User and Developer Interface Improvements To a Finite Difference Time Domain Code

A Thesis Presented

by

Seth Daniel Baum

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Electrical Engineering

in the field of

Electromagnetics

Northeastern University
Boston, Massachusetts, USA

August 2006

Abstract

For electromagnetics problems involving complicated source patterns, scattering objects, and inhomogeneous background media, analytical solutions to electromagnetic field solutions are usually not available. For these cases, numerical solutions are desired. Of the various numerical techniques available, the finite difference time domain (FDTD) method is emerging as the most popular technique due to its conceptual elegance and the completeness of the results it produces. Though FDTD has existed since it was first proposed by Yee in 1966 it has only risen to prominence in recent years as the computer industry has made it possible and cost-effective to handle FDTD's significant memory requirements.

Our group at Northeastern University uses a Fortran-based FDTD code developed and placed on the public domain by the Luebbers group at Pennsylvania State University. Since its adoption by our group, it has been used by several group members to solve various electromagnetics problems and has also been extended by several group members with various improvements to the underlying computational technique. However, the code's use and extension has been restricted by its steep learning curve which forces users to learn the Fortran programming language, the FDTD technique, and the inner workings of the code itself, and its use was restricted to a limited set of predefined cases.

Here, we present an enhancement of our code making it user- and developer-friendly. Now, code users need only a superficial understanding of Fortran and the FDTD technique, and rarely if ever need to modify any lines of the code to use it successfully.

Arbitrary material distributions can be designed external to the code and loaded into it. Code use is augmented by an attractive User Manual that guides users through using the code. Furthermore, the code itself is now written in an elegant and modular form, which makes it easier for developers to extend the code.

Acknowledgements

I would like to thank Carey Rappaport, my advisor, for providing inspiration and direction for this project; Charles DiMarzio and Michael Silevitch for serving on my thesis committee; Dan Mahr and Stephen Wong, who as high school students participating in Northeastern University's Young Scholars Program contributed significantly to the work presented here; my office neighbors on Stearns 3, for continuous ideas and feedback; the Center for Subsurface Sensing and Imaging Systems (National Science Foundation Award Number EEC-9986821) for bringing me into a larger research community; and Mohammed Farid, my primary collaborator, for always answering my endless questions about the code and in doing so being instrumental to my success.

I am grateful for the financial support I have received from the Department of Electrical and Computer Engineering, the School of Engineering Technology, and the National Science Foundation GK-12 Program (Award Number DGE 0338255), without which this work may not have been possible. I owe particular thanks to Claire Duggan for her endless work in creating education outreach opportunities.

Finally, I must thank all of my teachers, especially Mom, Dad, and even Ma8ra, for helping make me the student I am today.

Seth Baum

Boston, Massachusetts

August 9, 2006

Table of Contents

Chapter 1
Introduction

1.1 Introduction

For electromagnetics problems involving complicated source patterns, scattering objects, and inhomogeneous background media, analytical solutions to electromagnetic field distributions are usually not available. For these cases, numerical solutions are desired. Of the various numerical techniques available, the finite difference time domain (FDTD) method is emerging as the most popular technique due to its conceptual elegance and the completeness of the results it produces. Though FDTD has existed since first proposed by Yee in 1966 [Yee 1966], it has only risen to prominence in recent years as the computer industry has made it possible and cost-effective to handle FDTD's significant memory requirements [Taflove 2000].

Our group at Northeastern University uses a Fortran-based FDTD code developed and placed on public domain by the Luebbers group at Pennsylvania State University [Luebbers 1990, Luebbers 1991, Luebbers 1992, Hunsberger 1992]. Since its adoption by our group, it has been used by several group members to solve various electromagnetics problems [Farid 2002, Farid 2006] and also extended by several members with various improvements to the underlying computational technique [Rappaport 1997, Weedon 1997, Rappaport 1999, Talbot 2000, Rappaport 2003, Kosmas 2004, Jalalinia 2006].

The problems that our code has been used to solve are primarily in the area of subsurface imaging. This area of research has both biomedical and civil imaging

applications. Biomedical applications which our code has been used for include breast tumor detection. Civil applications which our code has been used for include ground penetrating radar for pollutant monitoring and landmine detection.

The particular code discussed here was adapted from a version of our code customized for the ground penetrating radar problem and retains several features useful to this problem. However, the code, like the FDTD technique in general, is sufficiently versatile to handle any sort of electromagnetics problem. The fact that it can handle both biomedical problems, which typically use optical frequencies, and civil problems, which typically use radio frequencies, exemplifies this versatility.

However, past work has failed to exploit the code's full potential because of the code's steep learning curve. To customize simulations, users would need to make changes buried deep in the code. Thus, users needed to learn the Fortran programming language, the FDTD technique, and the code itself. This requirement restricted our code's use to a small number of dedicated users at the PhD level or higher.

Furthermore, development of the underlying code was impeded by a confusing code design. This is an understandable consequence of its status as a legacy code developed mostly for personal work. Because it was developed mostly for personal work, developers had less reason to include features that would make the code easier for others to understand. The fact that the code is a legacy code, i.e. a code that has been passed repeatedly from one user to the next, exacerbated this problem because successive developers would not understand previous developers' work. This combination was a considerable impediment to the development of new code features.

Here, we present an enhanced version of our code making it both more user-friendly and more developer-friendly. Now, code users need only a superficial understanding of Fortran and the FDTD technique, and rarely if ever need to modify any lines of the code to use it successfully for a wide variety of cases. An attractive User Manual guides users through using the code. Furthermore, the code itself is now written in an elegant and modular form, which makes it easier for developers to extend the code.

1.2 Comparison to Existing FDTD Software

At the time of this writing, several other FDTD software packages exist, both commercial and academic. Our program has advantages and disadvantages over each.

Compared to commercial packages, the main advantages of our software package are that it's available for free and open source. Commercial FDTD software, such as Remcom's XFDTD program, costs in the tens of thousands of dollars, and is thus unaffordable to many researchers; ours can be obtained free of charge. Also, commercial software typically hides its source code for proprietary reasons. However, users may want to build in features not currently present in the software. They would be able to do this with our program because the source code is not hidden.

Commercial software does have several advantages over ours. In particular, the commercial offerings generally include vastly more sophisticated user interfaces, often with powerful graphical environments for designing material distributions and built-in means of viewing results. Despite the user interface improvements presented here, our program still uses a crude command window interface for entering simulation parameters,

has minimal automation of the material distribution design process, and relies on separate software such as MatLab for viewing results.

Another advantage of commercial software is that users never need to touch the source code. (Indeed, they generally don't even have access to it.) Due to Fortran's inability to dynamically size arrays, we were unable to set up the code so that any simulation could be run exclusively from external input (i.e. without any changes to code). In addition to making our program somewhat more difficult to use, this means that users who need to make these code changes must recompile the code, which prevents our program from being distributed exclusively as an executable file.

The advantages of our code over other academic programs are technical in nature. Our code, for example, has better handling for frequency dispersive materials, based on dispersion approximations developed by our group. It also features better handling of infinite half-spaces.

1.3 Thesis Organization

This thesis is divided into four chapters to describe the code in general and the specific improvements made to it.

Chapter 1 provides the theoretical basis of the FDTD technique. This includes general FDTD theory as well as extensions to it specific to our code.

Chapter 2 describes improvements to the user interface. These include both work helping the code's full potential get used and work making code use easier.

Chapter 3 describes improvements to the developer interface. These include changes that helped the development of the code presented here as well as changes that will help future development work.

Chapter 4 shows images from a variety of simulations run with the code.

Chapter 5 describes of future work that could further improve the code.

Chapter 2
Theoretical Basis for the Software

2.1 Introduction

In this chapter we explain how the FDTD technique works and provide an overview of how it is implemented in our code. We first give a brief explanation of how finite difference discretization works. Using this, we then derive the main FDTD equations from the well-known Maxwell equations. Afterwards, we offer a physical explanation of these FDTD equations. Then, we discuss specific details of our implementation of the FDTD technique, including our frequency dispersion model and our absorbing boundary. Finally, we compare the FDTD technique to other computational electromagnetics techniques.

2.2 Finite Difference Method

The finite difference method approximates derivatives with a straightforward "rise over run" approach. The simplicity of this approach makes FD considerably more intuitive and easier to learn than other, more complicated techniques and is one of FD's main attractions. In typical FD schemes, the first derivative (slope) at an arbitrary point on an arbitrary one-dimensional function f(x) is approximated with either a forward difference, backward difference, or single- or double-step central difference (Equations (2-1) through (2-7)).

Forward first difference:

$$\frac{\partial}{\partial x} f(x) \approx \frac{1}{\Delta}\big(f(x+\Delta) - f(x)\big) \tag{2-1}$$

Backward first difference:

$$\frac{\partial}{\partial x} f(x) \approx \frac{1}{\Delta}\big(f(x) - f(x-\Delta)\big) \tag{2-2}$$

Double-step central first difference:

$$\frac{\partial}{\partial x} f(x) \approx \frac{1}{2\Delta}\big(f(x+\Delta) - f(x-\Delta)\big) \tag{2-3}$$

Single-step central first difference:

$$\frac{\partial}{\partial x} f(x) \approx \frac{1}{\Delta}\left(f(x+\frac{\Delta}{2}) - f(x-\frac{\Delta}{2})\right) \tag{2-4}$$

Finite differences of second derivatives are derived by taking finite differences of finite differences (Equations (2-5) through (2-7)).

Forward second difference:

$$\frac{\partial^2}{\partial x^2} f(x) \approx \frac{1}{\Delta^2}\big(f(x+2\Delta) - 2f(x+\Delta) + f(x)\big) \tag{2-5}$$

Backward second difference:

$$\frac{\partial^2}{\partial x^2} f(x) \approx \frac{1}{\Delta^2}\big(f(x) - 2f(x-\Delta) + f(x-2\Delta)\big) \tag{2-6}$$

Central second difference:

$$\frac{\partial^2}{\partial x^2} f(x) \approx \frac{1}{\Delta^2}\big(f(x+\Delta) - 2f(x) + f(x-\Delta)\big) \tag{2-7}$$

Following Yee's convention, FDTD usually uses single-step central first differences and central second differences because these have less error and because their geometry is well suited for the Maxwell equations [Yee 1966].

2.2.1 Finite Difference Time Domain Discretization of the Maxwell Equations

In the finite difference time domain technique for electromagnetics, the Maxwell curl equations

$$\frac{\partial E}{\partial t} = \frac{1}{\varepsilon}\left(\nabla \times H\right) \tag{2-8}$$

$$\frac{\partial H}{\partial t} = -\frac{1}{\mu}\left(\nabla \times E\right) \tag{2-9}$$

E is the electric field intensity, measured in volts/meter. H is the magnetic field intensity, measured in amperes/meter. T is time, measured in seconds. ε is the complex permittivity defined by

$$\varepsilon = \varepsilon_{real} + \frac{\sigma}{i\omega} \tag{2-10}$$

ε_{real} is the real component of the complex permittivity, measured in farads per meter. σ is electrical conductivity, measured in siemens per meter. ω is the angular frequency, measured in radians per second. Finally, μ is magnetic permeability, measured in henries per meter.

In discretizing the Maxwell curl equations, they are first broken down into their directional components:

$$\frac{\partial E_x}{\partial t} = \frac{1}{\varepsilon}\left(\frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z}\right) \tag{2-11}$$

$$\frac{\partial E_y}{\partial t} = \frac{1}{\varepsilon}\left(\frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x}\right) \tag{2-12}$$

$$\frac{\partial E_z}{\partial t} = \frac{1}{\varepsilon}\left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y}\right) \tag{2-13}$$

$$\frac{\partial H_x}{\partial t} = \frac{1}{\mu}\left(\frac{\partial E_y}{\partial z} - \frac{\partial E_z}{\partial y}\right) \tag{2-14}$$

$$\frac{\partial H_y}{\partial t} = \frac{1}{\mu}\left(\frac{\partial E_z}{\partial x} - \frac{\partial E_x}{\partial z}\right) \tag{2-15}$$

$$\frac{\partial H_z}{\partial t} = \frac{1}{\mu}\left(\frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x}\right) \tag{2-16}$$

Then they are discretized using first and second differences. The continuous coordinates (x,y,z) are replaced by their discrete equivalents (i*dx,j*dy,k*dz), where dx, dy, and dz are the distances between successive points in the x, y, and z directions, respectively. The dx, dy, and dz terms are omitted here in order to keep the notation more concise.

Following the convention established by Yee, a single-step central difference is used:

$$\begin{aligned}&\frac{1}{\Delta t}\left(E_x(i+\frac{1}{2}, j, k, n+1) - E_x(i+\frac{1}{2}, j, k, n)\right) = \\ &\frac{1}{\varepsilon}\left(\begin{aligned}&\frac{1}{\Delta y}\left(H_z(i+\frac{1}{2}, j+\frac{1}{2}, k, n+\frac{1}{2}) - H_z(i+\frac{1}{2}, j-\frac{1}{2}, k, n+\frac{1}{2})\right) \\ &-\frac{1}{\Delta z}\left(H_y(i+\frac{1}{2}, j, k+\frac{1}{2}, n+\frac{1}{2}) - H_y(i+\frac{1}{2}, j, k-\frac{1}{2}, n+\frac{1}{2})\right)\end{aligned}\right)\end{aligned} \tag{2-17}$$

$$\frac{1}{\Delta t}\left(E_y(i,j+\frac{1}{2},k,n+1)-E_y(i,j+\frac{1}{2},k,n)\right)=$$
$$\frac{1}{\varepsilon}\left(\begin{array}{l}\frac{1}{\Delta z}\left(H_x(i,j+\frac{1}{2},k+\frac{1}{2},n+\frac{1}{2})-H_x(i,j+\frac{1}{2},k-\frac{1}{2},n+\frac{1}{2})\right)\\ -\frac{1}{\Delta x}\left(H_z(i+\frac{1}{2},j+\frac{1}{2},k,n+\frac{1}{2})-H_z(i-\frac{1}{2},j+\frac{1}{2},k,n+\frac{1}{2})\right)\end{array}\right) \qquad (2\text{-}18)$$

$$\frac{1}{\Delta t}\left(E_z(i,j,k+\frac{1}{2},n+1)-E_z(i,j,k+\frac{1}{2},n)\right)=$$
$$\frac{1}{\varepsilon}\left(\begin{array}{l}\frac{1}{\Delta x}\left(H_y(i+\frac{1}{2},j,k+\frac{1}{2},n+\frac{1}{2})-H_y(i-\frac{1}{2},j,k+\frac{1}{2},n+\frac{1}{2})\right)\\ -\frac{1}{\Delta y}\left(H_x(i,j+\frac{1}{2},k+\frac{1}{2},n+\frac{1}{2})-H_x(i,j-\frac{1}{2},k+\frac{1}{2},n+\frac{1}{2})\right)\end{array}\right) \qquad (2\text{-}19)$$

$$\frac{1}{\Delta t}\left(H_x(i,j+\frac{1}{2},k+\frac{1}{2},n+\frac{1}{2})-H_x(i,j+\frac{1}{2},k+\frac{1}{2},n-\frac{1}{2})\right)=$$
$$\frac{1}{\mu}\left(\begin{array}{l}\frac{1}{\Delta z}\left(E_y(i,j+\frac{1}{2},k+1,n)-E_y(i,j+\frac{1}{2},k,n)\right)\\ -\frac{1}{\Delta y}\left(E_z(i,j+1,k+\frac{1}{2},n)-E_z(i,j,k+\frac{1}{2},n)\right)\end{array}\right) \qquad (2\text{-}20)$$

$$\frac{1}{\Delta t}\left(H_y(i+\frac{1}{2},j,k+\frac{1}{2},n+\frac{1}{2})-H_y(i+\frac{1}{2},j,k+\frac{1}{2},n-\frac{1}{2})\right)=$$
$$\frac{1}{\mu}\left(\begin{array}{l}\frac{1}{\Delta x}\left(E_z(i+1,j,k+\frac{1}{2},n)-E_z(i,j,k+\frac{1}{2},n)\right)\\ -\frac{1}{\Delta z}\left(E_x(i+\frac{1}{2},j,k+1,n)-E_x(i+\frac{1}{2},j,k,n)\right)\end{array}\right) \qquad (2\text{-}21)$$

$$\frac{1}{\Delta t}\left(H_z(i+\frac{1}{2},j+\frac{1}{2},k,n+\frac{1}{2})-H_z(i+\frac{1}{2},j+\frac{1}{2},k,n-\frac{1}{2})\right)=$$
$$\frac{1}{\mu}\left(\begin{array}{l}\frac{1}{\Delta y}\left(E_x(i+\frac{1}{2},j+1,k,n)-E_x(i+\frac{1}{2},j,k,n)\right)\\ -\frac{1}{\Delta x}\left(E_y(i+1,j+\frac{1}{2},k,n)-E_y(i,j+\frac{1}{2},k,n)\right)\end{array}\right) \quad (2\text{-}22)$$

Then, in each equation, the later field component in the time difference is solved for, giving the FDTD update equations:

$$E_x(i+\frac{1}{2},j,k,n+1)=E_x(i+\frac{1}{2},j,k,n)$$
$$+\frac{\Delta t}{\varepsilon}\left(\begin{array}{l}\frac{1}{\Delta y}\left(H_z(i+\frac{1}{2},j+\frac{1}{2},k,n+\frac{1}{2})-H_z(i+\frac{1}{2},j-\frac{1}{2},k,n+\frac{1}{2})\right)\\ -\frac{1}{\Delta z}\left(H_y(i+\frac{1}{2},j,k+\frac{1}{2},n+\frac{1}{2})-H_y(i+\frac{1}{2},j,k-\frac{1}{2},n+\frac{1}{2})\right)\end{array}\right) \quad (2\text{-}23)$$

$$E_y(i,j+\frac{1}{2},k,n+1)=E_y(i,j+\frac{1}{2},k,n)$$
$$+\frac{\Delta t}{\varepsilon}\left(\begin{array}{l}\frac{1}{\Delta z}\left(H_x(i,j+\frac{1}{2},k+\frac{1}{2},n+\frac{1}{2})-H_x(i,j+\frac{1}{2},k-\frac{1}{2},n+\frac{1}{2})\right)\\ -\frac{1}{\Delta x}\left(H_z(i+\frac{1}{2},j+\frac{1}{2},k,n+\frac{1}{2})-H_z(i-\frac{1}{2},j+\frac{1}{2},k,n+\frac{1}{2})\right)\end{array}\right) \quad (2\text{-}24)$$

$$E_z(i,j,k+\frac{1}{2},n+1)=E_z(i,j,k+\frac{1}{2},n)$$
$$+\frac{\Delta t}{\varepsilon}\left(\begin{array}{l}\frac{1}{\Delta x}\left(H_y(i+\frac{1}{2},j,k+\frac{1}{2},n+\frac{1}{2})-H_y(i-\frac{1}{2},j,k+\frac{1}{2},n+\frac{1}{2})\right)\\ -\frac{1}{\Delta y}\left(H_x(i,j+\frac{1}{2},k+\frac{1}{2},n+\frac{1}{2})-H_x(i,j-\frac{1}{2},k+\frac{1}{2},n+\frac{1}{2})\right)\end{array}\right) \quad (2\text{-}25)$$

$$H_x(i, j+\frac{1}{2}, k+\frac{1}{2}, n+\frac{1}{2}) = H_x(i, j+\frac{1}{2}, k+\frac{1}{2}, n-\frac{1}{2}) + \frac{\Delta t}{\mu}\left(\begin{array}{l} \frac{1}{\Delta z}\left(E_y(i, j+\frac{1}{2}, k+1, n) - E_y(i, j+\frac{1}{2}, k, n)\right) \\ -\frac{1}{\Delta y}\left(E_z(i, j+1, k+\frac{1}{2}, n) - E_z(i, j, k+\frac{1}{2}, n)\right) \end{array}\right) \quad (2\text{-}26)$$

$$H_y(i+\frac{1}{2}, j, k+\frac{1}{2}, n+\frac{1}{2}) = H_y(i+\frac{1}{2}, j, k+\frac{1}{2}, n-\frac{1}{2}) + \frac{\Delta t}{\mu}\left(\begin{array}{l} \frac{1}{\Delta x}\left(E_z(i+1, j, k+\frac{1}{2}, n) - E_z(i, j, k+\frac{1}{2}, n)\right) \\ -\frac{1}{\Delta z}\left(E_x(i+\frac{1}{2}, j, k+1, n) - E_x(i+\frac{1}{2}, j, k, n)\right) \end{array}\right) \quad (2\text{-}27)$$

$$H_z(i+\frac{1}{2}, j+\frac{1}{2}, k, n+\frac{1}{2}) = H_z(i+\frac{1}{2}, j+\frac{1}{2}, k, n-\frac{1}{2}) + \frac{\Delta t}{\mu}\left(\begin{array}{l} \frac{1}{\Delta y}\left(E_x(i+\frac{1}{2}, j+1, k, n) - E_x(i+\frac{1}{2}, j, k, n)\right) \\ -\frac{1}{\Delta x}\left(E_y(i+1, j+\frac{1}{2}, k, n) - E_y(i, j+\frac{1}{2}, k, n)\right) \end{array}\right) \quad (2\text{-}28)$$

Using single-step central differences results in the prevalence of half-step coordinates. This complicates FDTD's implementation in code because programming languages, including the Fortran used here, generally don't permit non-integer matrix indices. Thus, a scheme for converting half-step coordinates into integer coordinates is required. In our code, we use the typical approach of subtracting ½ from all half-step coordinates. This is analogous to performing a series of half-step coordinate shifts.

To calculate a value for a field component at a given time step using this formulation, only values from the previous time step and half time step will be needed. For example, calculating a value for Ex at time = 11 requires a value for Ex at time = 10 and values for Hy and Hz at time = 10½. Thus, electric and magnetic field values for two

time steps each must be stored in computer memory; values at all previous time steps can be discarded. The frequency dispersion model used in the work presented here requires keeping in memory four total time steps for the electric field and two for the magnetic field.

In our code, each required time step exists in a separate matrix. This allows us to drop the time dimension matrix index and thus not perform a coordinate shift on it. This gives the shifted update equations:

$$E_x(i,j,k) = E_x(i,j,k) + \frac{\Delta t}{\varepsilon}\left(\begin{array}{l}\frac{1}{\Delta y}\left(H_z(i,j,k) - H_z(i,j-1,k)\right) \\ -\frac{1}{\Delta z}\left(H_y(i,j,k) - H_y(i,j,k-1)\right)\end{array}\right) \tag{2-29}$$

$$E_y(i,j,k) = E_y(i,j,k) + \frac{\Delta t}{\varepsilon}\left(\begin{array}{l}\frac{1}{\Delta z}\left(H_x(i,j,k) - H_x(i,j,k-1)\right) \\ -\frac{1}{\Delta x}\left(H_z(i,j,k) - H_z(i-1,j,k)\right)\end{array}\right) \tag{2-30}$$

$$E_z(i,j,k) = E_z(i,j,k) + \frac{\Delta t}{\varepsilon}\left(\begin{array}{l}\frac{1}{\Delta x}\left(H_y(i,j,k) - H_y(i-1,j,k)\right) \\ -\frac{1}{\Delta y}\left(H_x(i,j,k) - H_x(i,j-1,k)\right)\end{array}\right) \tag{2-31}$$

$$H_x(i,j,k) = H_x(i,j,k) + \frac{\Delta t}{\mu}\left(\begin{array}{l}\frac{1}{\Delta z}\left(E_y(i,j,k+1) - E_y(i,j,k)\right) \\ -\frac{1}{\Delta y}\left(E_z(i,j+1,k) - E_z(i,j,k)\right)\end{array}\right) \tag{2-32}$$

$$H_y(i,j,k) = H_y(i,j,k) + \frac{\Delta t}{\mu}\left(\begin{array}{l}\frac{1}{\Delta x}\left(E_z(i+1,j,k) - E_z(i,j,k)\right) \\ -\frac{1}{\Delta z}\left(E_x(i,j,k+1) - E_x(i,j,k)\right)\end{array}\right) \quad (2\text{-}33)$$

$$H_z(i,j,k) = H_z(i,j,k) + \frac{\Delta t}{\mu}\left(\begin{array}{l}\frac{1}{\Delta y}\left(E_x(i,j+1,k) - E_x(i,j,k)\right) \\ -\frac{1}{\Delta x}\left(E_y(i+1,j,k) - E_y(i,j,k)\right)\end{array}\right) \quad (2\text{-}34)$$

These equations are known as the FDTD update equations because they're used to update the field component values in simulations: The previous value and the other terms update the next value.

