*(dot(u,m1)*f3i0[0] + dot(u,m2)*f3i0[1] + dot(u,m3)*f3i0[2]);
  va_reciprocal_R += 0.5*(dot(v,m1)*f3i0[0] + dot(v,m2)*f3i0[1] + dot(v,m3)*f3i0[2]);

  u_reciprocal_R = u0*NrecipR.x + ua_reciprocal_R;
  v_reciprocal_R = v0*NrecipR.x + va_reciprocal_R;

  // convert the singular integral into a normal one
  conversion[0*3 + 1] = -1.0/length.z;
  conversion[0*3 + 2] = (u3/length.z-1.0)/v3;
  conversion[1*3 + 1] = 1.0/length.z;
  conversion[1*3 + 2] = -(u3/length.z)/v3;
  conversion[2*3 + 2] = 1.0/v3;

  s_intes[0] = NrecipR.x;
  s_intes[1] = u_reciprocal_R;
  s_intes[2] = v_reciprocal_R;

  NrecipR.y += conversion[0] * s_intes[0] + conversion[1] * s_intes[1] + conversion[2] *
s_intes[2];
  NrecipR.z += conversion[3] * s_intes[0] + conversion[4] * s_intes[1] + conversion[5] *
s_intes[2];
  NrecipR.w = v_reciprocal_R/v3;

  return NrecipR;
}

FTYPE2 SelfCellG1(FTYPE edge_L1I, FTYPE4 node_P1I,
          FTYPE4 node_P2I, FTYPE4 node_P3I, FTYPE edge_L1J,
          FTYPE4 node_P1J, FTYPE4 node_P2J, FTYPE4 node_P3J,
         int m_point, int n_point, FTYPE4 kpm, int debug, int debug2)
{
  int i;
  FTYPE2 out;
  FTYPE2 GreenFun;
  int mp, np;

  FTYPE4 r_m[3];
  FTYPE4 r_n[3];
  FTYPE4 rm1379[9];
  FTYPE4 rn1379[9];
  FTYPE4 rou_m[9];
  FTYPE4 rou_n[9];
  FTYPE4 rm, rn;
  FTYPE w_m[9];
  FTYPE w_n[9];
  FTYPE s;
  FTYPE4 NrecipR;
  FTYPE area;
  FTYPE SingularInt1overR;
```

```
r_m[0] = (FTYPE4)(node_P1I.x, node_P1I.y, 0.0, 0.0);
r_m[1] = (FTYPE4)(node_P2I.x, node_P2I.y, 0.0, 0.0);
r_m[2] = (FTYPE4)(node_P3I.x, node_P3I.y, 0.0, 0.0);

r_n[0] = (FTYPE4)(node_P1J.x, node_P1J.y, 0.0, 0.0);
r_n[1] = (FTYPE4)(node_P2J.x, node_P2J.y, 0.0, 0.0);
r_n[2] = (FTYPE4)(node_P3J.x, node_P3J.y, 0.0, 0.0);

if (m_point == 1) {
  rm1379[0] = getr_1(r_m[0], r_m[1], r_m[2]);
  rou_m[0] = rm1379[0] - r_m[0];
  w_m[0] = 1.0;
} else if (m_point == 3) {
  getr_3(r_m, rm1379);
  for (i = 0; i < 3; ++i) {
    rou_m[i] = rm1379[i] - r_m[0];
    w_m[i] = 1.0/3.0;
  }
} else if (m_point == 7) {
  getr_7(r_m, rm1379);
  for (i = 0; i < 7; i++) {
    rou_m[i] = rm1379[i] - r_m[0];
    if ( i < 4)
   w_m[i] = 0.125939180544827152595683945500l8;
    else
   w_m[i] = 0.132394152788506180737649387833l5;
  }
  w_m[0] = 0.225;
} else {
  getr_9(r_m, rm1379);
  for (i = 0; i < 9; i++) {
    rou_m[i] = rm1379[i] - r_m[0];
    w_m[i] = 1.0 / 9.0;
  }
}

if (n_point == 1) {
  rn1379[0] = getr_1(r_n[0], r_n[1], r_n[2]);
  rou_n[0] = rn1379[0] - r_n[0];
  w_n[0] = 1.0;
} else if (n_point == 3) {
  getr_3(r_n, rn1379);
  for (i = 0; i < 3; ++i) {
    rou_n[i] = rn1379[i] - r_n[0];
    w_n[i] = 1.0/3.0;
  }
} else if (n_point == 7) {
  getr_7(r_n, rn1379);
  for (i = 0; i < 7; i++) {
    rou_n[i] = rn1379[i] - r_n[0];
    if ( i < 4)
   w_n[i] = 0.125939180544827152595683945500l8;
    else
   w_n[i] = 0.132394152788506180737649387833l5;
  }
  w_n[0] = 0.225;
} else {
  getr_9(r_n, rn1379);
  for (i = 0; i < 9; i++) {
    rou_n[i] = rn1379[i] - r_n[0];
    w_n[i] = 1.0 / 9.0;
  }
}