An FDTD simulation begins at some initial point in time with some known initial field distribution (generally zero). At the next time step, field component values at each point in space are calculated using the update equations. If the initial distribution is all zero, an excitation source is applied over some points in time so that not all field component values are zero. Time thus moves with each successive iteration of update equation calculations, with fields taking form based on initial fields, excitations, and material distributions.

2.2.2 Physical Interpretation of Field Component and Material Locations

The half-step shifts required by the single-step central differences complicate the physical interpretation of field component and material locations.

The “Yee cube”, introduced in Yee's original paper on FDTD [Yee 1966], offers an effective description of the placement of the field components. In it, field components paired together from finite differences are placed along a cube. In the traditional discretization scheme shown in **Figure 1**, electric field components lie along the cube's edges and magnetic field components lie along the cube's faces.

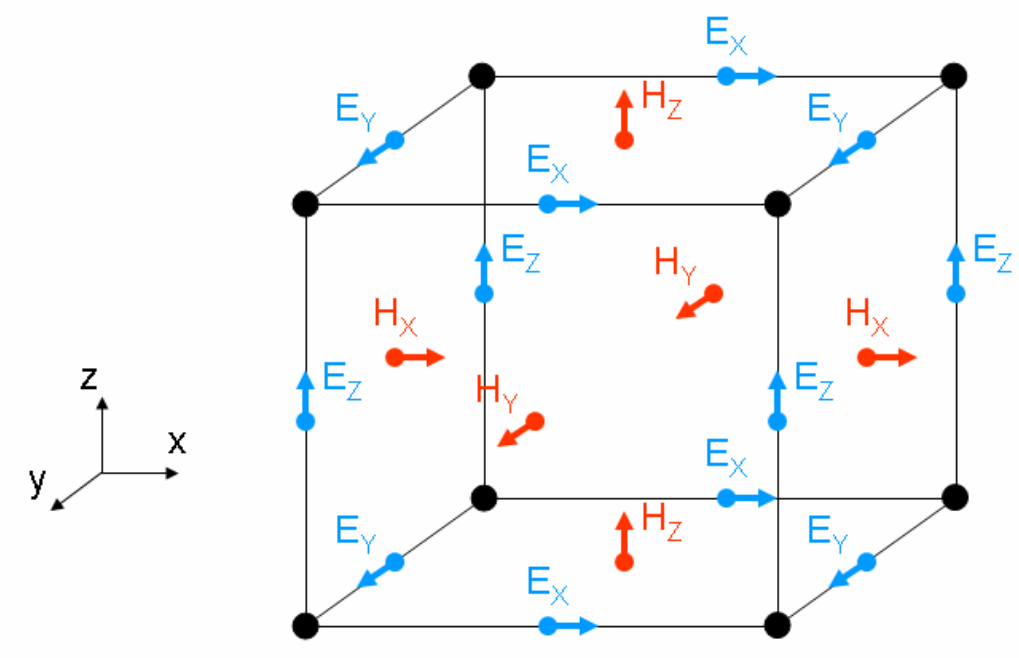


Figure 1: Yee cube

While the Yee cube effectively captures the locations of field components, it leaves ambiguous the placement of material points. Indeed, the FDTD technique in general can support a variety of interpretations of material point locations. In general, there are two typical types of interpretations. The first is for material points to represent the material value at a specific point within the Yee cube or along its outer boundary. The second is for material points to represent an average or predominant material within the Yee cube.

The question of what material points represent is only important when material properties change significantly within distances comparable to grid cell dimensions. For slowly-varying material distributions, any choice of material point systems will bring approximately the same results. However, for rapidly-varying material distributions, this choice can significantly affect results, and care needs to be taken in the process of building material distribution.

Taflove and Hagness suggest creating a separate material variable for each field component [Taflove 2000]. While this approach may be more accurate, it also requires six times more memory per grid cell than a system in which a cluster of the six field components share a material point. For our purposes, the memory cost of this approach is excessive whereas the accuracy gain is negligible. We thus have only one material point per Yee cube. This approach permits the material point to be placed anywhere in the Yee cube. In our formulation, this point is on the corner closest to the origin.

When material points are not collocated with field components, as is this case with our model, a system is required for connecting the two. Clearly, the choice of system only matters when the surrounding materials' properties are not the same. For our code, we have chosen to place material points at integer coordinates and link field components to the material point one half-step closer to the origin, as shown in **Figure 2**.

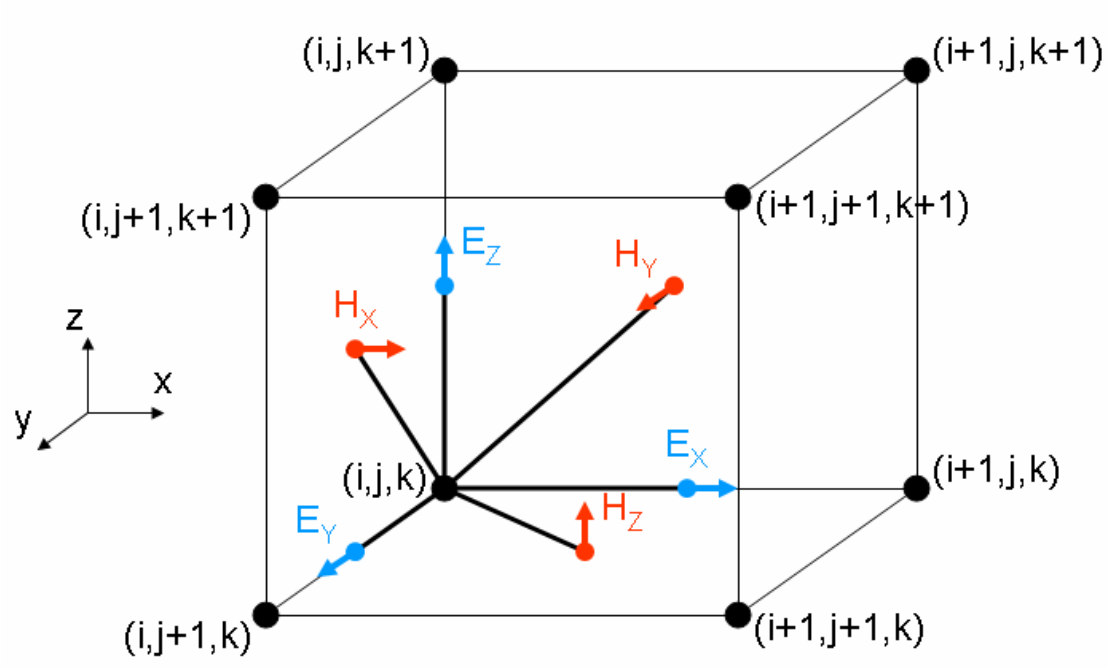


Figure 2: Yee cube with material point-field component links

2.3 Frequency Dispersion Approximation

The Northeastern code uses a model of dispersive media such as soil that represents an improvement in terms of accuracy and memory requirement over previous techniques such as the Debye and Lorentz models [Sullivan 1992, Sullivan 1995, Young 1995, Rappaport 1997, Weedon 1997, Rappaport 1999, Talbot 2000, Rappaport 2003, Kosmas 2004, Jalalinia 2006]. The model is based on a Z-transform conductivity approximation, in which the right hand side of the Maxwell equation $\nabla \times H = \frac{\partial D}{\partial t} + J$ is Z-transformed to

$$\frac{1 - Z^{-1}}{\Delta t} \varepsilon_0 \varepsilon_{Av} E + \sigma(Z) E \tag{2-35}$$

using

$$Z = e^{i\omega\Delta t} \tag{2-36}$$

Δt is the time step size, measured in seconds. ε_0 is the electric permittivity of free space, measured in farads per meter. ε_{Av} is the average relative electric permittivity, which is frequency-independent. ε_{Av} is unitless. σ is electrical conductivity, measured in siemens per meter.

The conductivity is approximated using a single pole, two zero model:

$$\sigma(Z) = \frac{J(Z)}{E(Z)} = \frac{b_0 + b_1 Z^{-1} + b_2 Z^{-2}}{1 + a_1 Z^{-1}} \tag{2-37}$$

This conductivity model gives the update equations:

$$E_x = \frac{1}{1+e_0}(-e_1 E_x(n) - e_2 E_x(n-1) - e_3 E_x(n-2) + \Delta H_x) \tag{2-38}$$

$$E_y = \frac{1}{1+e_0}(-e_1 E_y(n) - e_2 E_y(n-1) - e_3 E_y(n-2) + \Delta H_y) \tag{2-39}$$

$$E_z = \frac{1}{1+e_0}(-e_1 E_z(n) - e_2 E_z(n-1) - e_3 E_z(n-2) + \Delta H_z) \tag{2-40}$$

$$\Delta H_x = \frac{\Delta t}{\varepsilon_{Av}}\left(-\frac{\Delta H_y}{\Delta z}\left(n+\frac{1}{2}\right) - a_1 \frac{\Delta Hy}{\Delta z}\left(n-\frac{1}{2}\right) + \frac{\Delta H_z}{\Delta y}\left(n+\frac{1}{2}\right) + a_1 \frac{\Delta H_z}{\Delta y}\left(n-\frac{1}{2}\right)\right) \tag{2-41}$$

$$\Delta H_y = \frac{\Delta t}{\varepsilon_{Av}}\left(-\frac{\Delta Hz}{\Delta x}\left(n+\frac{1}{2}\right) - a_1 \frac{\Delta Hz}{\Delta x}\left(n-\frac{1}{2}\right) + \frac{\Delta Hx}{\Delta z}\left(n+\frac{1}{2}\right) + a_1 \frac{\Delta Hx}{\Delta z}\left(n-\frac{1}{2}\right)\right) \tag{2-42}$$

$$\Delta H_z = \frac{\Delta t}{\varepsilon_{Av}}\left(-\frac{\Delta H_x}{\Delta y}\left(n+\frac{1}{2}\right) - a_1 \frac{\Delta H_x}{\Delta y}\left(n-\frac{1}{2}\right) + \frac{\Delta H_y}{\Delta x}\left(n+\frac{1}{2}\right) + a_1 \frac{\Delta H_y}{\Delta x}\left(n-\frac{1}{2}\right)\right) \tag{2-43}$$

$$e_0 = 1 + b_0 \frac{\Delta t}{2\varepsilon_{Av}} \tag{2-44}$$

$$e_1 = a_1 - 1 + (b_0 + b_1)\frac{\Delta t}{2\varepsilon_{Av}} \tag{2-45}$$

$$e_2 = -a_1 + (b_1 + b_2)\frac{\Delta t}{2\varepsilon_{Av}} \tag{2-46}$$

$$e_3 = b_2 \frac{\Delta t}{2\varepsilon_{Av}} \tag{2-47}$$

Here, the spatial coordinates have been omitted and the spatial differences written in the form $\frac{\Delta H_x}{\Delta y}$ to keep the notation concise.

2.4 Absorbing Boundary Conditions

The Northeastern code uses a version of the first-order Mur absorbing boundary that includes the frequency dispersion approximation described above. It comes from [Kosmas 2004]. Beginning with a first-order approximation of the frequency-domain one-way wave equation (Equation 4 in Kosmas 2004):

$$\left(\frac{\partial}{\partial_x} - i\omega\sqrt{\mu_0\varepsilon_0\varepsilon^{'}(\omega)}\left(1 - \frac{i\sigma(\omega)}{\omega\varepsilon_0\varepsilon^{'}(\omega)}\right)^{1/2}\right)W(\omega)_{x=0} = 0 \tag{2-48}$$

Our formulation uses the approximation $\sqrt{1-x} \approx 1 - x/2$ for the conductivity term $\left(1 - \frac{i\sigma(\omega)}{\omega\varepsilon_0\varepsilon^{'}(\omega)}\right)^{1/2}$. The ½ factor is folded into the variables b_0, b_1, and b_2 in the conductivity approximation. Thus, in the time domain version of Equation (2-48) (Equation 6 in Kosmas 2004)

$$\left\{\left(\frac{\partial}{\partial_x} - \frac{e}{c_0}\frac{\partial}{\partial_t}\right)(W^n + \alpha_1 W^{n-1}) - \eta\left(\beta_0 W^n + \beta_1 W^{n-1} + \beta_2 W^{n-2}\right)\right\}_{x=0} = 0 \tag{2-49}$$

the following definitions are used:

$$e = \sqrt{\varepsilon'} \tag{2-50}$$

$$\alpha_1 = a_1 \tag{2-51}$$

$$\beta_0 = \frac{b_0}{2} = \frac{\sigma}{2\sqrt{\varepsilon'}} \tag{2-52}$$

$$\beta_1 = \frac{b_1}{2} \tag{2-53}$$

$$\beta_2 = \frac{b_2}{2} \tag{2-54}$$

As with the update equations, our absorbing boundary formulation uses averages to obtain more robust values. Where a spatial difference is taken ($\frac{\partial}{\partial_x}$ in Equation (2-49)), a temporal average is taken, resulting in averaged field components of the form

$$W^n = \frac{1}{2}\left(E(i, j, k, n) - E(i, j, k, n-1)\right) \tag{2-55}$$

Similarly, where a temporal difference is taken ($\frac{\partial}{\partial_t}$ in Equation (2-49)), a spatial average is taken, resulting in averaged field components of the form

$$W^n = \frac{1}{2}\left(E(i, j, k, n) - E(i-1, j, k, n)\right) \tag{2-56}$$

Spatial averages are also used where no difference is taken ($\eta\left(\beta_0 W^n + \beta_1 W^{n-1} + \beta_2 W^{n-2}\right)$ in Equation (2-49)).

This gives absorbing boundary equations in the form:

(2-57)

$$
\begin{aligned}
0 = & \\
& \frac{1}{\Delta x}\left(E_y(2,\, j,k,n) - E_y(1,\, j,k,n) + E_y(2,\, j,k,n-1) - E_y(1,\, j,k,n-1)\right) \\
& + \frac{a_1}{\Delta x}\left(E_y(2,\, j,k,n-1) - E_y(1,\, j,k,n-1) + E_y(2,\, j,k,n-2) - E_y(1,\, j,k,n-2)\right) \\
& - \frac{1}{c\Delta t}\left(E_y(2,\, j,k,n) - E_y(2,\, j,k,n-1) + E_y(1,\, j,k,n) - E_y(1,\, j,k,n-1)\right) \\
& - \frac{a_1}{c\Delta t}\left(E_y(2,\, j,k,n-1) - E_y(2,\, j,k,n-2) + E_y(1,\, j,k,n-1) - E_y(1,\, j,k,n-2)\right) \\
& - \frac{\eta b_0}{2}\left(E_y(2,\, j,k,n-1) + E_y(1,\, j,k,n-1)\right) \\
& - \frac{\eta b_1}{2}\left(E_y(2,\, j,k,n-2) + E_y(1,\, j,k,n-2)\right) \\
& - \frac{\eta b_3}{2}\left(E_y(2,\, j,k,n-3) + E_y(1,\, j,k,n-3)\right)
\end{aligned}
$$

In the code, this equation is solved for $E_y(1,j,k,n)$.

Chapter 3:
User Interface Improvements

3.1 Introduction

A simulation is only as good as what it will accomplish. Relative to other techniques, the FDTD method is both computationally powerful and conceptually simple. Its main limitation is its large computer memory requirement. However, specific implementations of the FDTD method vary in how computationally powerful, simple to handle, and memory-intensive they are. These three factors affect how easily and effectively the code can be used.

Our group's original code included several features making it more powerful and easier to use, especially for simulating the ground penetrating radar problem. Features making it more powerful included an improved frequency dispersion model. Features making it easier to use included a built-in system for generating soil layers and for building objects including monopole antennae and landmines.

These features made the code an attractive option for use. However, other aspects of the code make it less attractive. Excessive customization for specific cases prevented users from exploiting the full power of the FDTD technique. A complicated and poorly documented system of defining simulations made it difficult to run even simple simulations. Memory requirements were unnecessarily large, limiting the size of simulations the code was able to perform on a given computer. Finally, a handful of errors would lead to incorrect results in certain cases.

The net result of these drawbacks was that the simulation was only used by a small handful of researchers, each of whom was heavily invested in developing the code itself. Other researchers who were interested in using the code for simulations but were unable to invest so much time and energy to learn the code simply didn't use it.

A core goal of the work presented here was to reverse this. By making the code more versatile, easier to initialize, more computationally efficient, and more error-free, we have produced a code that is both accessible to a much wider range of users and easier to use for all users, from the most novice to the most sophisticated.

3.2 Making the Code More Versatile

The FDTD method provides a numerical solution to the forward version of Maxwell's equations. It takes a known material distribution and excitation and calculates the corresponding electromagnetic field distribution. Thus, in the code, the material distribution and excitation must be defined by the user for the simulation and the field distribution must be outputted from the simulation to the user.

In order to exploit the full power of the FDTD technique, the code must be versatile enough to enable the user to easily and arbitrarily define material distributions and excitations and view arbitrary portions of the resulting field distribution. Our code contains improvements in each of these three areas, making it much more versatile than its predecessor.

3.2.1 More Control Over Material Distribution

Unlike other discretization techniques, the finite difference method samples points within the computational domain along a uniform grid. While this is the root of the FDTD method's large memory requirement, it also permits complicated material distributions to be easily simulated. However, in order to take advantage of this property, the implementing code must permit the user to develop complex material distributions.

The original code offered some flexibility in developing material distributions, particularly those of interest to select ground penetrating radar problems. For example, the code featured a built-in library of material types, including free space, perfect electric conductor, a non-dispersive dielectric material, and various types of soils. The code also implemented a system of building material distributions using background layers and foreground objects. Background layers are homogenous rectangular prisms that span the entire horizontal (x-y) extent of the computational domain. They can be used, for example, for building a computational domain which includes a flat air-soil interface. Foreground objects are material distributions of various shapes that are placed “on top of” background layers, i.e. their material values overwrite the material values of the collocated background layer points. These shapes include the homogenous rectangular prism, the homogenous sphere, the homogenous cylinder, and a monopole antenna consisting of perfect electric conductor and non-dispersive dielectric.

Several improvements for building material distributions have been made to the new code to greatly improve options for building material distributions. These improvements include a new option for user-defined materials, improvements to the original code's schemes for developing background layers and foreground objects as well as the introduction of a powerful new type of object, the material input file. Finally,

changes to the absorbing boundaries made this part of the code more flexible with respect to material distributions.

3.2.1.1 New Option For User-Defined Materials

In an FDTD simulation, a material is an abstract entity defined by certain properties relevant to the simulation. These properties include permittivity, conductivity, and dispersion parameters. Thus, a material is effectively a set of numbers corresponding to these properties, along with a label used for identification purposes. It is through these properties that the material interacts with the rest of the code.

In the original code, the user was limited to a set of about 20 built-in materials. These materials were all materials relevant to the ground penetrating radar cases of interest to the research group. They included free space, metal, non-dispersive and dispersive dielectrics, and various dispersive soils [Hipp 1974, Curtis 1996]. Users interested in simulating cases involving materials not already built into the code would need to go into the code itself and build new materials from within, a lengthy process requiring extensive knowledge of the code.

In the final code, the option exists to build new materials through the external user interface. The user simply states how many new materials to build (including 0 if no new materials are desired), and then for each new material, inputs values for the material's properties. By doing this, the user gains complete control over the materials available for the simulation.

Because these user-defined materials are inputted via the command window instead of via changes to the code itself, the code will not remember materials from one

simulation to the next. However, a user wishing to use materials in multiple simulations can simply input the same exact answers to the corresponding questions. (See Section 3.3.)

3.2.1.2 Background Layers Improvements

The first step in the code's process of developing a material distribution is the built-in background layer system. The final code extends the original code's scheme, which was designed for the ground penetrating radar problem. In this scheme, horizontal layers of material are built from the bottom of the computational domain to the top. This mimics the horizontal layering common to actual soil deposits and to the soil-air interface at the surface of the ground.

The original code was designed specifically for simulating ground penetrating radar problems and thus required all layers to be composed of one of the built-in soil types. While it included an option for a background consisting of both free space and soil, it required the presence of at least some soil in the background medium.

The new code, however, striving to be more generally applicable, permits any combination of materials. While this system does allow the user to create physically unrealistic backgrounds, such as a layer of free space between two layers of soil, it enables the user to design whatever background layers are desired.

Another set of improvements to the background layer system that the final code makes is the handling of the number and thicknesses of layers. In the original code, an arbitrary number of layers was allowable, regardless of how many grid cells had been allocated for soil layers. Thus, users were allowed to create more soil layers than there

were grid cells available to construct them. Furthermore, individual layers could have any thickness, so users could potentially run out of grid cells for remaining layers. Also, the sum of soil layer thicknesses were not required to add up to the total thickness of soil. Finally, if the a geometry called for a layer of air to be placed above soil layers, there was no check to make sure enough room for air was left.

The final code addresses all of these glitches. First, the distinction between soil and air layers is removed, leaving a system in which all layers are handled the same, regardless of their composition, thereby streamlining the simulation considerably. The number of layers is now limited to the number grid cells are available. The maximum allowable thickness for a layer is defined as the thickness that would leave all remaining layers with the minimum possible thickness (one grid cell). The thickness of the top layer is automatically defined as the distance from the top of the previous layer to the highest available grid cell. This guarantees that the sum of layer thicknesses will equal the total thickness of layers.

3.2.1.3 Foreground Object Improvements

After the background layers are built, users can add foreground objects, including rectangular prisms, cylinders, spheres, monopole antennae, and material input files.

Except for material input files, the above-mentioned shapes were all present in the original code. However, they required some adjustment to get them to work properly. For example, in the original code for building rectangular prisms, when the shape's width in the x-direction was entered as either 2, 10, or 25 grid cells, the code would actually build a cube whose width in each direction was one grid cell.

The original code also included a landmine object and a dipole antenna object. The dipole antenna foreground object was found not to be a reasonable representation of an actual dipole antennae and was removed. The landmine object was removed as a separate option because landmines can be reasonably approximated with a cylinder. The final code notes this in its user interface.

One shortcoming of the original code's scheme for inserting objects was that only one of each available object could be built. Thus, a user wishing to simulate a problem with, for example, two monopole antennae present, was unable to do so without rewriting the code. The final code addresses this shortcoming by replacing the original code's series of yes/no questions for each object type with the question of how many objects of any kind the user would like to build and then permitting the user to build any of the available objects for each of these.

As with the original code, if two foreground objects overlap, the material values for the object built later overwrite those of the object built earlier. In the original code, the sequence of building objects was fixed, so, for example, a sphere could not overwrite a monopole antenna. The new code allows users to build objects in any order, letting the user take advantage of the simulation’s material value overwriting system.

3.2.1.4 The New Material Input File Object

Users of the original code were restricted to building material distributions composed of background layers and the handful of built-in foreground objects. This restricted them to simulating only a small range of electromagnetics cases. The final code's improvements to the background layer and foreground object systems bring greater flexibility in

designing material distributions, but still limit the user to combinations of available shapes. While users wishing total control over material distributions can use combinations of cubes of single grid cell width to define materials point by point, this process is tedious and prone to error.

To make the process of totally controlling material distributions more efficient, the final code features a new object type called the material input file. A material input file is a text file containing an arbitrary two-dimensional material distribution. In the final code, such files can be read into a simulation; the file's material distribution becomes the material distribution of a corresponding two-dimensional region of the computational domain. The files can be oriented perpendicular to either the x, y, or z axes. They can be as small as one grid point or as large as a two-dimensional slice of the entire computational grid. This gives users complete control over material distribution.

3.2.2 More Flexible System For Absorbing Boundary Equation Selection

The original code contained two separate sets of absorbing boundary equations: one for free space and one for other materials. The free space absorbing boundary was used on the top free space layer (if there was one); the other absorbing boundary was used for all other points.

This system works well as long as no other materials are placed along the boundaries within the free space region. However, this is an unreasonable expectation for future users, who may want to model rough soil layers or place objects of various materials along the boundaries of the free space region. Free space placed along the

boundaries within the rest of the computational domain was not a problem because the other absorbing boundary worked properly with free space.

To address this issue, the new code simply removes the free space absorbing boundary and uses the other absorbing boundary for all materials. Thus, users can build arbitrary material distributions without worrying about whether or not the absorbing boundary will function properly.

3.2.3 More Flexible Excitation System

In addition to the material distribution, the user must also define the simulation's excitation. The excitation, or source, is essentially an electric field to be added to the simulation so that the resulting electromagnetic field values are not always zero. Without an excitation, there would be nothing to simulate.

An excitation has several attributes. It can be either a “hard source” or a “soft source”, depending on how it is added to the simulation. It also has a location, a direction, and a strength, the latter of which typically varies in time. This variation of excitation strength with time is called the pulse shape. Pulses are generally used in FDTD because the initial field strength is generally zero, as is the case in this code.

The original code featured excitations specific to the ground penetrating radar cases it was used to simulate. However, as the new code strives to be used more broadly, it includes several extensions of the original code's excitation system permitting either hard or soft sources, more control over the location and direction of excitations, and new pulse shapes.

3.2.3.1 Option For Hard Or Soft Sources

The two common types of excitation sources, “hard sources” and “soft sources”, correspond to the two common ways of adding the source's electric field to a simulation. In a hard source, the electric field values at the excitation points are fixed to the source's values. In a soft source, the excitation points' field values are the sum of the source field values and the field values already present at the points.

The original code did not offer the user a choice between hard or soft sources. Instead, it used a hard source for one pulse shape and a soft source for its other pulse shape, presumably based on the needs of its previous user. The new code gives the user the option of choosing either a hard source or a soft source regardless of what the pulse shape is.

3.2.3.2 Improved Control Over Excitation Location And Direction

Once it is determined how the excitation's electric fields will be added to the simulation, it must be determined where the fields will be added and in what direction the fields will be oriented.

In the original code, it was assumed that the excitation would be attached to one of the built in objects, either a monopole antenna or a dipole antenna. However, the new code does not make this assumption. Indeed, since one important application of the new code is to simulate arbitrarily shaped antennae, it is important for the user to be able to create arbitrarily located and oriented excitations.

Thus, while the new code kept the original code's system for exciting monopole antennae, it added a system for user-defined excitation location and direction. (The system for exciting dipole antennae was removed when the dipole antenna itself was removed due to flaws in its design.) The user is first asked how many excitation points are desired. Then, the user can define the location and orientation of each point. The location is a simple set of grid coordinates. The orientation is defined by a set of strengths in each of the x, y, and z directions, typically (but not necessarily) within the range [-1,1]. These values are the coefficients of the x, y, and z components of the electric field excitation vector. This system permits the user to define whatever excitation location and direction is desired.

3.2.3.3 New Pulse Shapes

The original code featured a Gaussian pulse and a cosine-modulated Gaussian pulse.

$$s = \exp\left(-\frac{1}{w}(n-p)\right)^2 \tag{3-1}$$

$$s = \exp\left(-\frac{1}{w}(n-p)\right)^2 \cos(2\pi f n \Delta t) \tag{3-2}$$

s is excitation strength, which is unitless. w is the pulse width, measured in time steps. n is the time step number, measured in time steps. p is the pulse peak time, measured in time steps. f is the pulse frequency, measured in hertz. Δt is the time step size, measured in seconds.

The new code corrected errors in the original formulation of these (see Section 3.5.) and added two new pulse shapes, the half-Gaussian and the cosine-modulated half-Gaussian. These two pulses are equivalent to the Gaussian and cosine-modulated

Gaussian, respectively, before the pulse peak time. After the pulse peak time, the half-Gaussian is fixed to one while the cosine-modulated half-Gaussian is fixed to the value from the cosine term:

$$t = \begin{cases} \exp\left(-\frac{1}{w}(n-p)\right)^2, n < p \\ 1, else \end{cases} \tag{3-3}$$

$$t = \begin{cases} \exp\left(-\frac{1}{w}(n-p)\right)^2 \cos(2\pi f n \Delta t), n < p \\ \cos(2\pi f n \Delta t), else \end{cases} \tag{3-4}$$

3.2.4 More Powerful System For Viewing Results

Once the material distribution and excitation for an FDTD simulation has been defined, the simulation can progress, calculating electric and magnetic fields throughout the computational domain. It is for these fields that the code is written and the simulation is run. However, in order for these fields to be useful, they must not only be calculated but also presented to the user in such a fashion that the user is able to ascertain what they are.

The electric and magnetic fields from a three-dimensional FDTD simulation are each three-dimensional vector fields. Because the code cannot handle vectors, it instead separately calculates each of the three components of both the electric and magnetic field vectors at each point in space-time.

To allow the user to view the fields, both the original and final codes write portions of the fields to text files. (Writing the entire fields is unnecessary and requires excessive disk space.) However, text files are inherently two-dimensional, so each file contains a two-dimensional slice through space at a given point in time for one electric or

magnetic field component vector. The codes are designed to write to file a series of field component slices at evenly spaced time intervals.

The original code's output system, however, had several limitations. It limited the user in what field component slices could be outputted. Also, it did not automatically reformat results files when the grid size changed. Finally, it used a confusing system for naming results files. All of these limitations were addressed in the final code. In addition, a set of MatLab companion codes designed for easy viewing of results files is included with the final code.

3.2.4.1 More Control Over Results Selection

One aspect of the FDTD technique that makes it so powerful is that it calculates the electric and magnetic fields at all points in space and time within the specified computational domain. However, in order for this powerful aspect to be worth anything, the user must have access to the results of these calculations.

In the original code, the user was only presented a fixed selection of the field calculations. While this may have been sufficient for the ground penetrating radar cases for which the code was designed, it prevents the code from being more widely useful.

The new code addresses this issue by letting the user define what portions of the field calculations to output. In the new code, any time series of two-dimensional slices (sliced in either the x-y, x-z, or y-z plane) of field component values can be written to a corresponding series of results files, which can then be viewed by the user.

3.2.4.2 New Automatic Results File Formatting

In the original code, the formatting for data output was designed so that each time the grid size changed, many lines of code needed to be changed accordingly in order for output to be formatted properly. However, the need to make these changes as well as how to go about making them were not documented or otherwise explained to the user, despite being both crucial to producing meaningful results and difficult to perform due to subtleties of the Fortran programming language.

In the final code, the output data formatting system has been overhauled so that output automatically reformats when the grid size is changed, sparing the user from ever having to adjust format statements within the code. Furthermore, all of the format statements have been consolidated into a single line of code, simplifying this part of the code for anyone reading or modifying it.

3.2.4.3 Improved Results File Naming System

Once the results files are written, the user must be able to identify what field component values lie within them. To do this, it helps tremendously if the results files' names correspond intuitively to the field component values they contain.

The original code used a system of file names that often did not correspond intuitively to the field component values within the corresponding files. For example, a file containing a slice of the electric field in the x-direction (Ex) along the x-z plane at the point y=13 and at the second output time step would be called ``mbdvx001.dat''. Because the original code did not offer full control of which slices were outputted, its file naming system did not include the coordinate (in this case, in the y-direction) through which the

slice was taken. Also, the 001.dat signified the second output time step because the counting system began at 0 instead of at 1.

The final code replaces the original code's results file naming system with a more intuitive system. For example, the same file in the final code would be called ``ex_y013_t002.dat"", signifying that the file contained the values for the electric field in the x-direction taken along the plane y=13 at the second output time step.

Output time steps are the time steps at which results are outputted. It would be undesirable to require results to be outputted at all time steps because it would cause the code to run more slowly (because writing data takes time) and cause hard drives to fill up more quickly for data that is often not needed by the user. Thus, the user has the option of specifying how often to output results. This value is measured in time steps per output time step. Thus, a value of 10, for example, corresponds to results being outputted every 10 time steps.

3.2.4.4 New MatLab Companion Codes For Viewing Results

Once results files are written and the user understands what values they contain, they must somehow be viewed by the user for analysis. The insights gained by viewing results are ultimately the motivation for developing the code and running simulations. Without an effective means of viewing the results, the rest of the effort is meaningless.

Users can easily open up results files in a text editor and look at the field component value numbers directly. However, finding meaning in the raw numbers can

be difficult, especially since there are often too many numbers on each line of the file to avoid line wrapping.

In order to facilitate the viewing of results files, a set of companion codes for the MatLab software package has been developed. The codes can easily be extended by anyone familiar with the MatLab language. One code reads the numbers from time series of results files into three-dimensional MatLab matrices. Another produces animations of these matrices. These animations are a highly effective means of viewing the results produced by FDTD simulations. A third code helps users obtain specific field component values from their respective matrices.

3.3 Making Simulations Easier to Initialize

A simulation can have many powerful features, but if they are not easy to implement, then they are of little use to most would-be users. The user needs some easy means of describing to the code what simulation to run. This description must include the parameters specifying the material distribution, the excitation, and the desired results, as well as the parameters connecting all of these.

The final code contains several improvements to the original code that make these parameters easier to enter into simulations. One key improvement is that more of the parameters are inputted through an external command window interface instead of through changes to the code itself. Furthermore, the sequence of inputting parameters through the command window was restructured so that the parameters are inputted in a more logical sequence. Another improvement makes it easier for the user to change those few parameters that need to be changed within the code. Another improvement is

the automatic generation of an input file which can be used to speed the inputting of simulation parameters. A final improvement is the addition of catches in the code of parameter entries that are in some way erroneous.

3.3.1 More Parameters Defined Through External Input

To use the original version of our code, one would answer a series of command window questions and adjust certain parameters within the source code. Once the internal parameters were properly set, running new cases was straightforward. However, this limits the user to varying only a portion of the available parameters, an undesirable limitation. Alternatively, setting these internal parameters was not trivial, requiring an intimate understanding of a sophisticated, 6,000 line code.

Now, the code is in a format such that a user does not, with one possible exception (grid size), ever need to adjust any parameters within the code, and thus can adjust the full range of parameters without ever looking at the code.

An example of an important feature that has become easy to control externally is the set of results that get outputted for viewing and visualization. Initially, this set was restricted in three ways: First, by the range of data that could be viewed, second, by the cryptic nature of the output file names (discussed elsewhere), and third, by the format of the data returned to the user, which was restricted by a line-wrapping parameter contained within the format commands in the code.

The first problem was addressed by allowing the user to define what data to output from within the command window interface. Now, the user can specify points,

lines, planes, or solids of data at any location within the computational domain to output for viewing, and can select which field components within these regions to view.

The third problem was addressed by modifying the format statements within the code so that line wrapping of results would be automatically set to the desired dimensions of the data. This feature permits new users to produce and view simulation data without learning anything about Fortran format statements, or even that such statements exist and matter.

3.3.2 Improved Parameter Input Sequence

In the original code, the order in which those parameters that users did input (as opposed to changing inside the code) was not entirely intuitive. For example, between a question asking if a monopole antenna should be built and a series of questions asking the dimensions of the antenna if it should be built, the user would be asked whether or not to print magnetic field values, a parameter entirely unrelated to the monopole antenna.

In the final code, care was taken to order input questions in a more logical fashion. To achieve this, all input was arranged into categories. Questions regarding monopole antennae fall under the ``foreground objects'' section; questions about printing field values fall under the ``field component output'' section.

Incidentally, this improved organization also makes code development easier because it has the effect of modularizing input, making it easier to find or add input questions without worrying about whether other input sections would be affected.

3.3.3 Improvements To Adjusting Parameters In Code

The one set of parameters that occasionally must be adjusted within the code is the set of grid size parameters. These parameters represent the number of finite difference grid cells in each of the three spatial dimensions; the number of grid points equals the product of these three values. The memory requirement of a simulation is roughly proportional to the number of grid points. Because the Fortran programming language does not permit the user to allocate memory to grid size variables without adjusting the code (i.e. it cannot dynamically size arrays), users must adjust these variables in the code in order to avoid running simulations that require more memory than their computer has.

To make the process of adjusting these variables as painless as possible, two actions were taken. First, the variables which control how much memory gets allocated for grid cell variables were pulled to the top of the parameters file. This code was then commented thoroughly to highlight where these variables are and how to change them. Second, a thorough explanation of how to change these variables was included in the user manual, including why these variables must be changed from within the code, where to look in the code to change them, and what restrictions exist in changing these variables.

Given these improvements, the code's expected users, predominantly engineers, will likely be capable of making these adjustments to the code. Indeed, test trials by various users including graduate students, undergraduate students, and high school students indicate that the requirement of adjusting these variables in the code is not a severe limitation.

3.3.4 The New Input File

Following the command window's question and answer format is tedious when running multiple simulations. It is especially tedious when successive simulations have similar input parameters, such as when studying the effects of varying one or more input parameters. For a user interested in avoiding re-typing the answers each time the program is run, several options exist.

Because a user can paste text into the command window, input can be stored in a text file and copied in. If the text file includes line breaks in the appropriate places, multiple lines of command window questions can be pasted in, including an entire set of input. Once such an “input file” has been created, the user can easily change the values of parameters within it and re-paste the entire input set into a new command window simulation. Such an input file can also be used as a record of the parameters used for a particular simulation. Indeed, due to hard disk limitations, it is often advantageous to store a simulation's input file instead of its results because the results take up considerable memory and can be reproduced from its input file if needed.

One disadvantage of using such a user-created input file is it must be recreated each time the structure of the input is changed. For example, if the user decides to change the input to add a monopole antenna, a series of questions determining the antenna's dimensions must be answered. The answers to these questions would need to be added to the input file. To do this, the user can either add these answers to the input file as they're typed into the command window or add them from prior knowledge of what they should be, either from the user's memory or from sample monopole antenna answers found in other text files. While this disadvantage of user-created input files does not make their use prohibitive, it does mean that their use requires some learning.

Another disadvantage of user-created input files is that they don't easily show the user where mistakes in the input are if any exist. If an entire set of input is pasted into the command window and contains an error somewhere in it, the program will crash without providing any information regarding where the error is located. The user must then either attempt to locate the error by examining the input file or by pasting successively smaller segments of the input file into the command window in order to isolate the error. As with the problems of modifying user-created input files for different input structures, the problem of finding mistakes in user-created input files is not excessively difficult, but it does make their use more less friendly.

In response to these disadvantages of user-created input files, the new code automatically generates an input file as data is entered. Each time the user enters a new line of data into the command window, regardless of what technique is used to enter the data, a new line containing this data and a commented explanation of it is added to code-created input file “input.txt”. Due to formatting limitations of the Fortran programming language, this file may not be as visually attractive as a user-created input file can be. However, it is functional and has several advantages over a user-created input file.

One advantage of the code-created input file over the user-created input file is that it is automatically generated, whereas the user-created input file must be built by hand. This is of particular value for a user answering input questions directly into the command window. For example, if the user catches a mistake in earlier lines of input, the code-created input file can be opened up and used to quickly correct the mistake and re-enter the corrected set of input up to the point where the user was when making the correction.

Also, the code-created input file can be used to reproduce the simulation or to run a similar case.

Another advantage is that it can be used to automatically detect where any errors may exist in the input: If an input file containing one or more errors is pasted into the command window, the code-created input file will contain no lines past the line containing the first error. Thus, the last line in the code-created input file contains an error, which then can be analyzed and corrected.

One possibility for future work would be to build a form for entering data in an environment such as Visual Basic or MatLab's GUIDE. Such a form could be more visually attractive than the command window or input files. It would also have the advantage of permitting the user to change answers to previous questions without restarting the questioning process. However, care would have to be taken to ensure that the user did not have to re-answer all questions each time a new set of input was desired, and to ensure that the form could easily feed the answers given it into the main simulation- possibly via the creation and automatic reading of an input text file.

3.3.5 New Error Catches

Many input parameters have specific sets or ranges of values that they can take in order for simulations to run properly. For example, the number of background layers to build can have any integer value from 1 through nz, where nz is the number of grid points in the z-direction. However, sometimes users enter values not within these sets or ranges, either by accident or because they didn't know what the sets or ranges were. In these cases, it is important for the code to catch these errors and respond in a way that permits

the users to recognize their errors, figure out how to correct them, and then input the corrected values.

The original code contained several error catches, but it also lacked error catches in places where they would have been helpful. Error catches may not be needed in these places when the code is used by knowledgeable, attentive users, but they become important when the code is used by a wider range of users. For this reason, several new error catches were added to the final code, such as those for grid size and background layer thickness. Error catches were also added for input new to the final code, such as the number of new materials to add and foreground object type number.

3.3.5.1 Courant Condition Check

Another example of a feature making the code smarter is its check that the user's input satisfies the Courant condition. Previously, no restriction on time step size existed, and users could fail to satisfy the Courant condition, leading to flawed results. Now, as the user inputs the time step size, a calculation is performed checking that the Courant condition has been satisfied. If it has not been, the user is told what time step size values are acceptable and is prompted to re-enter a value for time step size.

3.3.5.2 Points Per Wavelength

Another example of a feature making the code smarter is its display of points per wavelength. The original code included no consideration of points per wavelength. However, typically, at least about 10 points (grid cells) per wavelength are desired to ensure that resolution is sufficient to capture the details within each wavelength. This 10

points per wavelength limit is not strict, and users may occasionally wish to dip below it. So, instead of requiring that it be met, the new code simply calculates points per wavelength values and reports them back to the user.

Because wavelength is inversely proportional to the square root of electric permittivity, the material in the computational domain with the highest permittivity is used. This ensures that the calculations show the minimum number of points per wavelength possible in the simulation. Four different points per wavelength values are calculated: one in each of the x, y, and z directions, and one on the diagonal from one corner of a grid cell to the opposite corner. This information helps the user gauge if the grid cell size being used is small enough to offer sufficient resolution.

3.4 Making the Code More Powerful

With a code at once versatile and easy to initialize, a wide range of users can run a wide range of cases. However, the range of cases they can run may still be limited by the power of the computers they have available to run the cases. Indeed, its large memory requirement is a main limitation of the FDTD technique.

Studies of the code, confirmed by simple experimental results, show that memory requirements can be divided into fixed memory overhead and variable memory that is linearly proportional to the grid size (i.e., the number of grid points). This means that for large grid sizes, the amount of memory required is approximately linearly proportional to the grid size. This also means that the maximum allowable grid size is approximately linearly proportional to the amount of memory available.

Because of this, it became important to streamline code so that large grid sizes could be accurately simulated. By reducing the memory requirement of field component values from 8 bytes to 4 bytes, the final code enables grid sizes approximately twice as large to be simulated. Experimental simulation results demonstrate that this memory reduction does not cause any significant loss of accuracy.

3.5 Making the Code More Error-Free

The original code contained a small number of errors affecting the code's functionality that have been corrected for the final version.

One such error was in the excitation pulse shapes. The original code contained equations for both Gaussian and cosine-modulated Gaussian pulses. The code also used separate definitions of cosine-modulated Gaussian pulses for monopole antennae and for other excitations (Equations (3-5) through (3-7)).

Gaussian pulse:

$$s = \exp\left(\frac{-1}{w}(n\Delta t - p)\right)^2 \tag{3-5}$$

Monopole antenna, cosine-modulated Gaussian pulse:

$$s = \exp\left(-f(n\Delta t - p)\right)^2 \cos(2\pi f n\Delta t) \tag{3-6}$$

Other excitation, cosine-modulated Gaussian pulse:

$$s = \begin{cases} 0, & |n - 200| > 200 \\ \exp\left(-f(n\Delta t - p)\right)^2 \cos(2\pi f n\Delta t), & else \end{cases} \tag{3-7}$$

s is excitation strength, which is unitless. w is the pulse width, measured in time steps. n is the time step number, measured in time steps. Δt is the time step size,

measured in seconds. p is the pulse peak time, measured in time steps. f is the pulse frequency, measured in hertz.

Each of these equations contain errors, which were corrected for the final code. Corrected Gaussian pulse:

$$s = \exp\left(-\frac{1}{w}(n-p)\right)^2 \tag{3-8}$$

Corrected cosine-modulated Gaussian pulse:

$$s = \exp\left(-\frac{1}{w}(n-p)\right)^2 \cos(2\pi f n \Delta t) \tag{3-9}$$

The final code also consolidates the two cosine-modulated Gaussian equations. Finally, two new pulse shapes, the half-Gaussian and cosine-modulated half-Gaussian. (See Section 3.2.3.3.)

3.6 User Manual

Despite the user interface improvements described here, learning to successfully use the code is not trivial, especially to users not familiar with command window interfaces or with FDTD.

To help users learn how to use the code, an extensive and attractive User Manual has been developed. It introduces the FDTD technique and explains how the code implements it. It explains how to use the command window interface and explains all of the input parameters one could enter into it. Finally, it follows sample simulations from input to results visualization to give users examples of how to use the code.

The complete User Manual is included as an attachment.

Chapter 4:
Developer Interface Improvements

4.1 Introduction

Our code is primarily intended for use in an academic environment which, in addition to running FDTD simulations for its research, conducts research to improve the FDTD technique itself. Therefore, it is desirable to have a code that is easily modified to include such improvements or other new features.

In this section we discuss modifications to the code that make the code easier to be developed. These are strictly “cosmetic” modifications to the code, i.e. modifications that do not in any way affect the code's functionality. While they are transparent to users, they are of tremendous benefit to developers.

Developer interface improvements fall into three interrelated categories: Those making the code easier to read, those making the code better organized, and those making it easier to add on to the code.

4.2 Features Making The Code Easier To Read

Writing computer code is similar to writing poetry in that considerable freedom exists to lay words and other symbols out on the page in any particular fashion. While changing the layout generally won't affect the code's functionality, it will affect how easy the code is to read, understand, and modify.

Because the original code was written for personal use, it was not designed to be easy for others to pick up and use. Furthermore, because the code passed through so

many different hands and was adjusted slightly by each of them, it wound up a confusing hodgepodge of various programming styles. Because of this, considerable opportunity existed to improve the code's readability for the final version.

Ways in which the final code was improved to make it easier to read include the use of more intuitive variable names, improved capitalization schemes, consolidation of redundant and non-functional code, removal of goto commands and line label numbers, improved indentation schemes, a simplified material i.d. system, and removal of dead-end variables and subroutines.

4.2.1 More Intuitive Variable Names

To ensure that codes are easy to follow, it is important for their variables to have names that clearly correspond to their function. For example, one should probably never use the variable “pi” for anything other than the constant 3.14159.

The final code replaced several variable names from the original code with more intuitive variable names. For example, the variables containing the material type at each grid point and in each direction (anticipating future need to handle anisotropy, the code allows for grid points to have different material types in different directions) were renamed from idone, idtwo, and idthre to mtx, mty, and mtz. In the original code, "id" referred to "material identification number", which was problematic because the code had multiple types of identification numbers. The final code's "mt" clearly refers to "material". Indeed, in the final code, all variables relating to materials contain the characters "mt". In the original code, "one", "two", and "thre" refer to the directions x, y, and z. This is less obvious than simply using "x", "y", and "z", as done in the final code.

Another example comes from the monopole antenna's built-in excitation. Because the excitation is circularly symmetric, sines and cosines need to be calculated from corresponding triangle side lengths, which are in turn calculated from their endpoints. The original code used two variables, "fltetax" and "fltetay" for the length of the triangle's hypotenuse. Because the variable names don't allude to hypotenuse in any way, it was hard for developers to realize that that's what the variables represented. Furthermore, using two separate variables suggested that there were two separate values, presumably one in the x-direction and one in the y-direction, when upon inspection one finds that both variables contain the same value. In the final code, these variables have been replaced by a variable "hyp" which clearly alludes to the value it contains. The final code also creates variables "adj" and "opp", containing values for the lengths of the sides adjacent to and opposite to the angle in question, further clarifying what the code is doing- all without changing the code's functionality.