out = (FTYPE2) (0.0, 0.0);

for (mp = 0; mp < m_point; mp++) {
```

```
    rm = rm1379[mp];
    for (np = 0; np < n_point; np++) {
      rn = rn1379[np];
      // Green's Function
      s = sqrt(pow(rm.x-rn.x,2) + pow(rm.y-rn.y,2));
      GreenFun = GreenFun2_Singularity(s, kpm.x);
      out += w_m[mp]*w_n[np]*(dot(rou_m[mp],rou_n[np]) -
     4.0 / pow(kpm.x,2))*GreenFun;
    }

    NrecipR = singular_inte(rm, r_n[0], r_n[1], r_n[2], debug, debug2);
    area = tri_area3n(r_n[0], r_n[1], r_n[2]);

    SingularInt1overR = (dot(rou_m[mp], r_n[0])*NrecipR.y +
      dot(rou_m[mp], r_n[1]) * NrecipR.z +
      dot(rou_m[mp], r_n[2]) * NrecipR.w -
      dot(rou_m[mp], r_n[0]) * NrecipR.x - 4.0 / pow(kpm.x,2) *
      NrecipR.x) / area;
      out.x += w_m[mp] * SingularInt1overR;

  }
  out = rcmult( - edge_L1I * edge_L1J / (8.0 * PI), out);

  return out;
} // SelfCellG1

FTYPE2 NearCellG1(FTYPE edge_L1I, FTYPE4 node_P1I,
          FTYPE4 node_P2I, FTYPE4 node_P3I, FTYPE edge_L1J,
          FTYPE4 node_P1J, FTYPE4 node_P2J, FTYPE4 node_P3J,
         int m_point, int n_point, FTYPE4 kpm, int debug, int debug2)
{
  int i;
  FTYPE2 out;
  FTYPE2 GreenFun;
  int mp, np;

  FTYPE4 r_m[3];
  FTYPE4 r_n[3];
  FTYPE4 rm1379[9];
  FTYPE4 rn1379[9];
  FTYPE4 rou_m[9];
  FTYPE4 rou_n[9];
  FTYPE4 rm, rn;
  FTYPE w_m[9];
  FTYPE w_n[9];
  FTYPE s;

  r_m[0] = (FTYPE4)(node_P1I.x, node_P1I.y, 0.0, 0.0);
  r_m[1] = (FTYPE4)(node_P2I.x, node_P2I.y, 0.0, 0.0);
  r_m[2] = (FTYPE4)(node_P3I.x, node_P3I.y, 0.0, 0.0);

  r_n[0] = (FTYPE4)(node_P1J.x, node_P1J.y, 0.0, 0.0);
  r_n[1] = (FTYPE4)(node_P2J.x, node_P2J.y, 0.0, 0.0);
  r_n[2] = (FTYPE4)(node_P3J.x, node_P3J.y, 0.0, 0.0);

  if (m_point == 1) {
    rm1379[0] = getr_1(r_m[0], r_m[1], r_m[2]);
    rou_m[0] = rm1379[0] - r_m[0];
    w_m[0] = 1.0;
  } else if (m_point == 3) {
    getr_3(r_m, rm1379);
    for (i = 0; i < 3; ++i) {
      rou_m[i] = rm1379[i] - r_m[0];
      w_m[i] = 1.0/3.0;
    }
  } else if (m_point == 7) {
    getr_7(r_m, rm1379);
```

```
    for (i = 0; i < 7; i++) {
      rou_m[i] = rm1379[i] - r_m[0];
      if ( i < 4)
     w_m[i] = 0.125939180544827152595683945500 18;
      else
     w_m[i] = 0.132394152788506180737649387833 15;
    }
    w_m[0] = 0.225;
  } else {
    getr_9(r_m, rm1379);
    for (i = 0; i < 9; i++) {
      rou_m[i] = rm1379[i] - r_m[0];
      w_m[i] = 1.0 / 9.0;
    }
  }