4.2.2 Comments

As is often the case with code passed among several developers, each of which worked on it for personal use, the original code contained sparse and confusing commenting. The state of the code's commenting was such that the code was difficult to follow and thus difficult to modify.

As the current code was being developed, care was taken to include good commenting. Previous unclear comments were reworded or removed, and additional commenting was written where it would be useful. The net effect was to make the code easier for developers new and old to understand and work with.

Examples of comments removed from the original include “ahai inajst” (which may be a clear comment in a language other than English) above code related to lossy dielectrics and “asl” next to code defining the core of a monopole antenna.

An example of commenting added is that done to identify distant yet connected statements such as “For each time step” next to the end of a do loop which iterated through each time step. In addition, considerable commenting was added throughout the code to simply explain the functions of various pieces of code.

4.2.3 Indentation

Indentation is traditionally used to vertically line up lines of code at the same level within loops, conditional statements, etc. Considerable care is required when using indentation in Fortran in order to avoid pushing code past the 72nd column, because Fortran does not run any characters placed beyond this column, i.e. it treats all such characters as commenting.

While the original code did use indentation, it did so unevenly, giving it a choppy appearance on the screen and making its logic structure more difficult for developers to follow. Indentations would appear where the code's logic structure would not suggest they should be, or use an unusual or inconsistent spacing, or not appear where they might be expected.

The new code uses a more consistent indentation scheme, giving it a smoother appearance. Non-intuitive indentations were avoided, and consistent spacing was used where permitted by Fortran's column width restriction. Care was taken to resolve column width issues intuitively and attractively.

4.2.4 Blank Line Removal

Blank lines in the code have no function but can serve to separate sections of code visually. However, they also cause code to take up more lines, shrinking the amount of code that developers can examine at a time. The original code was written with a very sparse style featuring many blank lines which often broke up a logically continuous section of code. This not only reduced the amount of code that could be analyzed at a time, but it also made it more difficult to determine how lines of code related to each other.

The new code uses a much more concise, smooth formatting style. Only a minimal number of blank lines appear, helping the user view more of the code at one time. Indeed, between removing blank lines and removing commented out and other non-functional lines of code, the code size was reduced from about 6,300 lines to about 2,800 lines. In some instances, however, blank lines were added to help developers follow the code.

4.2.5 Redundant Line Consolidation

A further way in which the code was made more concise was by consolidating redundant lines of code. These are instances in which multiple lines of code perform the same task. Not only do superfluous lines unnecessarily consume space in the code, but they can also confuse developers, who typically assume that each line has a separate, essential function.

One example of consolidating redundant code is with the absorbing boundary code. The absorbing boundary code scans along the absorbing boundaries one electric field component and one pair of opposite faces at a time. In the original code, a separate set of identical equations appeared for each of these two faces. In the final code, a do-loop runs a single set of these equations twice, once for each face. Because the absorbing boundary equations are long and complicated, this system significantly shortens the code and reduces the chance for errors exactly where errors are most likely to occur.

Another example is with the definition of built-in materials' relative permittivities. The original code performs this one task a total of three times: once in the parameters file, once in the main code, and once in a subroutine. Users adding a new soil type thus need to add three lines of code to ensure that one value is defined properly. The final code does this only once, in the subroutine. The subroutine also defines other material properties and is the logical location for the relative permittivity definitions, making the relative permittivity definition process not only easier but also more intuitive.

4.2.6 Capitalization

Fortran is not case-sensitive, permitting variables and commands to be capitalized or uncapitalized in any way. Thus, capitalization can be used to add emphasis, consistency, or for other visual purposes. The original code made no standard use of capitalization, passing on an opportunity to make the code easier to read. Furthermore, it used considerable shouting, or excess capitalization, which actually makes the code more difficult to read.

The new code addresses these issues by applying a uniform standard for capitalization using minimal amounts of capital letters. This allows code readers to use capitalization to help understand the code, because when capitalization is used, it is used for a reason, generally to emphasize the letters or words being capitalized.

4.3 Features Making The Code Better Organized

In addition to the visual code improvements described above, changes were made to the structure of the code that make it easier for a developer to follow but have no affect on the code's functionality. These include improved organization of the parameters file, a simplified material i.d. system, improved subroutine ordering, and the removal of dead-end code.

4.3.1 Parameters File Organization

The code includes a separate parameters file in which parameters, arrays, and global variables, which we will refer collectively to as “terms”, are defined. The original code had these terms organized haphazardly in the parameters file, making it difficult to ascertain what the terms did. The final code reorganized these terms in an intuitive categorization, making the file easier to follow and adjust.

The first level of organization in the new parameters file is by how frequently the terms may need to be changed. At the top are terms that need to be changed most frequently. At the bottom are terms that are derived from other terms and thus never need to be changed.

The second level of organization in the new parameters file is by how terms function in the rest of the code. For example, there are sections for “Material Variables”, “Excitation Variables”, “Space & Time Dimension Variables”.

The effect of this organization is to make it easier for developers to both find specific terms and understand what they are used for. When combined with the features making the code easier to read and the removal of dead-end variables described elsewhere, this organization results in a much more accessible parameters file.

4.3.2 Material I.D. System

Material types make for natural programming objects because they each contain the same set of attributes. These are the material properties, including permittivity, conductivity, and the frequency dispersion approximation variables. However, Fortran is not an object-oriented programming language, so alternative arrangements must be made to handle material types.

The code does this by assigning an identification number to each material type. For example, in the final code, 1 is free space, 2 is metal, and 10 is a lossy Bosnian soil.

The original code included a confusing and unnecessary twist to this numbering scheme: It ran two parallel numbering systems. One, presented to the user, started numbering soils at 1. The other, used for all computations, started the soils at 14. When the user inputted a soil type, the code immediately added 13 to its i.d. number. This arrangement forces any developer working with material i.d.'s to know where each system is used, thereby distracting the developer from other work.

The final version of the code contains a single numbering system, used on both the user side and the code side.

4.3.3 Subroutine Ordering

The main way in which the code has been modularized is through the creation of subroutines which handle certain tasks that are usually repeated frequently throughout the code. These include the building of foreground objects, the calculation of field component values, and the writing of results files. As measured in lines of code, the subroutines make up the overwhelming majority of the code.

Due to their nature, subroutines can appear in any ordering. However, illogical orderings make the code more confusing and difficult to follow. In the original code, the subroutines were not always ordered in a logical way, with some parts of the ordering making sense and others not making sense. This was corrected in the final code, making it easier for developers to follow.

4.3.4 Dead-End Code

Dead-end code is code that doesn't actually affect any aspect of simulations. This does not include commented-out code, which has no potential to affect simulations. Because dead-end code doesn't affect simulations, it is harmless to users. However, it can cause considerable confusion to code developers who are likely to assume that all code present has some relevant function. If left in, dead-end code serves only to distract developers from their work.

The original code contained dead-end code in two forms: dead-end variables and dead-end subroutines. The final code addressed both of these problems.

4.3.4.1 Dead-End Variables

Dead-end variables are variables that appear in the code but don't actually affect any aspect of the simulation. Typical dead-end variables are variables that get defined but never get used. They could also be variables that are only used to define variables that never get used, or variables that are only used to define variables that are only used to define variables that never get used, and so on.

The original code contained about 50 dead-end variables; the new code contains, to the best of the author's knowledge, none. Care was taken to ensure that variables being removed were indeed dead-end, including testing of new code versions to ensure that results were consistent with previous versions. This effort will help developers focus on their work instead of on deciphering what variables do or don't do, thereby increasing their productivity and lowering their frustration.

4.3.4.2 Dead-End Subroutines

In two instances in the original code, entire subroutines had no functional effect on simulations. One, a subroutine designed to handle metal antennae, was simply never called by the rest of the code and thus was never used. The other, a subroutine for initializing a set of variables to 0, had no effect on simulations because Fortran automatically defaults variable values to 0. In both cases, the subroutines were removed and tests were performed to confirm that no change in functionality occurred.

4.4 Features Making It Easier To Add On To The Code

While the changes described above making it easier for developers to read and understand the code are important, ultimately developers are not interested in reading and understanding the code but in modifying it. Thus, changes to the code have been made to make this specific task easier. They include the removal of line label numbers, increased modularization, improvements to features that may get added on to the code, improved implementation of arrays, and features made into lists.

4.4.1 Removal Of Line Label Numbers

Fortran offers the option of labeling any line of code with a line label number. Labeled lines can then be referenced for a variety of purposes.

One common use of line labels is in “goto” commands. For example, the command “goto 234” sends the code to line 234, wherever in the code that line is. Such commands can be used in place of loops or functions, or simply to rearrange code lines.

Use of such goto commands has several disadvantages. One is that sets of line label numbers require maintenance. All line label numbers must be distinct, and efforts to order the numbers meaningfully are easily stifled when new line label numbers must be inserted in the middle of the ordering.

Another disadvantage of using goto commands is that they can lead to awkwardly-structured ``spaghetti" code, or code whose execution skips throughout the code in a manner as challenging to follow as a piece of spaghetti on a plate full of the stuff. For this reason, use of goto commands is controversial within the programming

community. Legendary computer scientist Edsger Dijkstra went so far as to publish a paper "Go To Statement Considered Harmful" [Dijkstra 1968].

For these reasons, all goto commands and corresponding line label numbers have been removed from the code. In their place, programming techniques including if/endif and do/enddo were used.

The final code contains exactly two line label numbers. These are for the format lines used in write statements for the subroutines which write material and field component output files. Fortran syntax necessitates the use of line label numbers in these statements. However, multiple write statements can share the same format line if each references the same line label number. The original code used separate, identical lines of code for each format statement, each with its own label number. If the format needed to be changed, then it would need to be changed for each separate format statement, a tedious task. Because these statements were all the same, the final code consolidates them into a single line of code. This streamlines the code, makes it easier to modify the format of all write statements at once, and eliminates the maintenance requirements of larger sets of line label numbers, all without any change in functionality.

4.4.2 Increased Modularization

Modularization is the breaking of code into sections. It permits repetitive tasks to be performed with the same lines of code. It also makes the code easier to read and easier to modify, because code performing a specific task is isolated from the rest of the code.

The original code included several subroutines, laying the foundation for a well-modularized code. The final code added several modularized features. These include a subroutine to print material types.

4.4.3 Improvements To Features To Add On To The Code

One main way in which developers may modify the code is by adding new instances of certain code features without changing the overall structure or functionality of the code. These features include material types, material objects, pulse shapes, background medium geometries, and material dispersion models.

The original code made some effort to streamline the process of adding new instances. However, this process nonetheless often required extensive and complicated programming. The final code completed this effort, making this process considerably easier, thereby encouraging it's execution.

4.4.3.1 Material Types

In addition to using the new option for user-defined materials, code developers may want to build in new material types directly into the code because then users won't have to reenter the materials' properties each time they use them in a simulation. The final code features several improvements to the original code that made this process easier.

One improvement is the removal of redundant code lines defining a material's relative permittivity. Thus, when a new material is built in, only one line of code needs to be added to describe its relative permittivity, instead of the original three. Furthermore, as

explained in the Redundant Line Consolidation section, this one line appears in a logical place: the subroutine that defines all material properties and variables derived from material properties.

Another improvement to the original code making it easier to add new material types is the consolidation of electric field component update equations in both the interior region and along the absorbing boundary. The original code had five different update equations for electric fields in the interior region and two on the absorbing boundary, and developers adding new materials needed to take care to ensure that the proper update equations were used. The new code only uses two in both regions: one for perfect electric conductors (metal), and one for everything else. (The original code did not handle perfect electric conductors along the boundary.) Since a perfect electric conductor is already built in and thus need never be added, anyone adding new materials does not need to attend to the update equations in any fashion.

Previously, materials using the same dispersion model were grouped by having their i.d. numbers run consecutively. This grouping was used in code logic and thus had functional significance. The materials using the numbers 14 through 29, for example, were all materials using the dispersion approximation, so in certain lines of code, certain actions would be taken if a material's i.d. number was between 14 and 29. This system was problematic, however, when adding new materials, because it depended on numbers being reserved for new materials of a given type so that material's i.d. number can be within the grouping of consecutive numbers. Even if this could be done, the range would have to be adjusted throughout the code (i.e. from 14 to 30, etc.). Now that, in the new

code, the same equations are used for all material types, no such adjustments ever need to be made, and any type of material can be added in any order.

Another new feature facilitating the addition of new materials is a subroutine that prints a list of the built in materials to the command window. Previously, this list would be repeated in the code each time it would be presented to the user. Because of this, adding a new material required updating this list several times over. Using a subroutine for this list requires only one update when a new material is added.

The effect of these changes is that whereas in the original code, adding a new material required about 13 code changes, in the new code, only 4 code changes are required: the specification of the material's properties, the display of the material to the user, and the resizing of two parameters used to size arrays in the parameters file.

Making those changes have been facilitated by an improved commenting system. The final code extends a clever idea used in the original code of marking sections of the code requiring change with a unique character string, so that the sections can be located directly using the code editor's find function. One improvement is in the character string itself, changed from "type+19", which, while functional, is not meaningful, to "nwsltp", which is short for "new soil type". Also, the new code marks several places requiring change that were overlooked in the original code.

Another new feature that makes the process of adding new materials easier is a parameter used to automatically size those arrays which have a value for each material. The parameter's value is set to the total number of materials the code can handle, including both built-in materials and user-defined materials. The material array sizes are then set to whatever value this parameter has. Thus, when adding new materials, only one

number must be changed to resize all material arrays. This also ensures that all arrays will have the same length, helping conserve memory and avoid errors. This is a considerable improvement over the original code, which had about 27 different material arrays each needing to be changed individually. Indeed, these arrays had about 4 different sizes, which wastes memory and makes understanding the variables more difficult because it suggests that the arrays contain unrelated sets of information.

Thus, the final code features many changes to the original code which make adding new materials to the code considerably easier. As the new code strives to be more widely useful than the original code, it is expected that these changes will prove valuable.

4.4.3.2 Foreground Objects

Adding foreground objects to the final code is similar to doing so to the original code, because the original code already featured a fairly efficient system for it. The code building each object was encapsulated in its own subroutine, and a numbering system for object types already existed, although its use was not widespread throughout the code. There are no arrays associated with material objects, so there were no array-related complications with adding new objects.

The main change in the final code making it easier to add new foreground objects is the new system for adding them to simulations' material distributions. The original code's system for adding objects was poorly designed. It used a series of yes or no questions, one for each object type, to determine which objects should be added. For each yes answer, a series of questions would be asked to determine the object's properties. One problem with this system is that the user cannot build more than one of

any object type. The user must also answer all of these yes/no questions, even if the simulation does not include foreground objects. Finally, these yes/no questions and the object property questions were scattered throughout the sequence of command window questions, making the process of defining foreground objects more confusing.

The new code has one section in which all objects are added. It first asks how many foreground objects to build. Then, for each object, it asks what type of object it is, and then asks the appropriate object property questions. If the user answers 0 for the number of foreground objects to build, no further foreground object questions will be asked. This system makes the process of building foreground objects simpler and more versatile.

4.4.4 Improved Implementation Of Arrays

Because Fortran is not object-oriented, arrays must be used for features that would otherwise be designed as attributes of objects. For example, materials would make natural objects because they each have the same set of attributes: relative permittivity, conductivity, and dispersion parameters. The original code made effective use of arrays in general, although not in all cases.

For example, the original code used the number 0 as the i.d. number for the free space material. However, Fortran memory allocations for arrays begin with the index 1. Thus, any reference to an array index of 0 was reading from or writing to a separate memory space. While this caused no errors for the original code, as extensions to the code were being made, calls to index 0 of arrays were made, causing the code to produce incorrect results. This error can be challenging to detect because the variable sharing the

memory space with the array index 0 may be completely unrelated. Thus, to prevent this problem, material i.d. numbers were shifted so that all material i.d. numbers are natural numbers.

Another way in which the final code improved upon the original array system is in its system for declaring the arrays in the parameters file. The original code used numbers to size the arrays, forcing developers to adjust each array's number every time the arrays need to be resized, even though many of the arrays are always of the same size. In the final code, all arrays are sized using parameters. Thus, when a set of arrays need to be resized, only one change needs to be made- the parameter's value. In addition, improved parameters file organization and commenting, discussed elsewhere, further expedites this process. Finally, certain arrays in the original code were sized exactly large enough. Thus, adding even one additional feature to the code would require array resizing. The final code made most arrays larger than necessary in order to avoid this. Because these arrays were relatively small, their excess size did not significantly affect code performance. The one exception to this is in the grid size arrays (material distribution and field component values) because adding excess capacity there would adversely effect performance.

4.4.5 Features Made Into Lists

A code feature can be implemented in list form by assigning an identification number to each instance of the feature. Lists are a useful organization structure for code features because, in addition to being easy to use, it is easy to add on to them. For these reasons,

several features in the original code that were not in list form were put into list form for the final code. These include excitation pulse shape and source type.

4.4.5.1 Excitation Pulse Shape

Excitation pulse shape is the shape of the graph of excitation strength vs. time. The original code featured two options for excitation pulse shape: the Gaussian and the cosine-modulated Gaussian. These options correspond to common pulse shapes in the ground penetrating radar problem.

Because only two options were offered, the original code had users choose by asking a yes or no question if a regular Gaussian pulse was desired. This system has several limitations. First, it failed to tell the user that answering “no” meant a cosine-modulated Gaussian pulse would be used. This matter was easily fixed by changing the question's wording. However, more significantly, the system doesn't readily handle new pulse shapes because each new pulse shape requires additional questioning.

The new code addresses both of these issues by using a list for choosing excitation pulse shape instead of a yes or no question. Instead of a yes or no question, the user is asked to choose a pulse shape from a list which includes all available pulse shapes. Each pulse shape has a number, which is then used throughout the code when pulse shape is referenced. Thus, when a developer wants to add a new pulse shape, the shape is simply added to the list.

4.4.5.2 Source Type

Source type is the way in which the excitation is implemented in the code. This includes the choice of hard source or soft source (which define how the excitation field will be added to the fields at the excitation location) and excitation geometry (at what points and in what directions the excitation will be).

The original code offered no choice for source type because it only had one possibility: monopole soft source. However, as the new code became able to support hard sources as well as sources from other antennae, it needed to develop a system for the user to choose source types. As with other features, a list system was established, with each source type being represented by a number, which is then referenced throughout the code. Developers adding new source types then add to that list.

Chapter 5:

Sample Simulation Images

5.1 Introduction

In this chapter we present sample results obtained from simulations using our final code, in order to demonstrate its capabilities, especially in comparison to the original code.

5.2 Infinitesimal Dipole

FDTD models the infinitesimal dipole by a single excitation point. For a homogenous free space medium, this classic problem is readily solved analytically. Comparison of FDTD results to analytic results demonstrates the FDTD simulation's validity.

Here we present images from analytical and FDTD simulations of a z-directed dipole. Images show the y-z plane approximately one wavelength away from the dipole source. The dipole source is centered within the slice, as shown in Figure 3.

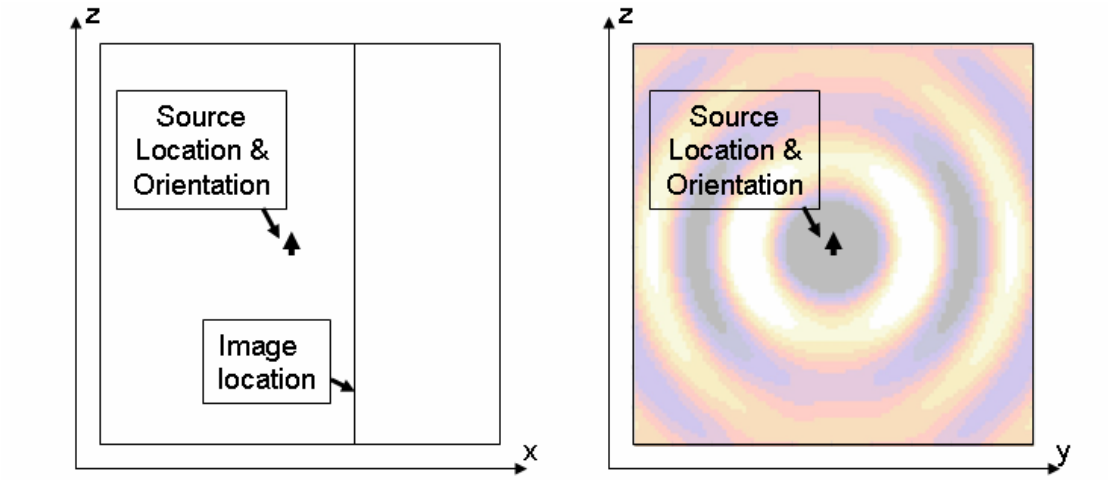


Figure 3: Infinitesimal dipole simulation geometry. Left: Perpendicular to image plane. Right: Image plane.

For each field component, four images are presented. The image on the top-left is from the analytic derivation. The image on the bottom-left is from the FDTD simulation and plotted using the MatLab companion code using the same scale as the analytic derivation image. The images on the right show the difference (FDTD – analytic) of the two data sets. The top-right image shows the difference plotted on the same scale as the images on the left. The bottom-left image shows the difference plotted on a scale ten times more narrow than the other images in order to observe finer detail in the difference.

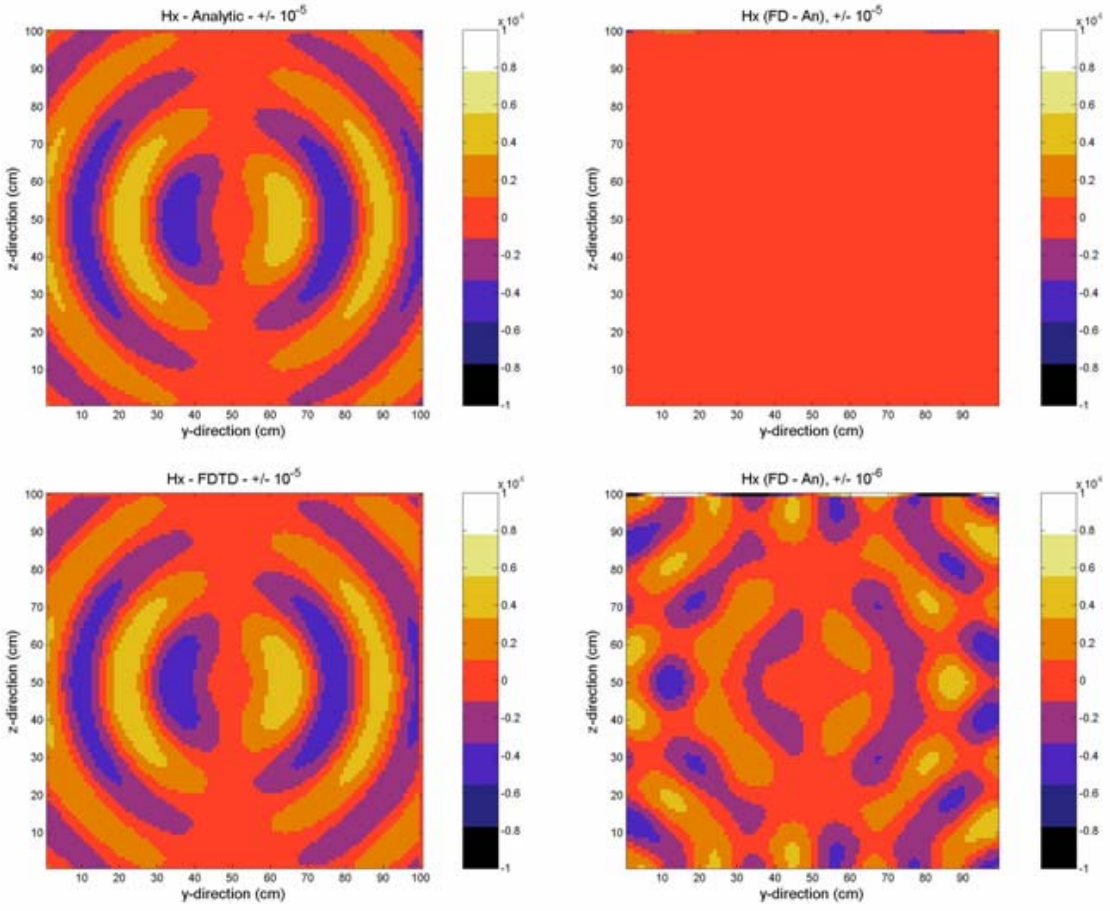


Figure 4: Magnetic field oriented in the x-direction (Hx). Top-left: From analytic derivation. Bottom-left: From FDTD simulation, same scale. Top-right: Difference (FDTD – analytic), same scale. Bottom-right: Difference, scale ten times more narrow.

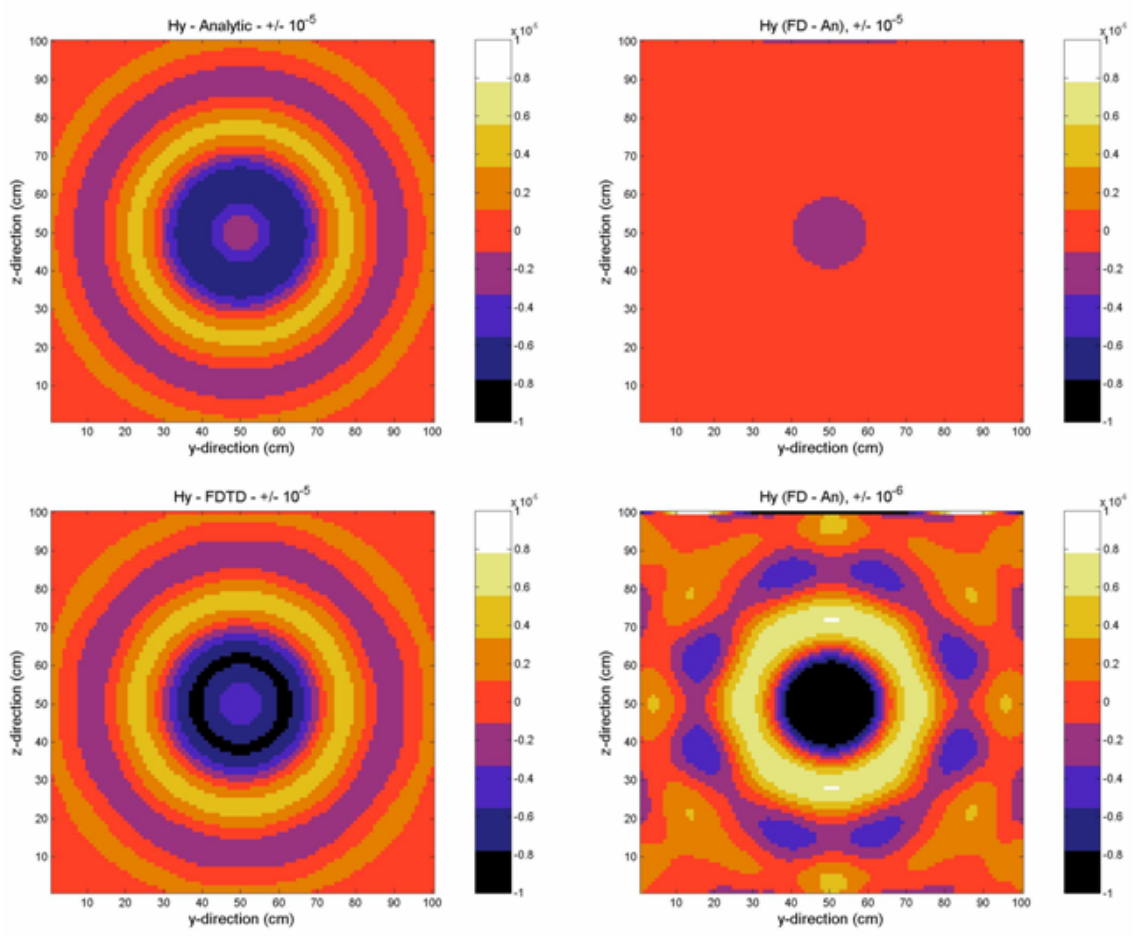


Figure 5: Magnetic field oriented in the y-direction (Hy). Top-left: From analytic derivation. Bottom-left: From FDTD simulation, same scale. Top-right: Difference (FDTD – analytic), same scale. Bottom-right: Difference, scale ten times more narrow.

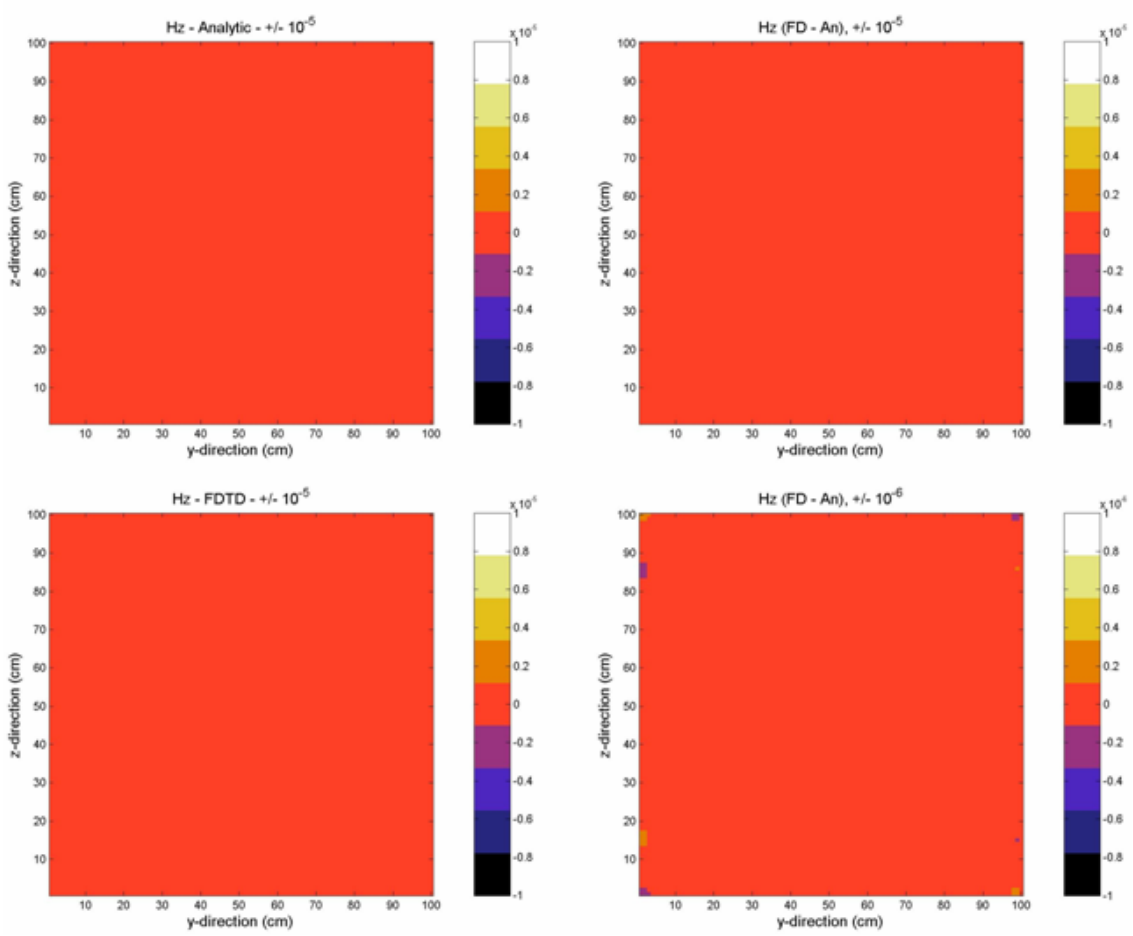


Figure 6: Magnetic field oriented in the z-direction (Hz). Top-left: From analytic derivation. Bottom-left: From FDTD simulation, same scale. Top-right: Difference (FDTD – analytic), same scale. Bottom-right: Difference, scale ten times more narrow.

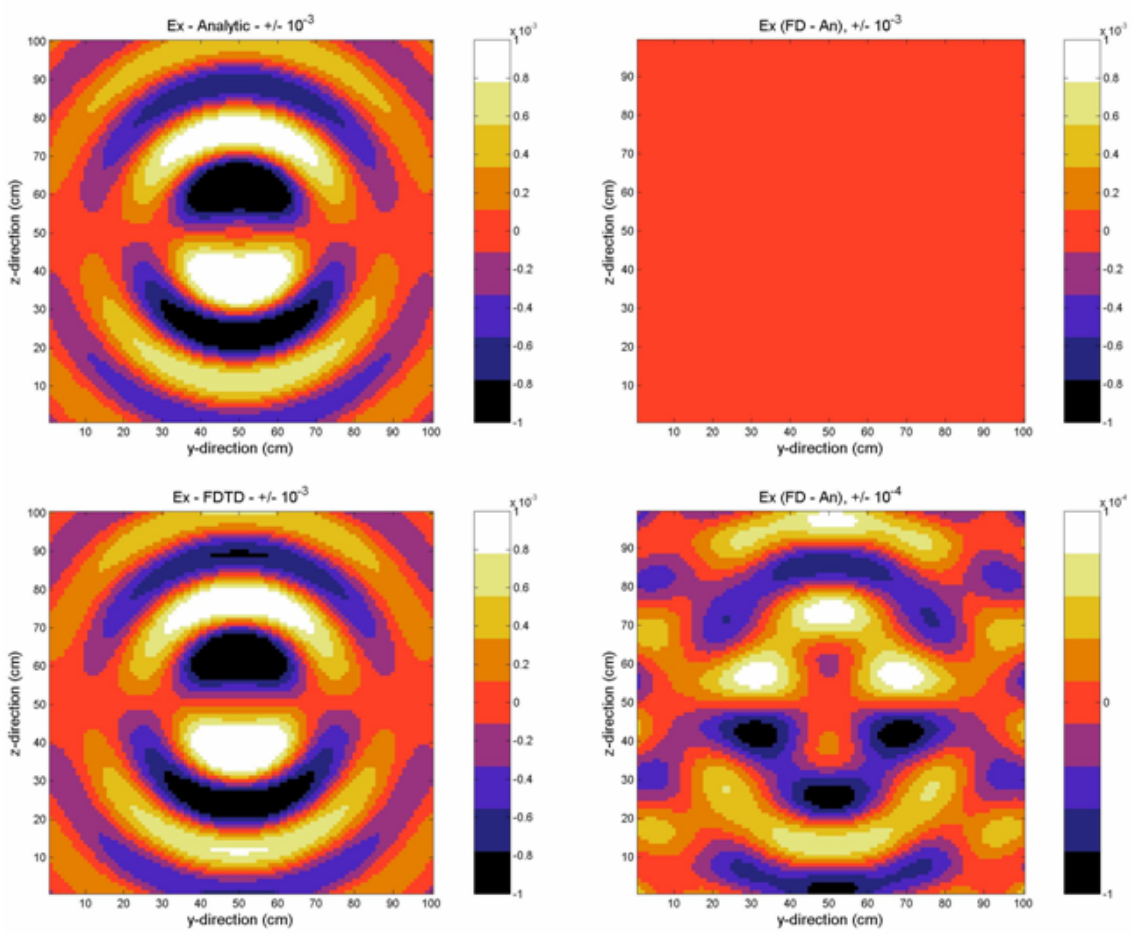


Figure 7: Electric field oriented in the x-direction (Ex). Top-left: From analytic derivation. Bottom-left: From FDTD simulation, same scale. Top-right: Difference (FDTD – analytic), same scale. Bottom-right: Difference, scale ten times more narrow.

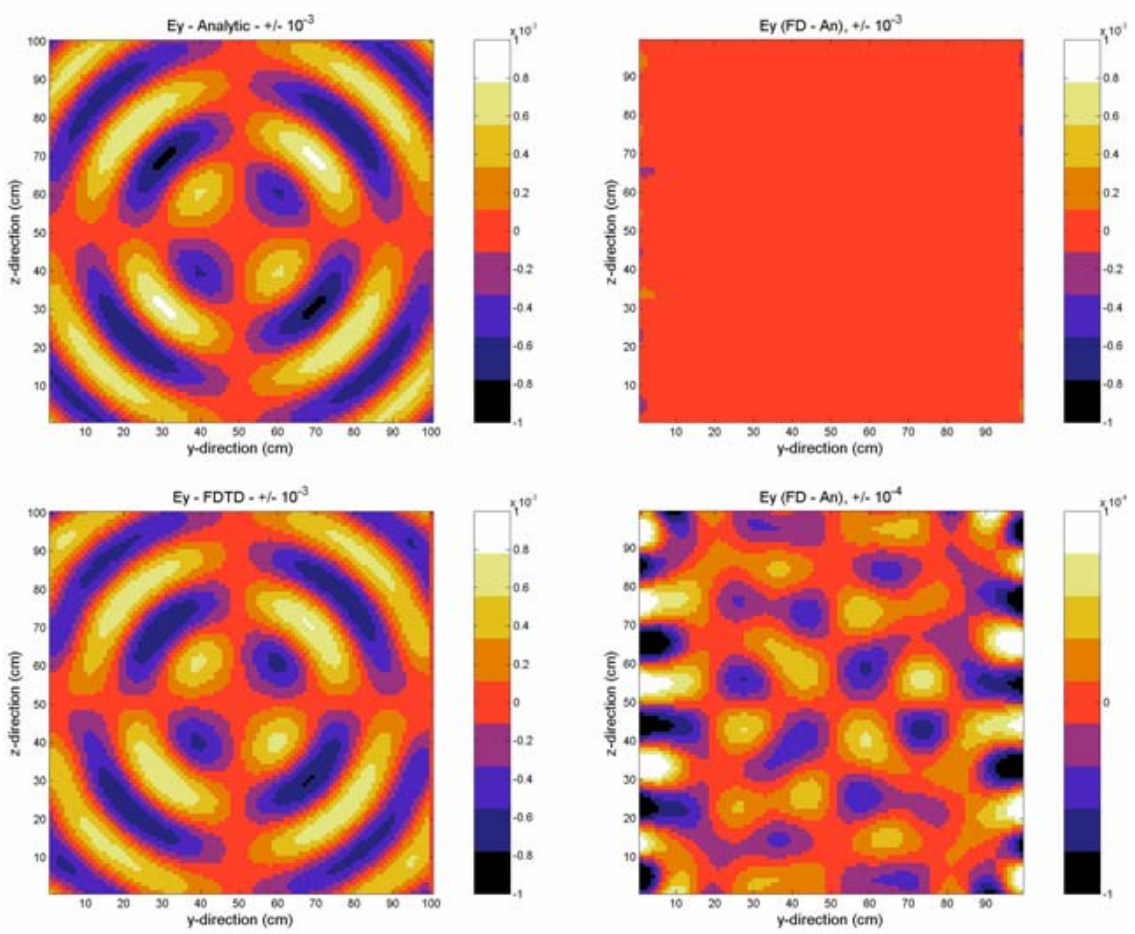


Figure 8: Electric field oriented in the y-direction (Ey). Top-left: From analytic derivation. Bottom-left: From FDTD simulation, same scale. Top-right: Difference (FDTD – analytic), same scale. Bottom-right: Difference, scale ten times more narrow.

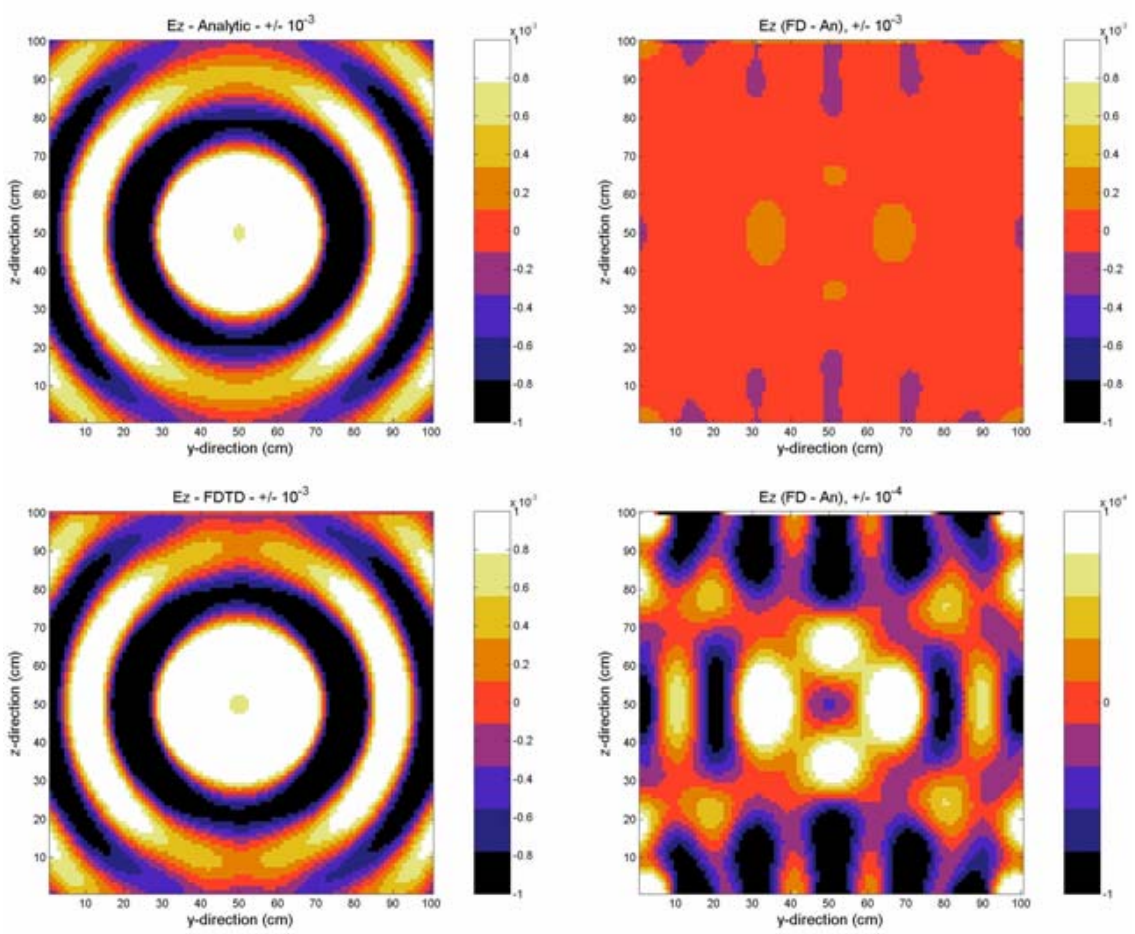


Figure 9: Electric field oriented in the z-direction (Ez). Top-left: From analytic derivation. Bottom-left: From FDTD simulation, same scale. Top-right: Difference (FDTD – analytic), same scale. Bottom-right: Difference, scale ten times more narrow.

5.3 Monopole Antenna

As the code was originally designed for the ground penetrating radar problem, the original code featured built-in monopole antenna design. The antenna's geometry can be found in the User Manual in the appendix. Figure 10 shows some images from a simulation using this feature.

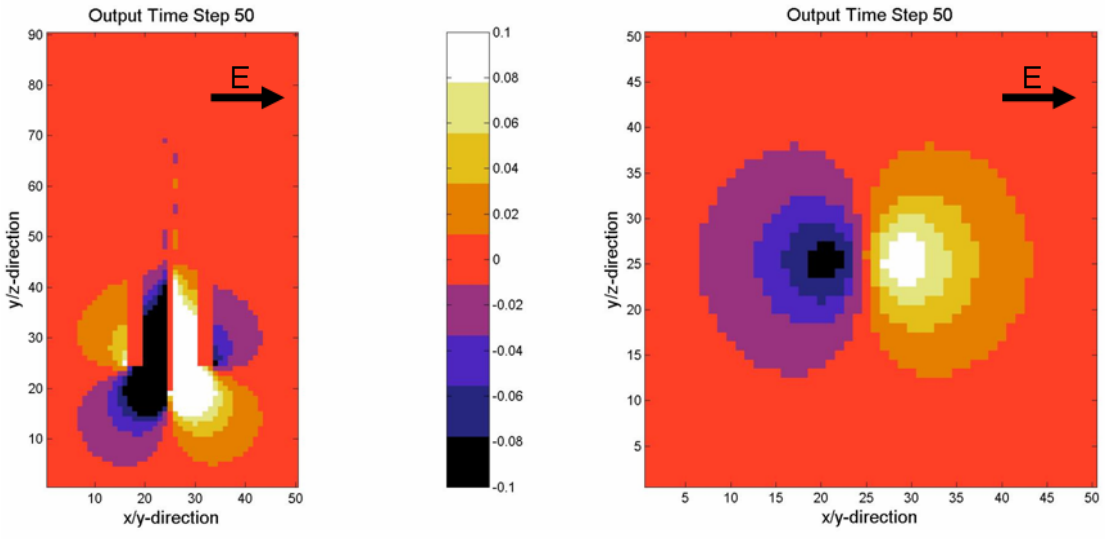


Figure 10: Monopole antenna simulation images. Left: Vertical (y-direction) slice through the middle of the antenna. Right: Horizontal (z-direction) slice beneath the antenna. The arrows indicate the orientation of the field component being displayed.

Both images are from the same time step of the same simulation. Both show the electric field in the x-direction. The image on the left shows a vertical (y-direction) slice through the middle of the antenna; the image on the right shows a horizontal (z-direction) slice beneath the antenna. This pairing shows the general form of the radiation pattern and the rotational symmetry in it.

5.4 Spiral Antenna

A key feature of the new code is its ability to easily handle arbitrary material distributions using material input files. This feature was used to simulate a spiral antenna of interest to our ground penetrating radar group. The spiral was modeled in a MatLab program, printed out to a text-based material input file, and read into a simulation. Other aspects of the material distribution were handled using the simulation's parameter input file. The images in Figure 11 show the pulse slightly counterclockwise past the axis of the field component shown.

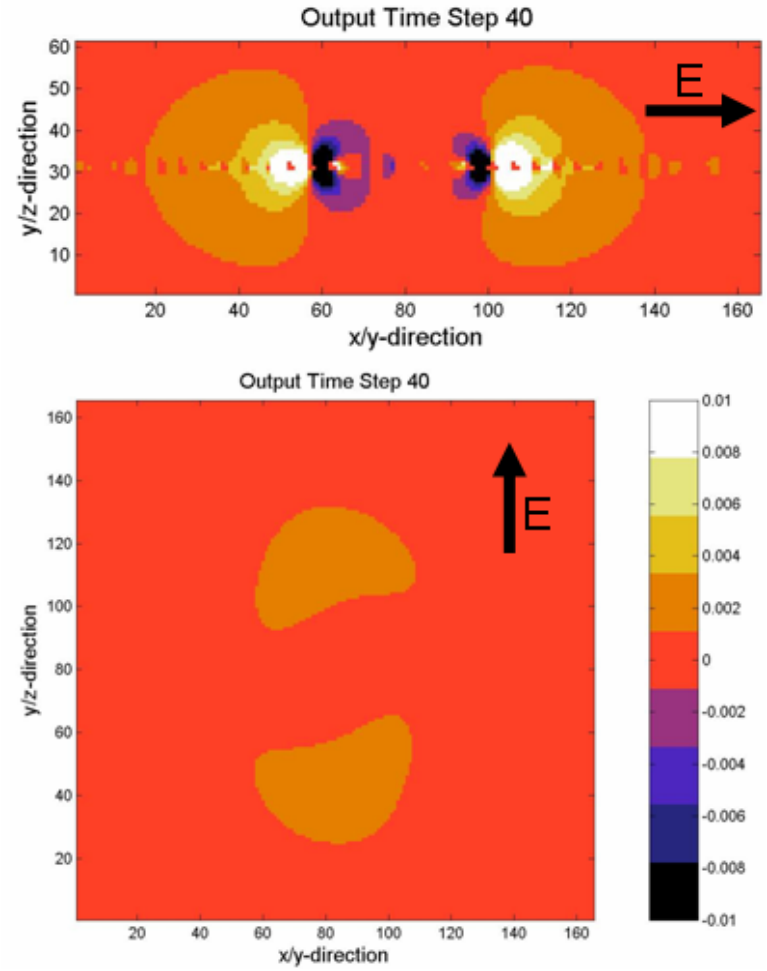


Figure 11: Spiral antenna simulation images. Top: Slice cutting through the center of the spiral antenna perpendicular to the antenna. Bottom: Slice parallel to the antenna, beneath it. The arrows indicate the orientation of the field component being displayed.