  if (n_point == 1) {
    rn1379[0] = getr_1(r_n[0], r_n[1], r_n[2]);
    rou_n[0] = rn1379[0] - r_n[0];
    w_n[0] = 1.0;
  } else if (n_point == 3) {
    getr_3(r_n, rn1379);
    for (i = 0; i < 3; ++i) {
      rou_n[i] = rn1379[i] - r_n[0];
      w_n[i] = 1.0/3.0;
    }
  } else if (n_point == 7) {
    getr_7(r_n, rn1379);
    for (i = 0; i < 7; i++) {
      rou_n[i] = rn1379[i] - r_n[0];
      if ( i < 4)
     w_n[i] = 0.125939180544827152595683945500 18;
      else
     w_n[i] = 0.132394152788506180737649387833 15;
    }
    w_n[0] = 0.225;
  } else {
    getr_9(r_n, rn1379);
    for (i = 0; i < 9; i++) {
      rou_n[i] = rn1379[i] - r_n[0];
      w_n[i] = 1.0 / 9.0;
    }
  }
  out = (FTYPE2) (0.0, 0.0);
  for (mp = 0; mp < m_point; mp++) {
    rm = rm1379[mp];
    for (np = 0; np < n_point; np++) {
      rn = rn1379[np];
      // Normal Green's Function
      s = sqrt(pow(rm.x-rn.x,2) + pow(rm.y-rn.y,2));
      GreenFun = GreenFun2_Normal(s, kpm.x);

      out += w_m[mp]*w_n[np]*(dot(rou_m[mp],rou_n[np]) - 4.0 / pow(kpm.x,2)) * GreenFun;
    }
  }

  out = rcmult( - edge_L1I * edge_L1J / (8.0 * PI), out);
  return out;
} // NearCellG1

FTYPE2 G2Num_Norm(FTYPE edge_L1I, FTYPE4 node_P1I,
          FTYPE4 node_P2I, FTYPE4 node_P3I, FTYPE edge_L1J,
          FTYPE4 node_P1J, FTYPE4 node_P2J, FTYPE4 node_P3J,
         int m_point, int n_point, FTYPE4 kpm, FTYPE4 kvec,
         FTYPE Sin_Gamma, FTYPE Cos_Gamma, int debug, int debug2)
{
  FTYPE2 out;
```

```
int i;
int mp, np;

FTYPE4 r_m[3];
FTYPE4 r_n[3];
FTYPE4 rm1379[9];
FTYPE4 rn1379[9];
FTYPE4 rou_m[9];
FTYPE4 rou_n[9];
FTYPE4 rm, rn;
FTYPE w_m[9];
FTYPE w_n[9];
FTYPE Cotan_Gamma = Cos_Gamma / Sin_Gamma;
FTYPE2 GreenFun;

r_m[0] = (FTYPE4)(node_P1I.x, node_P1I.y, 0.0, 0.0);
r_m[1] = (FTYPE4)(node_P2I.x, node_P2I.y, 0.0, 0.0);
r_m[2] = (FTYPE4)(node_P3I.x, node_P3I.y, 0.0, 0.0);

r_n[0] = (FTYPE4)(node_P1J.x, node_P1J.y, 0.0, 0.0);
r_n[1] = (FTYPE4)(node_P2J.x, node_P2J.y, 0.0, 0.0);
r_n[2] = (FTYPE4)(node_P3J.x, node_P3J.y, 0.0, 0.0);

if (m_point == 1) {
  rm1379[0] = getr_1(r_m[0], r_m[1], r_m[2]);
  rou_m[0] = rm1379[0] - r_m[0];
  w_m[0] = 1.0;
} else if (m_point == 3) {
  getr_3(r_m, rm1379);
  for (i = 0; i < 3; ++i) {
    rou_m[i] = rm1379[i] - r_m[0];
    w_m[i] = 1.0/3.0;
  }
} else if (m_point == 7) {
  getr_7(r_m, rm1379);
  for (i = 0; i < 7; i++) {
    rou_m[i] = rm1379[i] - r_m[0];
    if ( i < 4)
   w_m[i] = 0.125939180544827152595683945500 18;
    else
   w_m[i] = 0.132394152788506180737649387833 15;
  }
  w_m[0] = 0.225;
} else {
  getr_9(r_m, rm1379);
  for (i = 0; i < 9; i++) {
    rou_m[i] = rm1379[i] - r_m[0];
    w_m[i] = 1.0 / 9.0;
  }
}

if (n_point == 1) {
  rn1379[0] = getr_1(r_n[0], r_n[1], r_n[2]);
  rou_n[0] = rn1379[0] - r_n[0];
  w_n[0] = 1.0;
} else if (n_point == 3) {
  getr_3(r_n, rn1379);
  for (i = 0; i < 3; ++i) {
    rou_n[i] = rn1379[i] - r_n[0];
    w_n[i] = 1.0/3.0;
  }
} else if (n_point == 7) {
  getr_7(r_n, rn1379);
  for (i = 0; i < 7; i++) {
    rou_n[i] = rn1379[i] - r_n[0];
    if ( i < 4)
```