5.5 Antennae Designed By High School Students

An important aspect of the new code is that it is easy to use by a wide range of users. To demonstrate this, the code was presented to several local high school students. They were given a brief crash course in electromagnetics, the FDTD technique, and how to run simulations with the code presented here. Then, with some supervision and assistance, they designed their own antennae and ran simulations with them. While, due to lack of technical knowledge, they needed some assistance determining what parameters to input, they successfully handled the antenna design and parameter input processes. The images they produced clearly retain the geometries of the original antennae, demonstrating the facility with which the code can create arbitrary antenna shapes.

Figure 12 shows the process that the students went through to produce simulations using their antennae.

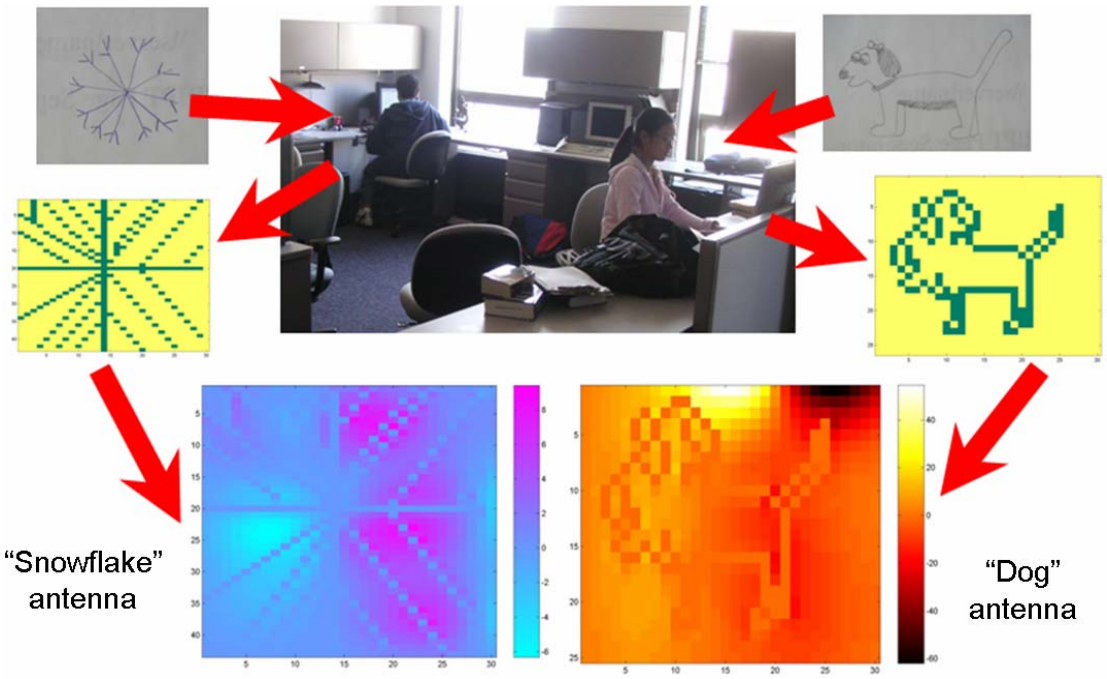


Figure 12: English High School students designing and simulating antennae. Top-left and top-right: Initial hand-drawn sketches of the desired antenna geometries, a "snowflake" antenna and a "dog" antenna. Top-middle: The two students, Robin Canale (left) and Ryda Chea (right), both sophomores at English High School (Boston Public Schools, Boston, MA, USA), creating discretized versions of these sketches using a text editor. Middle-left and middle-right: The discretized antennae. Bottom-left and bottom-right: Electric field results from simulations using these antennae.

Chapter 6:
Future Work

6.1 Introduction

The work presented here represents a large improvement to the original code in terms of how easily and effectively it can be used and developed. However, as with any research project, there are several further improvements that could be made. These include improvements to the code's frequency dispersion model, adding a graphical user interface, visualization programs for open source software, expanding material type and object shape libraries, and linking the code to improved computer hardware.

6.2 Frequency Dispersion Model Improvements

Our code's frequency dispersion model is one of our code's main advantages over other FDTD programs. However, the current code requires users to derive the model's dispersion parameters for any materials that aren't already built in. This can be a difficult task because many users won't be familiar with the model and because the calculation itself is difficult. Thus, a great improvement would be to build the calculation into the code itself, so that users could simply enter the more familiar parameters that the dispersion calculation is based on. Members of our group are currently working on this problem [Jalalinia 2006].

6.3 Graphical User Interfaces

Another feature that could make our code easier for some users is a graphical user interface (GUI) for the data input. The current code's command window interface has several disadvantages. First, it requires users to enter input sequentially and does not allow for them to go back and correct mistakes in input already entered. This problem is addressed by the code-generated input file, but this is an imperfect solution. It requires users to exit the simulation, adjust the input file, and paste the parameters in this adjusted input file into a new simulation. A GUI would improve on this by permitting users to correct their mistakes without going through all of this.

However, unless designed properly, such a GUI could require users to enter the full set of parameters for each simulation manually, a tedious process whose avoidance is one of the current code's strengths. One possible solution to this issue would be a code that uses either a GUI or the current command window. The GUI could even produce the same input file as the current command window, which could then be pasted into the command window, permitting users to take advantage of the command window's speed of use.

Similarly, a GUI for designing material distributions, comparable to those offered in some commercial FDTD packages, would greatly streamline the process of designing material distributions. Currently, this process is one of the most time-consuming parts of running simulations with our code. While the building of background media and the adding of built-in objects is reasonably efficient, these features are restricted to a handful of built-in geometries. Material files enable users to design any material distribution they want, but the code provides no easy way of designing them. By allowing the user to "color in" the material geometry by hand or place objects in the computational domain

via a drag-and-drop system, a GUI could greatly streamline the process of custom designing material distributions.

A final way in which a GUI could make the code better would be in a system for viewing results. Currently, results are written to text files, which must be viewed using programs other than the one that produced them. While a set of MatLab programs has been written to view simulation results, this requires users to own MatLab and know how to use it. While this may not be a problem for many code users, a system which automatically displays results would greatly help others and would make using the code more productive for everybody.

6.4 Visualization Programs For Open Source Software

An alternative to a GUI for results visualization would be a set of programs similar to the MatLab programs presented here, except written for free, open source software such as Octave or SciLab. These software programs share much of the same functionality as MatLab and may be adequate for the results visualization required here. If this is indeed the case, then users seeking to avoid purchasing MatLab could use these programs to visualize results. The author wrote the programs presented here in MatLab instead of in one of the free formats only due to existing acquaintance with and access to MatLab.

6.5 Material Type And Object Shape Libraries

Another area for future work is in the expansion of the built-in material type and object shape libraries. This work would be particularly useful if the problems that users will want to use the code for can be predicted. An expanded material library would be

especially valuable if the soil dispersion approximation is not built in. An expanded object shape library would be especially valuable if no GUI or other improved system for designing material distributions is developed.

6.6 Linking To Improved Computer Hardware

A main disadvantage of the FDTD technique is its large memory requirement. Even with advances in memory capacity and affordability, it is still relatively common for memory limitations to prevent the running of desired simulations.

One way code performance could be improved is by linking it to computer hardware designed for FDTD simulations. One example of this is through a programmable computer chip such as the FPGA chip customized for optimal FPGA simulations under development at Northeastern University by Leeser and Chen [Chen 2003, Chen 2004]. Another possible route for improving computer hardware is distributed memory systems.

Bibliography

Berenger, J.-P., “A perfectly matched layer for the absorption of electromagnetic waves”, J. Computat. Phys., vol. 114, p. 185-200, Oct. 1994.

Chen, W., Kosmas, P., Leeser, M., Rappaport, C. "An FPGA Implementation of Two-Dimensional Finite-Difference Time-Domain (FDTD) Algorithm." In Seventh Annual Workshop on High Performance Embedded Computing (HPEC2003). pp. 105--106. September 2003.

Chen, W., Kosmas, P., Leeser, M., and Rappaport, C., "An FPGA Implementation of the Two-Dimensional Finite-Difference Time-Domain (FDTD) Algorithm." In Twelfth ACM International Symposium on Field-Programmable Gate Arrays (FPGA2004), February 2004, pp. 213-222.

Curtis, J., “Dielectric Properties of Soils: Various Sites in Bosnia”, U.S. Army Corps of Engineers, Waterways Experimental Station Data Report, 20 August 1996.

Dijkstra, E., "Go To Statement Considered Harmful", Communications of the Association for Computing Machinery, Vol. 11, No. 3, pp. 147-148, March 1968.

Enquist, B. and Majda, A., “Absorbing boundary conditions for the numerical simulation of waves”, Mathematical Computat., vol. 3, p. 629-651, 1977.

Farid, M., Alshawabkeh, A., Rappaport, C. and Kosmas, P., “DNAPL Detection Using Cross-Well radar,” 4th International Congress on Environmental Geotechnics, August 11-15, 2002, Rio de Janeiro, Brazil.

Farid M., Alshawabkeh A., Rappaport C., "Validation and Calibration of a Laboratory Experimental Setup for Cross-Well Radar in Sand", Geotechnical Testing Journal, Vol 29, Issue 2, p. 158-167, March 2006.

Hipp, J., “Soil electromagnetic parameters as functions of frequency, soil density and soil moisture”, Proc. IEEE, vol. 62, p. 98-103, Jan. 1974.

Hunsberger, F., Luebbers, R., and Kunz, K., “Finite-difference time-domain analysis of gyrotropic media-I: Magnetized plasma”, IEEE Trans. Antennas Propagat., vol. 40, p. 1489-1495, Dec. 1992.

Jalalinia, M., Bishop, E., and Rappaport, C., "Modeling FDTD Wave Propagation in Dispersive Media Using Four-zeros Conductivity Function", 17th CDSP Research Workshop, March 24, 2006.

Kosmas, P. and Rappaport, C., "A Simple Absorbing Boundary Condition for FDTD Modeling of Lossy, Dispersive Media Based on the One-Way Wave Equation", IEEE Trans. Antennas Propagat, Vol. 52, No. 9, p. 2476-2479, Sept. 2004.

Luebbers, R., Hunsberger, F., Kunz, K., Standler, R., and Schneider, M., “A frequency-dependent finite-difference time-domain formulation for dispersive materials”, IEEE Trans. Electromagn. Compat., vol. 32, p. 222-227, Aug. 1990.

Luebbers, R., Hunsberger, F., and Kunz, K., “A frequency-dependent finite-difference time-domain formulation for transient propagation in plasmas”, IEEE Trans. Antennas Propagat., vol. 39, p. 29-34, Jan. 1991.

Luebbers, R. and Hunsberger, F., “FDTD for Nth-order dispersive media”, IEEE Trans. Antennas Propagat., vol. 40, p. 1297-1301, Nov. 1992.

Mur, G., "Absorbing boundary conditions for the Finite-Difference Approximation of the time-domain electromagnetic-field equations", IEEE Trans. Elec. Comp, Vol. 23, No. 4, p. 377-382, Nov. 1981.

Peng, J., and Balanis, C.. “A generalized reflection-free domain-truncation method: Transparent absorbing boundary”, IEEE Trans. Antennas and Propagation, 46(7), July 1998.

Rappaport, C., and Winton, S., “Modeling dispersive soil for FDTD computation by fitting conductivity parameters”, in 12th Annu. Rev. Progress Appl. Computat. Electromagn. Symp. Dig., Mar. 1997, p. 112-118.

Rappaport, C., Wu, S., and Winton, S., “FDTD Wave Propagation in Dispersive Soil Using a Single Pole Conductivity Model”, IEEE Trans. Magnetics, Vol 35, No. 3, p. 1542-1545, May 1999.

Rappaport, C., "A color map for effective black-and-white rendering of color-scale images", IEEE Antennas and Propagation Magazine, Vol. 44, No. 3, p. 94-96, Jun 2002.

Rappaport, C., Bishop, E., and Kosmas, P., “Modeling FDTD wave propagation in dispersive biological tissue using a single pole Z-transform function”, in Proc. Int. Conf. IEEE Engineering in Medicine and Biology Society, Cancun, Mexico, Sept. 2003.

Sullivan, D., “Frequency-dependent FDTD methods using Z transforms”, IEEE Trans. Antennas Propagat., vol. 40, p. 1223-1230, Oct. 1992.

Sullivan, D., “Nonlinear FDTD formulations using Z transforms”, IEEE Trans. Microwave Theory Tech., vol. 43, p. 676-682, Mar. 1995.

Taflove, A. and Hagness, S., “Computational Electromagnetics: The Finite-Difference Time-Domain Method”, Boston, MA: Artech House, 2000.

Talbot, J., Rappaport, C., and Kosmas, P., “An efficient Mur-type ABC for lossy scattering media”, in Proc. PIERS, July 2000.

Weedon, W. and Rappaport, C., “A General Method for FDTD Modeling of Wave Propagation in Arbitrary Frequency-Dispersive Media”, IEEE Trans. Antennas Propagat, p. 401-410, March 1997.

Yee, K., "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media", IEEE Trans. Antennas Propagat. Vol. 14, p. 302-307, May 1966.

Young, J., “Propagation in linear dispersive media: Finite difference time-domain methodologies,” IEEE Trans. Antennas Propagat., vol. 43, p. 422-426, Apr. 1995.

Appendix:

User Manual

The following is the User Manual designed for the code presented here. Some formatting may be inconsistent due to differences in formatting standards for the Manual and for this thesis.

User Manual

FDTD & Code Basics

The code is a three-dimensional finite difference time domain (FDTD) electromagnetics simulation. It discretizes the Maxwell equations within a three dimensional rectangular prism of space (the "computational domain") and a finite period of time. The material distribution within the computational domain is static: it is the same at all points in time. Each simulation requires as input the distribution of materials within the computational domain as well as values for an electric field excitation during the time period. It also requires that the electric and magnetic field values within the computational domain be zero at the beginning of the time period. The code then calculates electric and magnetic field component values (i.e. values of electric and magnetic fields in the x, y, and z directions) at each point in the computational domain for each point in time. As specified by the user, field component values and parts of the material distribution can be printed to data files ("output files") on the computer's hard drive.

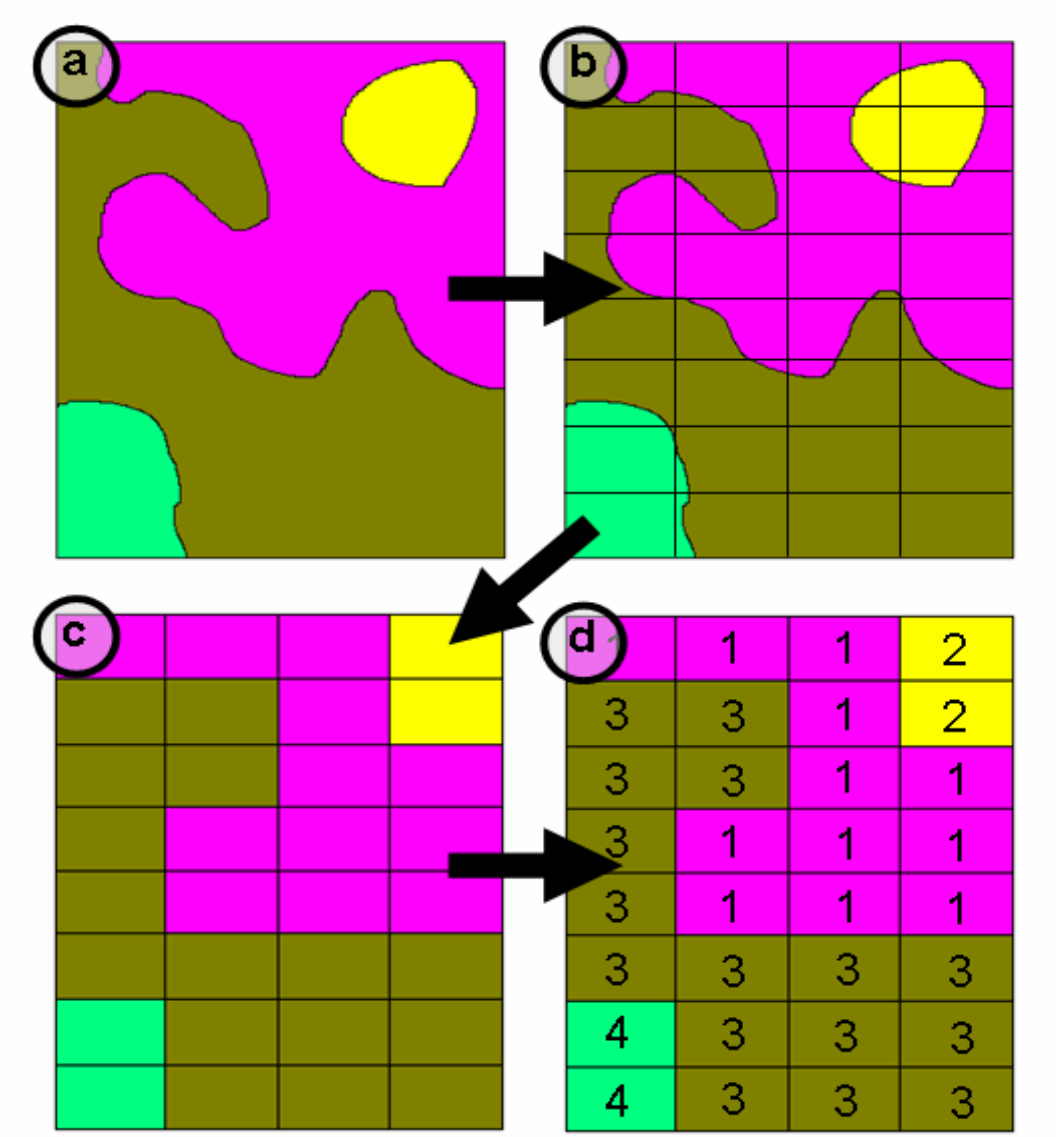
The code consists of two separate Fortran files: nufdtd3d.f and nufdtd3d_params.f. The first file, nufdtd3d.f, contains the main program. It never needs to be modified by the user. The second file, nufdtd3d_params.f, contains definitions of parameters and other variables used in the main code. Certain parts of it may occasionally need to be modified by the user. Any time the parameters file is changed, the main code must be recompiled, built, and executed. Otherwise, the same executable file can be used for any simulation.

A suite of companion MatLab codes is also included:

- fc_read.m: Reads in field component slice sequences, i.e. the data files for a 2D slice of specific field component at various points in time.
- fc_animate.m: Animates a field component slice sequence already read into MatLab.
- fc_animate2.m: Similar to fc_animate.m but adjusted to work with fc_mv.m.
- fc_mv.m: Reads in, animates, and creates .avi movie files for a collection of field component slice sequences.
- fc_jpg: Plots and prints a .jpeg image file of a field component slice already read into MatLab.
- mt_readplot.m: Reads and plots a material slice.
- pulse_plot: Plots pulses of various shapes given pulse parameters.

Finite Difference Discretization

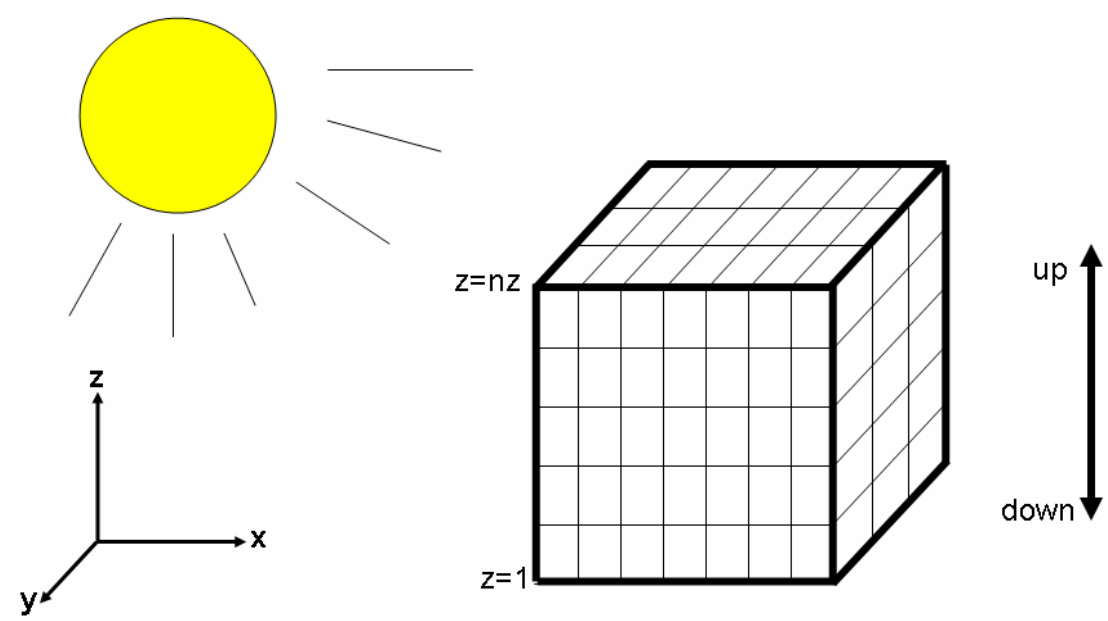
In all FD simulations (including both FDTD and FDFD), a 2D or 3D region of space is approximated as a uniformly-spaced 2D or 3D grid of "grid cells" or "grid points". The grid need only be uniform within the same direction: the number and spacing of grid points in different directions need not be the same. For example, a simple discretization of a 2D region containing four materials could be:



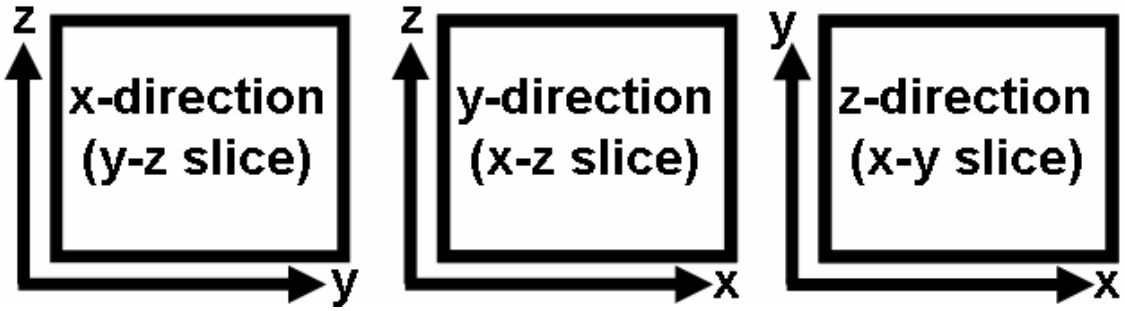
(a) shows the original material distribution, with each color representing some different material. (b) shows an FD grid superimposed on (a). The number and spacing of the grid points in the two directions (vertical and horizontal) were intentionally made different to demonstrate that they need not be the same. (c) shows an FD discretization of the material distribution based on this FD grid. It simply fills in each rectangle with the color that is predominant within the rectangle in (b). (d) adds a material numbering scheme. FD computer codes generally represent materials with number labels as seen here.

Coordinate Systems

z is the vertical direction; x and y are arbitrary horizontal directions. In the code, z counts upwards, i.e. z=1 is closer to the center of the Earth, and z=nz is closer to the sky. For this document, we will have x pointing to the right and y pointing towards the reader:

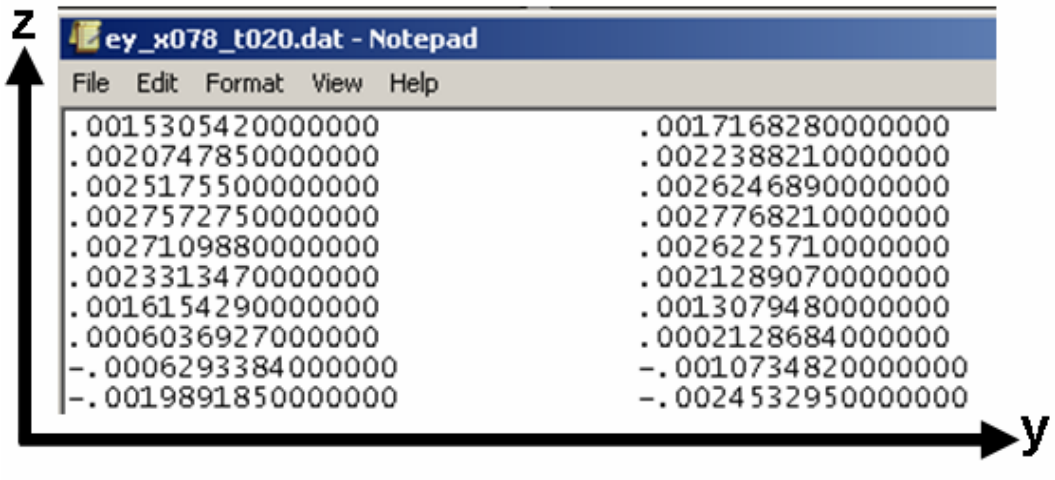


The Fortran code and the MatLab companion codes all make extensive use of 2D slices of the 3D grid. For all of these, standard Cartesian coordinates (as opposed to i-j matrix coordinates) are used:

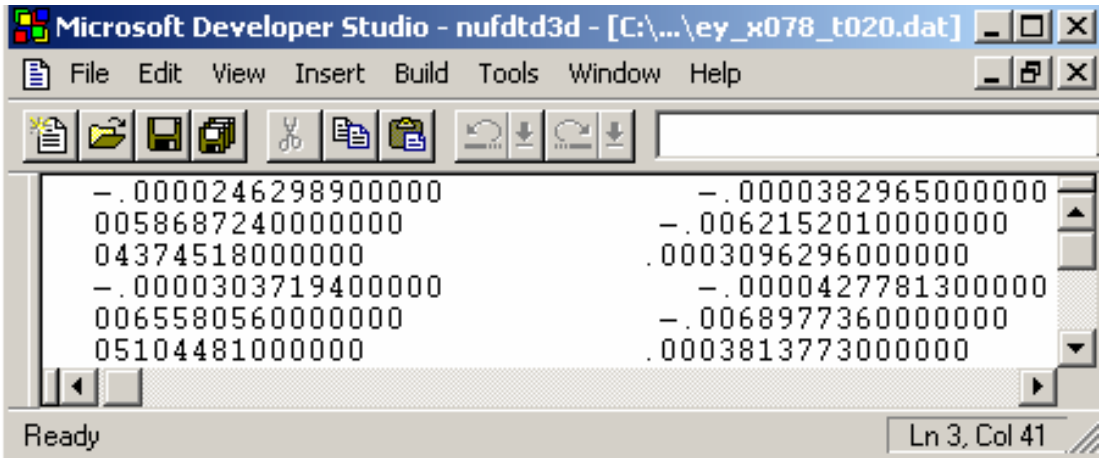


Text Editor Coordinate Systems

Care must be taken to ensure consistency in coordinate systems when converting matrices between data types. All Fortran matrices use the standard Cartesian coordinate systems explained below. When matrices are printed to data files, they are written so that they follow this convention when viewed with a text editor. For example, the following x-direction field component slice has the coordinate system shown here:



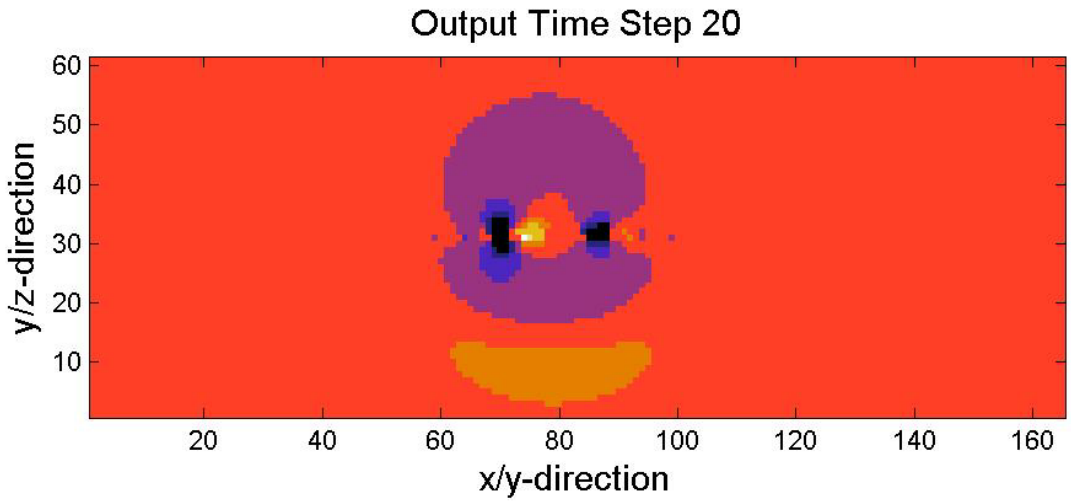
However, text editors typically use i-j coordinate systems in their (line, column) or (row, column) designations, as seen here in the bottom-right corner of this screenshot from Microsoft Developer Studio:



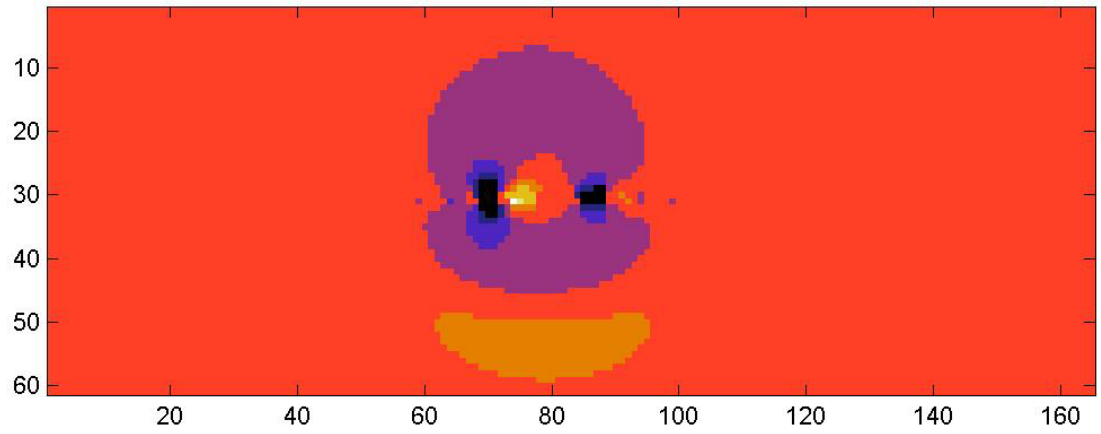
As text editors can be a useful means of analyzing files, such as for determining where to place excitation points for antennae built from material input files, understanding text editor coordinate systems can be important.

MatLab Coordinate Systems

Using the MatLab companion codes, this file produces an image oriented in the same fashion:



The companion codes use MatLab commands to rearrange the data ("flipud") and image ("axis xy") to achieve this. Plotting the above image without these commands gives an image with the same appearance but with a different coordinate system:



The bottom image's coordinates, printed without altering the data, match those of the actual matrix, as long as i-j matrix coordinates (vertical, horizontal) are used, as seen in the following lines. (Here, gold is positive and blue is negative.)

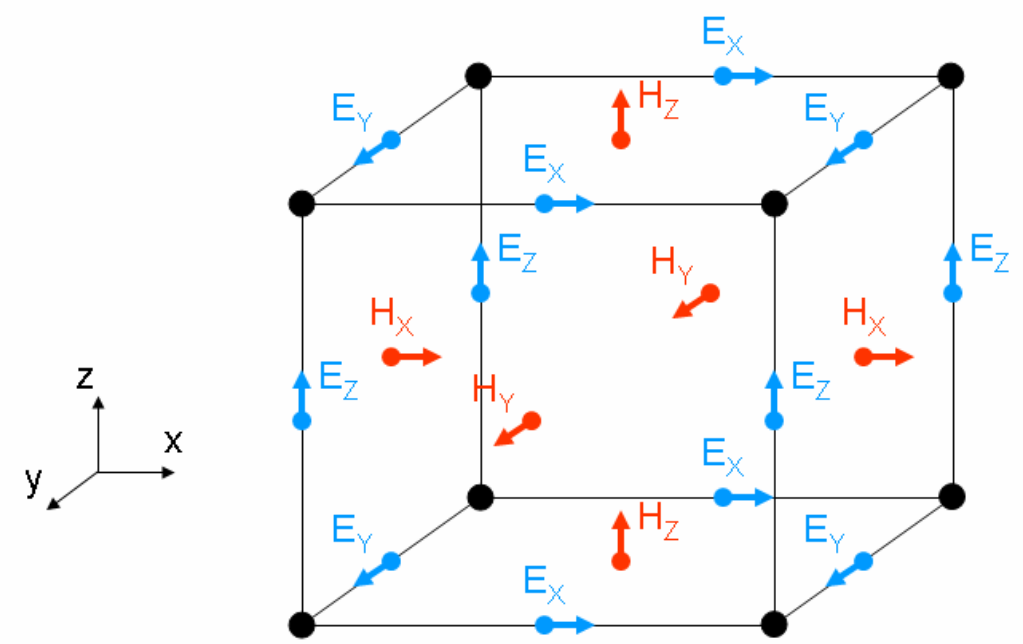
```
>> f(80,50)
??? Index exceeds matrix dimensions.

>> f(50,80)

ans =
```

```
    0.0116

>> f(10,80)

ans =

   -0.0122
```

Field Component Locations

In FDTD, electric and magnetic field vectors are broken up into components in the x, y, and z directions. These field components are all defined at each grid point. Thus, there are a total of seven separate 3D grids: Ex, Ey, Ez, Hx, Hy, Hz, and the material distribution. The field component grids are actually 4D because they also vary with time. This FDTD code does not handle time-varying material distributions, although such a scheme is possible with FDTD.

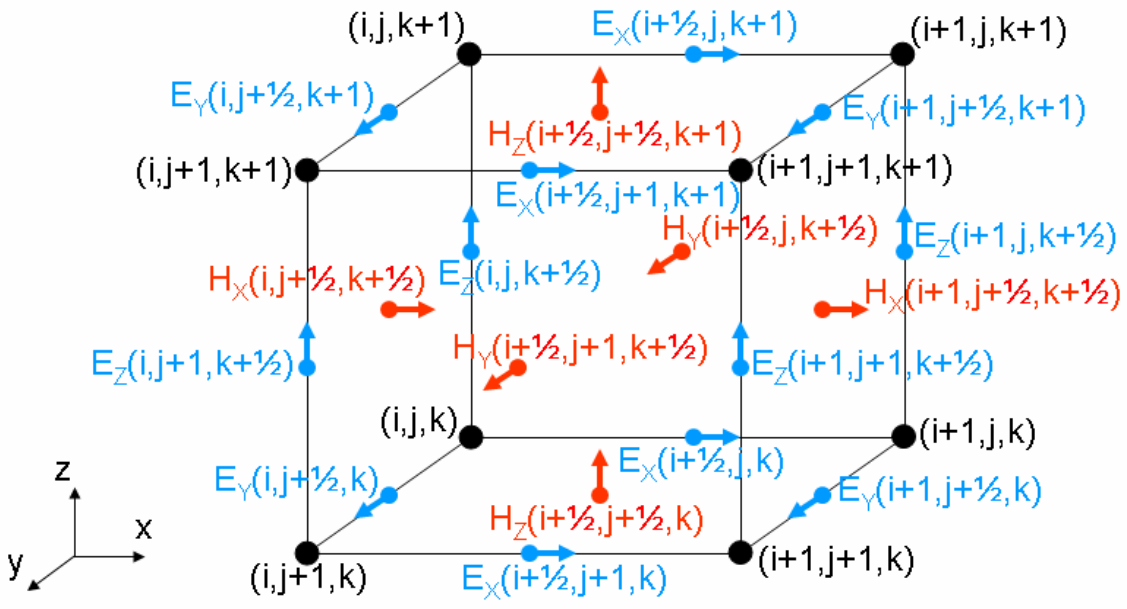
For subtle reasons not discussed here, field component points are not located at the same places as each other or the material points. The electric field components are shifted half of a grid cell in one direction: the same direction as the field component is oriented. Thus, Ex is shifted half of a grid cell in the x direction. The magnetic field components are shifted half of a grid cell in two directions: the directions that the field component is not oriented in. Thus, Hx is shifted half a grid cell in the y and z directions. For both electric and magnetic fields, the shifts are in the positive direction, i.e. away from the origin (1,1,1). A common representation of this is the Yee cube, named for Kane S. Yee, FDTD's main creator:



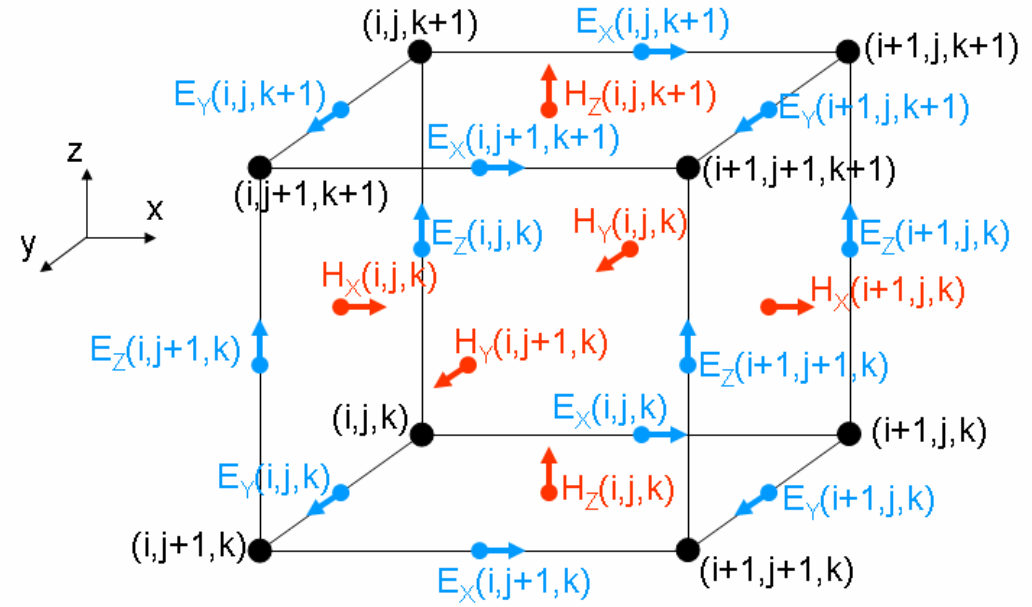
In this diagram, the black dots represent locations of material points, blue arrows represent electric field components, and red arrows represent magnetic field components.

Field Component Coordinates

The same Yee cube can be redrawn with material points' and field components' coordinates, assuming that the cube runs from point (i,j,k) to point (i+1,j+1,k+1):



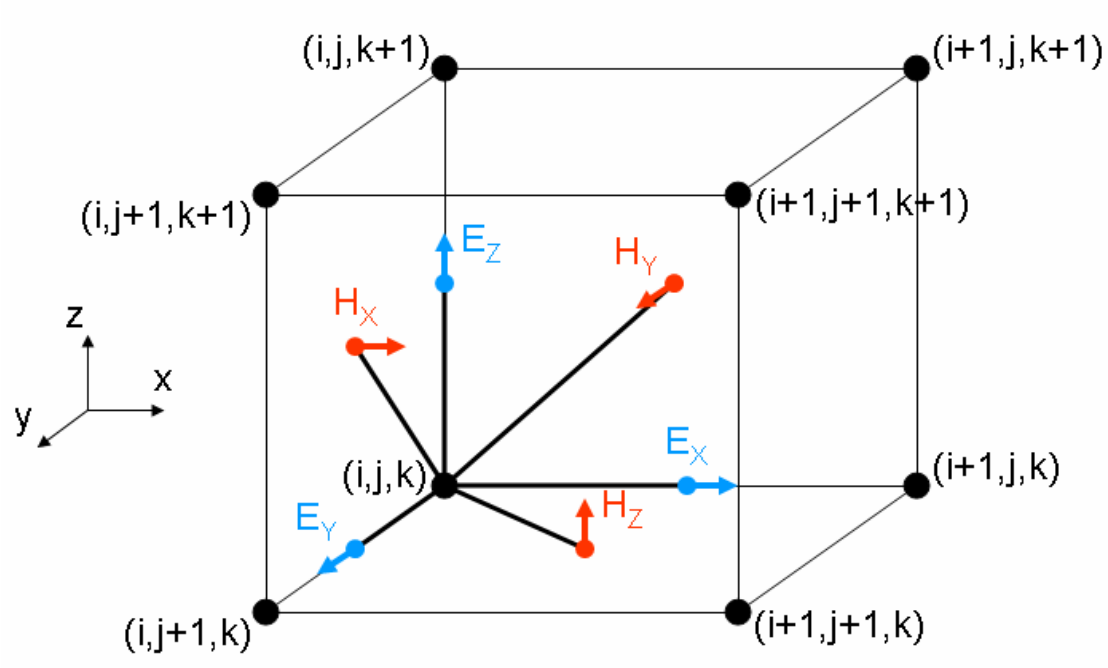
Clearly, the electric and magnetic field components' coordinates contain non-integer values. However, code matrices (and indeed, matrices in general) require integer indices. Thus, a scheme is required to convert the non-integer values to integer values. We simply subtract ½ from all non-integer matrices, giving the shifted Yee cube:



Field Components – Material Point Relationships

Using the previous Yee cube images, the location of a field component can be determined from its grid coordinate. For example, Ex(11,23,4) is between the material points (11,23,4) and (12,23,4); Hx(11,23,4) is between the material points (11,23,4), (11,23,5), (11,24,4), and (11,24,5).

While field components are located between multiple material points, for the code's calculations, each field component is linked to a single material point: the grid point in the direction of the origin, i.e. the grid point found by subtracting ½ from all of the shifted coordinates:



Thus, the code behaves as if the field components linked to a material point are in regions composed of that point's material. This is done to avoid the memory-inefficient measure of creating separate matrices of material points for each field component.

Overview Of How To Perform A Simulation

There are several steps necessary for performing a successful simulation. These are:

1. Run the code. This starts the simulation and opens the command window. How to run the code is explained below.
2. Input the simulation parameters into the command window. Once the parameters have been inputted, the simulation will run automatically. How to input is explained in the "Using The Command Window" section; the input parameters are explained in the "Input Parameters" section.
3. View the output files. Output files are the simulation's results. How to view the results is explained in the "Viewing Output Files" section.

How To Run The Code

The code was developed using the software application Microsoft Developer Studio Fortran PowerStation 4.0. If further work is done using different software, then instructions for running the code through the software may vary.

No matter what software is used, the code can be run by double-clicking on its executable file (nufdtd3d.exe). If the code was previously ran through PowerStation, then the executable file may appear in the "Debug" folder.

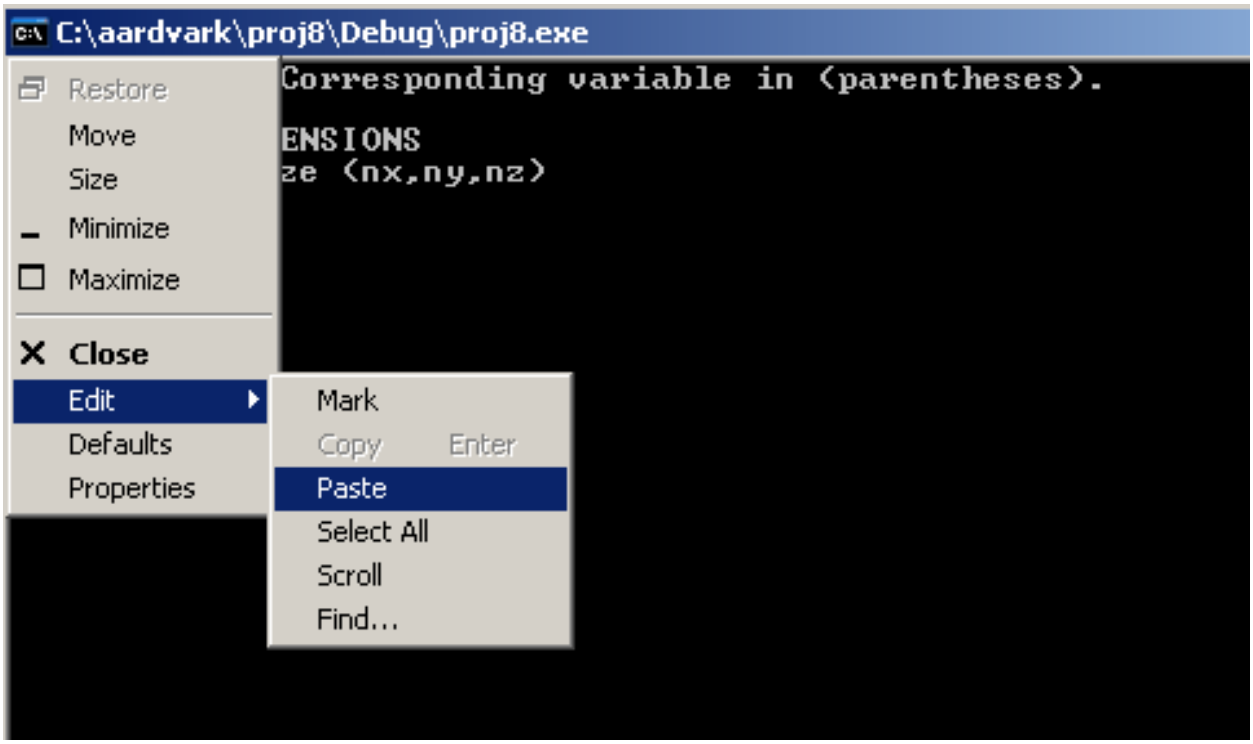
If changes to the code have been made since the previous time it was run, then the code needs to be recompiled. The code is automatically recompiled if it is executed from PowerStation. To execute the code from PowerStation, choose "Execute nufdtd3d.exe" from the Build menu at the top of the screen or hit Ctrl+F5.