```
      w_n[i] = 0.125939180544827152595683945500 18;
       else
      w_n[i] = 0.132394152788506180737649387833 15;
    }
    w_n[0] = 0.225;
  } else {
    getr_9(r_n, rn1379);
    for (i = 0; i < 9; i++) {
      rou_n[i] = rn1379[i] - r_n[0];
      w_n[i] = 1.0 / 9.0;
    }
  }

  out = (FTYPE2) (0.0, 0.0);
  for (mp = 0; mp < m_point; mp++) {
    rm = rm1379[mp];
    for (np = 0; np < n_point; np++) {
      rn = rn1379[np];

      // Spectral Domain Green's Function
      GreenFun = GreenFun_SpectralDomain(rm-rn, kpm.x, kvec,
     kpm.y, kpm.z, Sin_Gamma, Cotan_Gamma);

      out += rcmult(w_m[mp]*w_n[mp]*(dot(rou_m[mp],rou_n[np]) -
     4.0 / pow(kpm.x,2)),GreenFun);
    }
  }

  out = rcmult( - 0.5 * edge_L1I * edge_L1J, out);
  return out;
} // G2Num_Norm

FTYPE2 G2Num_Sing(FTYPE edge_L1I, FTYPE4 node_P1I,
          FTYPE4 node_P2I, FTYPE4 node_P3I, FTYPE edge_L1J,
          FTYPE4 node_P1J, FTYPE4 node_P2J, FTYPE4 node_P3J,
         int m_point, int n_point, FTYPE4 kpm, FTYPE4 kvec,
         FTYPE Sin_Gamma, FTYPE Cos_Gamma)
{
  FTYPE2 out;

  int i;
  int mp, np;

  FTYPE4 r_m[3];
  FTYPE4 r_n[3];
  FTYPE4 rm1379[9];
  FTYPE4 rn1379[9];
  FTYPE4 rou_m[9];
  FTYPE4 rou_n[9];
  FTYPE4 rm, rn;
  FTYPE w_m[9];
  FTYPE w_n[9];
  FTYPE Cotan_Gamma = Cos_Gamma / Sin_Gamma;
  FTYPE2 GreenFun;

  r_m[0] = (FTYPE4)(node_P1I.x, node_P1I.y, 0.0, 0.0);
  r_m[1] = (FTYPE4)(node_P2I.x, node_P2I.y, 0.0, 0.0);
  r_m[2] = (FTYPE4)(node_P3I.x, node_P3I.y, 0.0, 0.0);

  r_n[0] = (FTYPE4)(node_P1J.x, node_P1J.y, 0.0, 0.0);
  r_n[1] = (FTYPE4)(node_P2J.x, node_P2J.y, 0.0, 0.0);
  r_n[2] = (FTYPE4)(node_P3J.x, node_P3J.y, 0.0, 0.0);

  if (m_point == 1) {
    rm1379[0] = getr_1(r_m[0], r_m[1], r_m[2]);
    rou_m[0] = rm1379[0] - r_m[0];
    w_m[0] = 1.0;
```

```
} else if (m_point == 3) {
  getr_3(r_m, rm1379);
  for (i = 0; i < 3; ++i) {
    rou_m[i] = rm1379[i] - r_m[0];
    w_m[i] = 1.0/3.0;
  }
} else if (m_point == 7) {
  getr_7(r_m, rm1379);
  for (i = 0; i < 7; i++) {
    rou_m[i] = rm1379[i] - r_m[0];
    if ( i < 4)
   w_m[i] = 0.125939180544827152595683945500 18;
    else
   w_m[i] = 0.132394152788506180737649387833 15;
  }
  w_m[0] = 0.225;
} else {
  getr_9(r_m, rm1379);
  for (i = 0; i < 9; i++) {
    rou_m[i] = rm1379[i] - r_m[0];
    w_m[i] = 1.0 / 9.0;
  }
}

if (n_point == 1) {
  rn1379[0] = getr_1(r_n[0], r_n[1], r_n[2]);
  rou_n[0] = rn1379[0] - r_n[0];
  w_n[0] = 1.0;
} else if (n_point == 3) {
  getr_3(r_n, rn1379);
  for (i = 0; i < 3; ++i) {
    rou_n[i] = rn1379[i] - r_n[0];
    w_n[i] = 1.0/3.0;
  }
} else if (n_point == 7) {
  getr_7(r_n, rn1379);
  for (i = 0; i < 7; i++) {
    rou_n[i] = rn1379[i] - r_n[0];
    if ( i < 4)
   w_n[i] = 0.125939180544827152595683945500 18;
    else
   w_n[i] = 0.132394152788506180737649387833 15;
  }
  w_n[0] = 0.225;
} else {
  getr_9(r_n, rn1379);
  for (i = 0; i < 9; i++) {
    rou_n[i] = rn1379[i] - r_n[0];
    w_n[i] = 1.0 / 9.0;
  }
}

out = (FTYPE2) (0.0, 0.0);
for (mp = 0; mp < m_point; mp++) {
  rm = rm1379[mp];
  for (np = 0; np < n_point; np++) {
    rn = rn1379[np];

    // Spectral Domain Green's Function
    GreenFun = GreenFun_SpectralDomain_Sing(rm-rn, kpm.x, kvec,
   kpm.y, kpm.z, Sin_Gamma, Cotan_Gamma);
    out += rcmult(w_m[mp]*w_n[mp]*(dot(rou_m[mp],rou_n[np]) -
   4.0 / pow(kpm.x,2)),GreenFun);
  }
}

out = rcmult( - 0.5 * edge_L1I * edge_L1J, out);
```

```
  return out;
} // G2Num_Sing

FTYPE2 BIContribG1(bool lSelfCell, FTYPE edge_L1I, FTYPE4 node_P1I,
          FTYPE4 node_P2I, FTYPE4 node_P3I, FTYPE edge_L1J,
          FTYPE4 node_P1J, FTYPE4 node_P2J, FTYPE4 node_P3J, FTYPE4 kpm,
          FTYPE Sin_Gamma, FTYPE Cos_Gamma, FTYPE4 kvec, int debug, int debug2)
{
  bool loop = true;
  bool lSelfCell_temp;
  int order;
  int Npoint_test, Npoint_source;
  int Npoint_test_singularity, Npoint_source_singularity;
  int test;
  int2 mn;
  FTYPE4 node_P1J_addM, node_P2J_addM, node_P3J_addM;
  FTYPE Rho_a = kpm.y;
  FTYPE Rho_b = kpm.z;
  FTYPE TestPatch_center[2];
  FTYPE SourcePatch_center[2];
  FTYPE distance_center_IJ;//, distance_center_IJ_old;
  FTYPE threshold;
  FTYPE2 contrib;
  FTYPE2 contrib_dup, contrib_ind;
  FTYPE2 factor;
  FTYPE2 cj = (FTYPE2) (0.0, 1.0);

  TestPatch_center[0] = (node_P1I.x + node_P2I.x + node_P3I.x) / 3.0;
  TestPatch_center[1] = (node_P1I.y + node_P2I.y + node_P3I.y) / 3.0;
  SourcePatch_center[0] = (node_P1J.x + node_P2J.x + node_P3J.x)/3.0;
  SourcePatch_center[1] = (node_P1J.y + node_P2J.y + node_P3J.y)/3.0;
  test = 1;
  if (test == 0) {
    Npoint_test = 1;
    Npoint_source = 1;
  } else if ( test == 1 ) {
    Npoint_test = 3;
    Npoint_source = 3;
  }
  Npoint_source_singularity = 9;
  Npoint_test_singularity = 7;

  threshold = 0.001;
  mn.x = 0;
  mn.y = 0;
  contrib = (FTYPE2) (0.0, 0.0);
  contrib_dup = (FTYPE2) (0.0, 0.0);

  while (loop) {

    contrib_ind = (FTYPE2) (0.0, 0.0);
    node_P1J_addM.x = node_P1J.x + (FTYPE)mn.x * kpm.y + (FTYPE)mn.y*kpm.z*Cos_Gamma;
    node_P1J_addM.y = node_P1J.y + (FTYPE)mn.y * kpm.z * Sin_Gamma;

    node_P2J_addM.x = node_P2J.x + (FTYPE)mn.x*kpm.y + (FTYPE)mn.y*kpm.z*Cos_Gamma;
    node_P2J_addM.y = node_P2J.y + (FTYPE)mn.y * kpm.z * Sin_Gamma;

    node_P3J_addM.x = node_P3J.x + (FTYPE)mn.x*kpm.y + (FTYPE)mn.y*kpm.z*Cos_Gamma;
    node_P3J_addM.y = node_P3J.y + (FTYPE)mn.y * kpm.z * Sin_Gamma;

    SourcePatch_center[0] = (node_P1J_addM.x + node_P2J_addM.x +
      node_P3J_addM.x) / 3.0;
    SourcePatch_center[1] = (node_P1J_addM.y + node_P2J_addM.y +
      node_P3J_addM.y) / 3.0;

    distance_center_IJ = sqrt(pow(SourcePatch_center[0] -
      TestPatch_center[0],2) + pow(SourcePatch_center[1] -
```

```
      TestPatch_center[1],2));
    lSelfCell_temp = false;
    if (lSelfCell && mn.x == 0 && mn.y == 0)
      lSelfCell_temp = true;
    if (distance_center_IJ < 1.0e-6 )
      lSelfCell_temp = true;

    if (lSelfCell_temp) {
      contrib_ind = SelfCellG1(edge_L1I, node_P1I, node_P2I, node_P3I,
        edge_L1J, node_P1J_addM, node_P2J_addM,
        node_P3J_addM, Npoint_test_singularity,
        Npoint_source_singularity, kpm, debug, debug2);
    } else {
      contrib_ind = NearCellG1(edge_L1I, node_P1I, node_P2I, node_P3I,
        edge_L1J, node_P1J_addM, node_P2J_addM,
        node_P3J_addM, Npoint_test,
        Npoint_source, kpm, debug, debug2);
    }
    contrib_ind = kpm.x*kpm.x*contrib_ind;//rcmult(kpm.x*kpm.x, contrib_ind);
    factor = comp_exp(-(kvec.x*(mn.x*Rho_a +
      mn.y*Rho_b*Cos_Gamma) + kvec.y * mn.y * Rho_b*Sin_Gamma));
    contrib_ind = cmult(contrib_ind, factor);

    contrib_dup += contrib_ind;
    order = (abs(mn.x) >= abs(mn.y)) ? abs(mn.x) : abs(mn.y);

    if (mn.x == order && mn.y == -order) {
      contrib += contrib_dup;
      if (cabs(cdiv(contrib_dup, contrib)) < threshold) {
     loop = false;
        break;
      }

      contrib_dup = (FTYPE2) (0.0, 0.0);
    }

    mn = incr_indices2(mn);
  } // Loop


  return contrib;
} // BIContribG1

FTYPE2 BIContribG2(bool lSelfCell, FTYPE edge_L1I, FTYPE4 node_P1I,
          FTYPE4 node_P2I, FTYPE4 node_P3I, FTYPE edge_L1J,
          FTYPE4 node_P1J, FTYPE4 node_P2J, FTYPE4 node_P3J, FTYPE4 kpm,
          FTYPE Sin_Gamma, FTYPE Cos_Gamma, FTYPE4 kvec, int debug, int debug2)
{
  int test;
  int Npoint_test, Npoint_source;
  int Npoint_test_singularity, Npoint_source_singularity;
  FTYPE2 contrib = (FTYPE2) (0.0, 0.0);

  test = 1;
  if (test == 0) {
    Npoint_test = 1;
    Npoint_source = 1;
  } else if ( test == 1 ) {
    Npoint_test = 3;
    Npoint_source = 3;
  }
  Npoint_source_singularity = 9;
  Npoint_test_singularity = 7;

  if (lSelfCell) {
    contrib = G2Num_Sing(edge_L1I, node_P1I, node_P2I, node_P3I,
              edge_L1J, node_P1J, node_P2J, node_P3J,
```

```
                Npoint_test_singularity, Npoint_source_singularity,
                kpm, kvec, Sin_Gamma, Cos_Gamma);
  } else {
    contrib = G2Num_Norm(edge_L1I, node_P1I, node_P2I, node_P3I,
                edge_L1J, node_P1J, node_P2J, node_P3J,
                Npoint_test, Npoint_source,
                kpm, kvec, Sin_Gamma, Cos_Gamma, debug, debug2);
  }
  return contrib;
} // BIContribG2

FTYPE2 phaseshift(int sourceType, int testType, FTYPE2 PhaseXFwd, FTYPE2 PhaseXBwd,
          FTYPE2 PhaseYFwd, FTYPE2 PhaseYBwd)
{
  FTYPE2 phase;
  FTYPE2 PhaseX = (FTYPE2) (1.0, 0.0);
  FTYPE2 PhaseY = (FTYPE2) (1.0, 0.0);
  if ( sourceType == 2 || sourceType == 7 )
    PhaseX = cmult(PhaseX,PhaseXFwd);
  if (testType == 2 || testType == 7)
    PhaseX = cmult(PhaseX,PhaseXBwd);
  if (sourceType == 4 || sourceType == 6)
    PhaseY = cmult(PhaseY, PhaseYFwd);
  if (testType == 4 || testType == 6)
    PhaseY = cmult(PhaseY, PhaseYBwd);
  if (sourceType == 8) {
    PhaseX = cmult(PhaseXFwd, PhaseX);
    PhaseY = cmult(PhaseY, PhaseYFwd);
  }
  if (testType == 8) {
    PhaseX = cmult(PhaseX, PhaseXBwd);
    PhaseY = cmult(PhaseY, PhaseYBwd);
  }
  phase = cmult(PhaseX, PhaseY);
  return phase;
}

FTYPE2 CalZBI_via_EdgetoEdge_AIM(FTYPE4 kpm, int2 mn,
                                 __global FTYPE2 *pTri,
                                 __global FTYPE2 *mTri,
                                 __global int2 *edge,
                                 __global FTYPE *LenComm,
                                 __global int *pbc,
                                 __global int *pmEle,
                                 __global int *CommEdge,
                                 FTYPE2 csGamma, int nCom, FTYPE4 kvec,
                                 FTYPE2 PhaseXFwd, FTYPE2 PhaseXBwd,
                                 FTYPE2 PhaseYFwd, FTYPE2 PhaseYBwd, int debug, int
debug2)
{
  bool lSelf, lSelf_temp;
  int i;
  int tElement, sElement, testElement, sourceElement;
  int testEdge, sourceEdge;
  int m_temp, n_temp;
  FTYPE w_mTri, w_nTri;
  FTYPE edge_L1I, edge_L1J;
  FTYPE2 contribpm, contribpm_add;

  FTYPE4 node_PI[3], node_PJ[3];


  for (i = 0; i < 3; i++) {
    node_PI[i] = (FTYPE4) (0.0, 0.0, 0.0, 0.0);
    node_PJ[i] = (FTYPE4) (0.0, 0.0, 0.0, 0.0);
  }
```

```
  FTYPE2 Zmn = (FTYPE2) (0.0, 0.0);
  if (mn.x < nCom)
    m_temp = mn.x;
  else {
    m_temp = edge[(mn.x - nCom)].x;
  }
  if (mn.y < nCom)
    n_temp = mn.y;
  else {
    n_temp = edge[(mn.y - nCom)].x;
  }
  edge_L1I = LenComm[m_temp];
  edge_L1J = LenComm[n_temp];

  for (tElement = 0; tElement < 2; tElement++) {
    testEdge = pbc[m_temp*2 + tElement];
    testElement = pmEle[m_temp*2 + tElement];

    w_mTri = 0.0;
    if (CommEdge[mn.x*2 + tElement] == 1) {
      for (i = 0; i < 3; i++) {
        node_PI[i] = (FTYPE4) (pTri[m_temp*3 + i].x, pTri[m_temp*3 + i].y, 0.0, 0.0);
        w_mTri = 1.0;
      }
    } else if (CommEdge[mn.x*2 + tElement] == -1) {
      for (i = 0; i < 3; i++) {
        node_PI[i] = (FTYPE4) (mTri[m_temp*3 + i].x, mTri[m_temp*3 + i].y, 0.0, 0.0);
        w_mTri = -1.0;
      }
    } else {
      break;
    }

    for (sElement = 0; sElement < 2; sElement++) {
      sourceEdge = pbc[n_temp*2 + sElement];
      sourceElement = pmEle[n_temp*2 + sElement];

      w_nTri = 0.0;

      if (CommEdge[mn.y*2 + sElement] == 1) {
        for (i = 0; i < 3; i++) {
          node_PJ[i] = (FTYPE4) (pTri[n_temp*3 + i].x, pTri[n_temp*3 + i].y, 0.0, 0.0);
          w_nTri = 1.0;

        }
      } else if (CommEdge[mn.y*2 + sElement] == -1) {

        for (i = 0; i < 3; i++) {
          node_PJ[i] = (FTYPE4) (mTri[n_temp*3 + i].x, mTri[n_temp*3 + i].y, 0.0, 0.0);

          w_nTri = -1.0;

        }
      } else
        break;

      lSelf = false;
      if (testElement == sourceElement)
        lSelf = true;

      contribpm = 0.0;
      contribpm_add = (FTYPE2)(0.0, 0.0);
// Spatial
      lSelf_temp = lSelf;
      contribpm = BIContribG1(lSelf_temp, edge_L1I, node_PI[0],
          node_PI[1], node_PI[2], edge_L1J,
          node_PJ[0], node_PJ[1], node_PJ[2], kpm,
```

```
          csGamma.y, csGamma.x, kvec, debug, debug2);
      contribpm_add += contribpm;
// Spectral
      contribpm = BIContribG2(lSelf_temp, edge_L1I, node_PI[0],
          node_PI[1], node_PI[2], edge_L1J,
          node_PJ[0], node_PJ[1], node_PJ[2], kpm,
          csGamma.y, csGamma.x, kvec, debug, debug2);
      contribpm_add += kpm.x*kpm.x*contribpm;
      Zmn += w_mTri * w_nTri * cmult(contribpm_add,
                                     phaseshift(sourceEdge, testEdge, PhaseXFwd,
PhaseXBwd, PhaseYFwd, PhaseYBwd));

    }
  }
  return Zmn;
}

// OpenCL integration kernel for calculation of impedance matrix elements in BI
// Portion of PFEBI code.
// kpmp      - A pointer containing k and rho
//             kpm.x = k
//             kpm.y = Rho_a
//             kpm.z = Rho_b
// kvecp     - A pointer representing the k-vector
// greenType - Variable representing which Green's function to use since
//             OpenCL does not allow passing of functions by reference
//             0 - GreenFun2_Normal
//             1 - GreenFun2_Singularity
// csGammap  - csGamma.x = CosGamma
//             csGamma.y = SinGamma
__kernel void pfebi( __global FTYPE2 *Zdiag,
                     __global FTYPE2 *Zneare,
                     FTYPE4 kpm,
                     FTYPE4 kvec,
                     __global FTYPE2 *mTri,
                     __global FTYPE2 *pTri,
                     __global int2 *edge,
                     __global FTYPE *LenComm,
                     __global int *irowbag,
                     __global int *izcolloc,
                     __global int *pbc,
                     __global int *pmEle,
                     __global int *CommEdge,
                     FTYPE2 csGamma,
                     int N,
                     int M,
                     int nCom,
                     __global FTYPE *fpparams)
{
  int2 mn;
  int numEl = N;
  int maxDiff = M;
  int gid = get_global_id(0);
  int jid = get_global_id(1);
  int j;
  int m_temp;
  FTYPE2 PhaseXFwd = (FTYPE2)(fpparams[1], fpparams[2]);
  FTYPE2 PhaseXBwd = (FTYPE2)(fpparams[3], fpparams[4]);
  FTYPE2 PhaseYFwd = (FTYPE2)(fpparams[5], fpparams[6]);
  FTYPE2 PhaseYBwd = (FTYPE2)(fpparams[7], fpparams[8]);

  FTYPE2 Zmn = (FTYPE2) (0.0, 0.0);

  FTYPE4 zero4 = (FTYPE4) (0.0,0.0,0.0,0.0);
  FTYPE2 one2 = (FTYPE2) (1.0, 1.0);
  FTYPE2 cOne = (FTYPE2) (1.0, 0.0);
```

```
  if ( gid < numEl && jid < maxDiff) {
    mn.x = gid; // global id
    j = irowbag[gid] + jid;
    if (j < irowbag[gid+1]) {
      mn.y = izcolloc[j];
      Zmn = CalZBI_via_EdgetoEdge_AIM(kpm, mn, pTri, mTri, edge, LenComm,
                                pbc, pmEle, CommEdge, csGamma, nCom, kvec,
                                PhaseXFwd, PhaseXBwd, PhaseYFwd, PhaseYBwd, gid, mn.y );

      if (mn.x == mn.y) {
        if (mn.x < nCom) {
          m_temp = mn.x;

          Zdiag[m_temp] = Zmn;
        } else {
          m_temp = edge[mn.x - nCom].x;
          Zdiag[m_temp] += Zmn;
        }
      } // if (mn.x == mn.y)
      Zneare[j] = Zmn;
    } // for (j
  }  // if ( gid < numEl )
}
```

VITA

Jason Ashbach

- J. A. Ashbach, D.-H. Kwon, P. L. Werner, and D. H. Werner, "Low-Loss High-Q Optical Bandstop Filter Based on Chalcogenide Glass Grating Structures," Proceedings of the *2009 IEEE International Symposium on Antennas and Propagation and USNC/URSI National Radio Science Meeting*, Charleston, SC, USA, June 1-5, 2009.
- J. A. Ashbach, P. L. Werner, D. H. Werner, and F. Namin, "Single Material Alternative to a Multilayer Optical Window," Proceedings of the *2010 IEEE International Symposium on Antennas and Propagation and USNC/URSI National Radio Science Meeting*, Toronto, Canada, July 11-17, 2010.
- J. Ashbach, J. A. Bossard, X. Wang, and D. H. Werner, "Metamaterial Absorber for the Near-IR with Curvilinear Geometry based on Beziér Surfaces," Proceedings of the *2013 IEEE International Symposium on Antennas & Propagation and USNC/URSI National Radio Science Meeting*, July 7-13, 2013, Orlando, FL, USA.
- J. Ashbach, X. Wang, and D. H. Werner, "The Finite Element Boundary Integral Method Accelerated Using a Graphics Processing Unit," Proceedings of the *2013 IEEE International Symposium on Antennas & Propagation and USNC/URSI National Radio Science Meeting*, July 7-13, 2013, Orlando, FL, USA
- J. Ashbach and D. H. Werner, "An Oblique-Angle Infrared Circular Polarization Filter Using a Bezier Surface Representation," *2015 IEEE International Symposium on Antennas & Propagation and USNC/URSI National Radio Science Meeting*, July 19-24, 2015, Vancouver, BC, Canada.