Using The Command Window

Command Window Basics

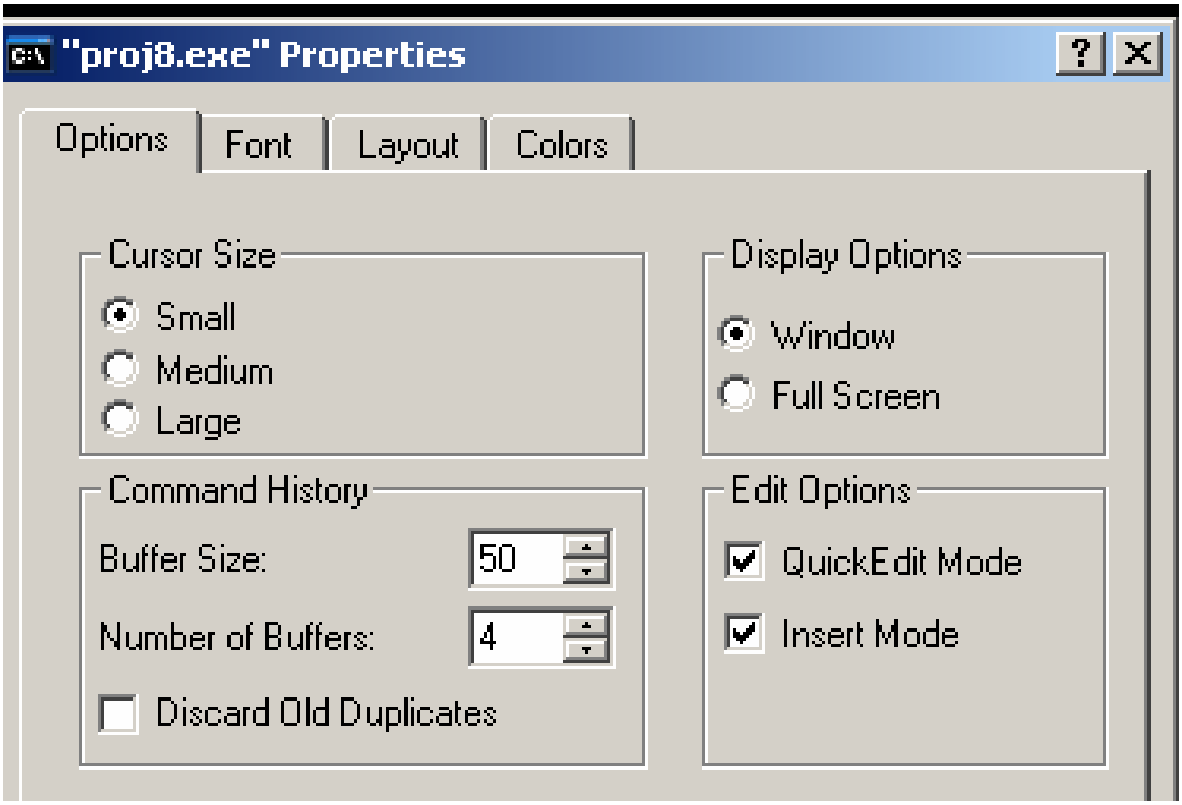
Input in the code is entered via a text-based command window interface. This interface asks the user a series of questions, which the user types in and then hits Enter to progress to the next question. Answers to previous questions cannot be changed. Instead, to correct earlier errors, the command window must be closed, the program must be re-run, and all of the data must be re-entered from the beginning. Input must also be in a certain format. For example, if a question asks for an integer, no letters can be included in the answer. If the question asks for a real number, letters cannot be included in the answer, except for an optional "e" for "exponent". For example, 2e1 = 2x10^1 = 20.

Input can be copied and pasted from a text file into the command window. The paste command is found by choosing Edit after clicking on the icon in the top-left corner of the command window:



Command Window Basics Continued

Alternatively, if QuickEdit Mode is checked from the Properties box, a paste can be performed by clicking the right mouse button. (Checking insert mode causes inserted text to be inserted between characters instead of overwriting them.)



Input Files

A good way to streamline the process of re-entering the same or similar sets of input is to create and use an "input file" containing some or all of the input which, if properly formatted, can then be copied and pasted in its entirety into the command window. Properly formatted input files include line breaks between lines of input and can include comments after an exclamation mark. (Using an exclamation mark for commenting comes from Fortran syntax.) For example, the input file text

```
4000  ! number of time steps
.1000E-11  ! Time step size (sec)
```

can be inserted into a command window producing

```
*** TIME DIMENSIONS
Enter total number of time steps to run the code
  4000  ! number of time steps
Enter time step size in sec (dt)
To statisfy Courant Condition, need dt <    1.713992E-12
 .1000E-11  ! Time step size (sec)
```

The code creates an input file automatically as input is being entered by the user. Thus, if the user stops entering input after 5 questions, then it will contain exactly those 5 lines of input. This feature is particularly useful when data is initially entered manually, i.e. by typing input in directly instead of copying and pasting it. The input file created appears on the hard drive, in the same directory as the simulation, with the file name "input.txt". Because the code automatically overwrites any pre-existing files named input.txt, it is a good idea to avoid using this file name for other purposes.

Note that the E is for "exponential". Here, aEb = a * 10^b, so, for example, .1000E-11 = 10^{-12}.

Input Parameters

The code groups its input into several sections. For example, the previous image shows part of the "Time Dimensions" input section. The following user manual parts describe each of these input sections in the order in which they appear in the command window:

1) Space Dimensions
2) Time Dimension
3) Create New Materials
4) Background Layers
5) Foreground Objects
6) Material Output
7) Excitation
8) Field Component Output

Sections 1 and 2 define the FDTD discretization. Sections 3, 4, and 5 define the material distribution. Section 7 defines the electric field excitation. Sections 6 and 8 define the portions of the material and field component distributions to save to the hard disk for analysis.

Space Dimensions Input

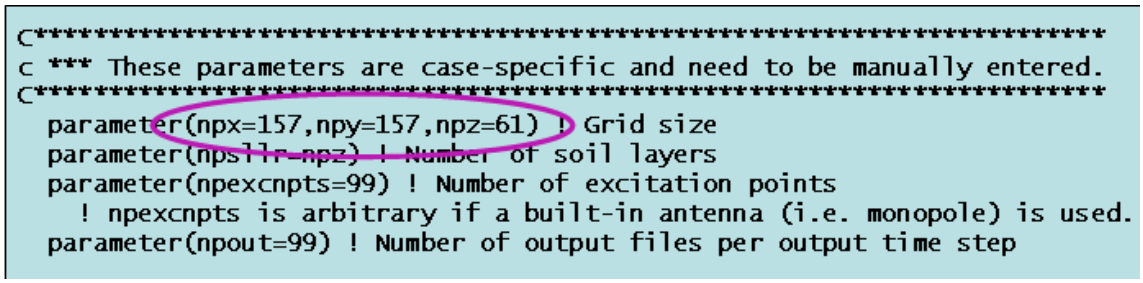
Two sets of numbers are required to define the FD grid:

- grid size in each direction (nx, ny, nz)
- grid cell size in each direction (Δx, Δz, Δz; dx, dy, dz; or delx, dely, delz)

The grid size variables represent the number of grid points in each direction and are unitless quantities. The grid cell size variables represent the size in each direction of one grid cell, i.e. how far apart the grid points are, and are measured in this code in centimeters. (Other FDTD codes may use other units of distance.)

In FDTD, the amount of computer memory (RAM) required to run a simulation is roughly proportional to the grid size. If the grid size is too large for a computer's RAM, then the simulation will use paging file and run extremely slowly. If the grid size is too large for a computer's RAM + paging file, then the simulation will not run.

The code allocates a fixed grid size in each direction, and the grid size cannot be set to anything larger without changing this allocation and recompiling the code. (Smaller grid sizes can be run without changing this and recompiling.) The grid size allocation is changed by changing the grid size numbers at the top of the code's parameters file:



Time Dimension Input

The input for the time dimension is similar to that for the space dimensions because, similar to how it handles space, FDTD discretizes time in uniform intervals or "time steps". There are two values to input:

- Number of time steps to run the code (nts)
- Time step size (dt or Δt)

nts is unitless; dt is measured in seconds. nts * dt = time (in seconds) the code will simulate.

The code runs one time step at a time, scanning through the spatial grid, calculating electric and magnetic field values. Thus, the larger the number of time steps is, the longer the code will take to run. However, the number of time steps has no effect on a simulation's memory consumption, so any computer can run any number of time steps.

In order for an FDTD simulation to run properly, input must satisfy the "Courant condition":

$$c_0 \Delta t < \left(\left(\frac{1}{\Delta x} \right)^2 + \left(\frac{1}{\Delta y} \right)^2 + \left(\frac{1}{\Delta z} \right)^2 \right)^{-1/2}$$

Here, c_0 is the speed of light in free space (vacuum); Δt is time step size; Δx, Δy, & Δz are space step (grid cell) sizes. Using c_0 gives an upper bound for Δt or, alternatively, a lower bound for Δx, Δy, & Δz. If the material distribution does not include free space, then this bound is not tight. It is used to guarantee that the Courant condition will not be violated, no matter what the eventual material distribution is.

In the code, Δx, Δy, & Δz are inputted before Δt, so if the value for Δt entered does not satisfy the Courant condition, the user is prompted to enter a new value for Δt. If instead, the user prefers changing Δx, Δy, & Δz, then that command window must be closed and input entry must be restarted.

Create New Materials Input

Our code comes with several built-in material types, each with a pre-set number label, as well as several pre-set number labels reserved for user-defined materials:

```
1=Free Space
2=Metal (PEC)
3=Dielectric; relative permittivity = 2.3
4=TNT
5=Dielectric modeled as dispersive material (relative permittivity = 2.3)
6=Lossy Puerto Rican soil; tss=20ps; f=1.5GHz
7=Lossy Puerto Rican soil; tss=10ps; f=1.5GHz
8=Lossy Bosnian soil;  tss=20ps; f=1.5GHz
9=Lossy Bosnian soil; tss=100ps; f=1.5GHz
10=Lossy Bosnian soil;   tss=2ps; f=1.5GHz
11=water; tss=50 ps; f=100MHz
12=Bosnian soil,  2.5% water; tss=50ps; f=100MHz
13=Bosnian soil,  5.0% water; tss=50ps; f=100MHz
14=Bosnian soil, 10.0% water; tss=50ps; f=100MHz
15=Bosnian soil, 20.0% water; tss=50ps; f=100MHz
16=Sandy soil,  4.0% water; tss=2ps; f=1.3GHz
17=Sandy soil, 17.0% water; tss=2ps; f=1.3GHz
18=Sandy soil,  4.0% water; tss=20ps; f=1.3GHz
19=Sandy soil, 17.0% water; tss=20ps; f=1.3GHz
20=Sandy soil, 17.0% water; tss=6ps; f=1.3GHz
21=Teflon
22-34=Optional user-defined material
```

When creating new materials, the first input needed is

- number of new materials

Then, for each material, the relevant physical properties must be entered:

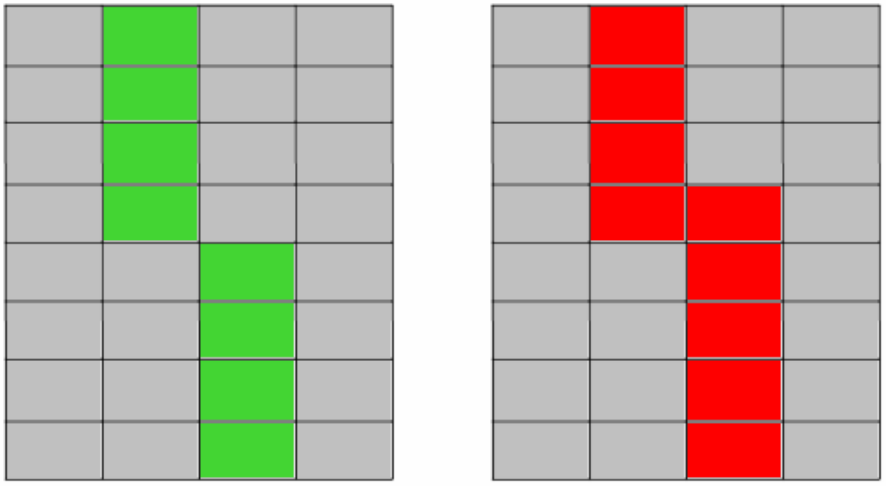
- relative permittivity
- If the material is not frequency dispersive (i.e. its properties do not vary as a function of the frequency of electromagnetic radiation passing through it), then electric conductivity must be entered.
    - If the material is a lossless dielectric, then the conductivity should be set to 0.
- If the material is frequency dispersive, then four special “dispersion variables”, a1, b0, b1, & b2 must be entered.
    - b0 is the same as the conductivity in the non-dispersive case.

Building Material Distributions

A big advantage of FD techniques is that any grid point can contain any material, which allows arbitrary material distributions to be modeled. This does mean that one could build material distributions that are completely unrealistic, such as a layer of air between two soil layers. It is up to the user to ensure that the simulated material distribution is accurate.

While FDTD makes field calculations with arbitrary material distributions routine, building those distributions can be challenging. This code features two techniques for building material distributions: background layers and foreground objects.

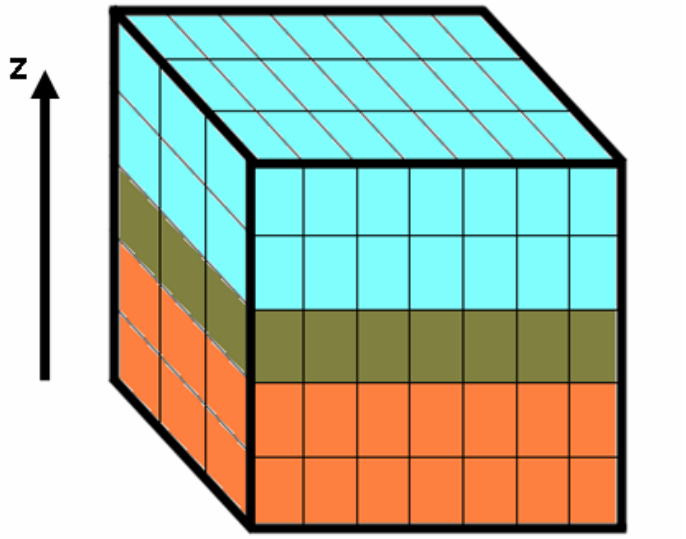
One important point to remember when building material distributions is that FDTD treats features diagonally adjacent as being separate pieces; for a feature to be treated as one continuous piece, it must be connected horizontally/vertically. (i.e., one must be able to move a chess rook along it.) Thus, in the figures below, the green feature on the left will be treated as two separate pieces, whereas the red feature on the right will be treated as one continuous piece:



This point is particularly important when designing antennae because fields can get through metal antenna pieces that are not connected horizontally/vertically, even though this is not physically realistic.

Background Layers Input

The first means of building material distributions is through “background layers”. This feature was initially developed for the code’s ground penetrating radar application, as to simulate stratified regions of ground. Background layers are homogenous rectangular prisms that span the entire the x-y plane:



The first input needed is

- number of background layers. There must be at least one layer and there cannot be more layers than there are grid points in the z direction.

Then, for each background layer, the following input is needed:

- layer thickness (in grid cells). Layers must be at least one grid cell thick and must be small enough that all remaining layers can be at least one grid cell thick.
    - For the top background layer, no thickness input is taken because its thickness is automatically set to the number of remaining grid cells in the z direction.
- material type

A homogenous background can be created by making one layer; this is equivalent to making all layers with the same material.

Foreground Objects Input

Once the layer(s) have been defined, the user can place any number (at least zero) of foreground objects in the material distribution. They are called "foreground" objects because their material values overwrite those of the background layers. Objects come in various shapes and sizes, can be placed anywhere in the grid, and can be of any available material. Each object type is labeled in the code by a number, similar to material type numbers. Available object types include:

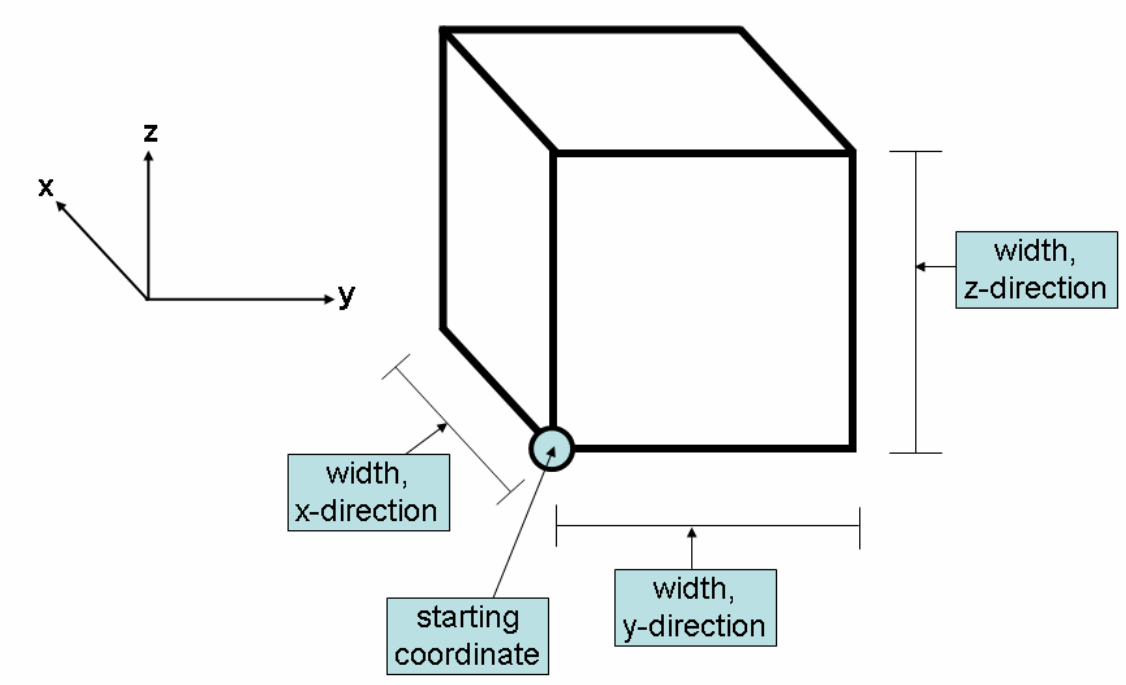
```
0 - Material Input File
1 - Rectangular Prism
2 - Cylinder (Use for landmines)
3 - Sphere
4 - Monopole Antenna
```

Each object type has its own set of input to be entered.

Rectangular Prism Input

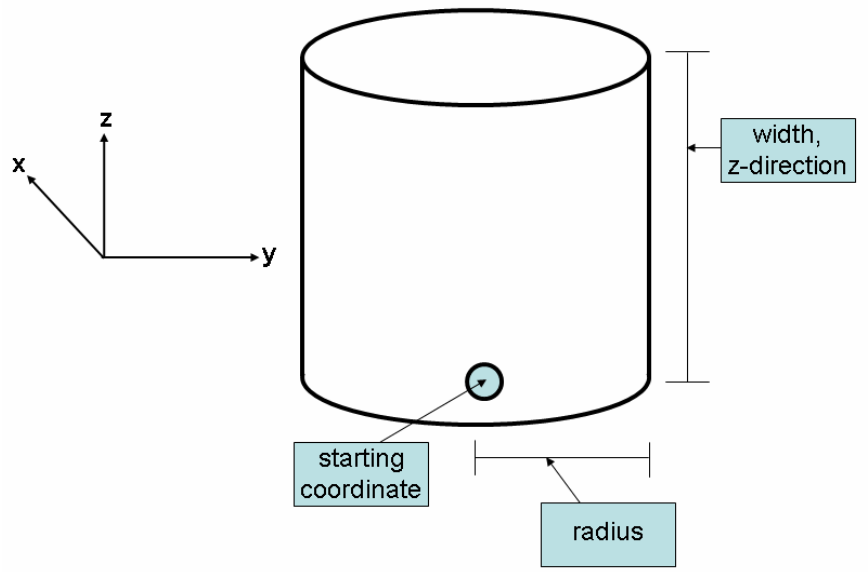
Creates a solid rectangular prism of uniform material. Making a rectangular prism of width (1,1,1) is an easy way to define the material in a single grid point.

- Starting coordinates: coordinates (in grid points) of the corner of the prism closest to the grid point (1,1,1)
- Width in each direction, in grid points
- Material type



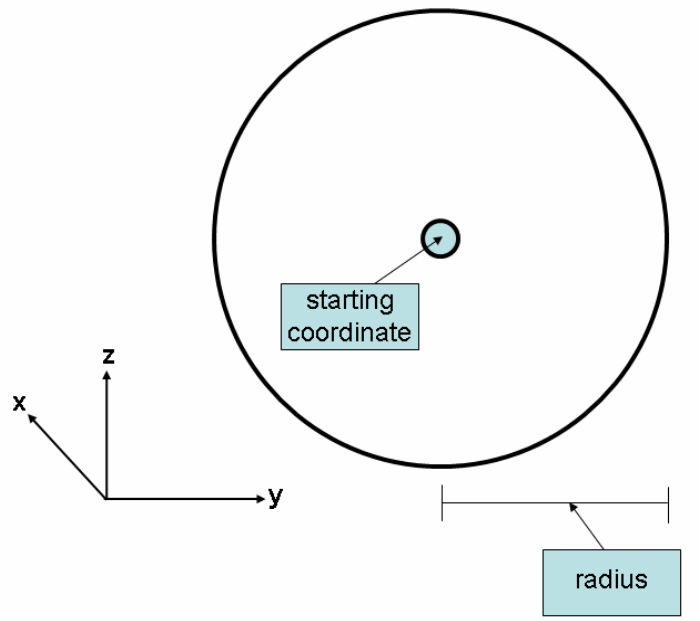
Cylinder Input
Creates a solid cylinder oriented with its parallel planes oriented along the x-y plane. A cylinder made with the TNT material type reasonably approximates some landmines.

- Starting coordinates: coordinates (in grid points) of the circle's center at it's lowest point (closest to z=1)
- Circle radius and vertical height, in grid points
- Material type



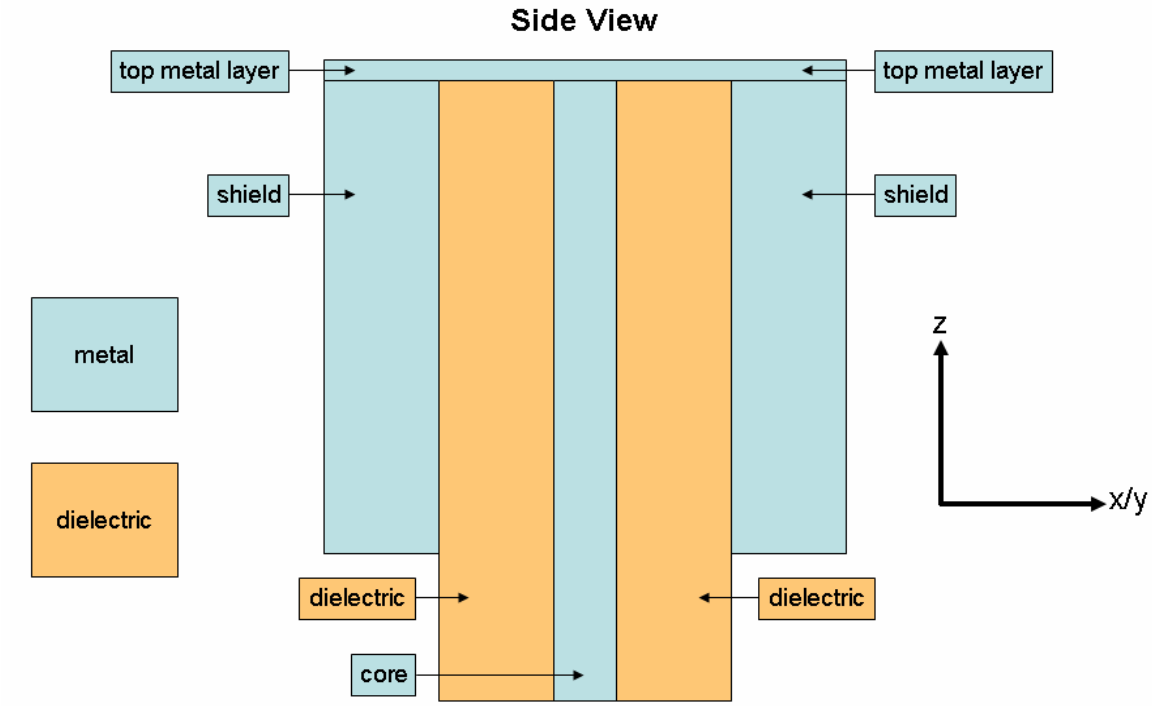
Sphere Input
Creates a solid sphere.

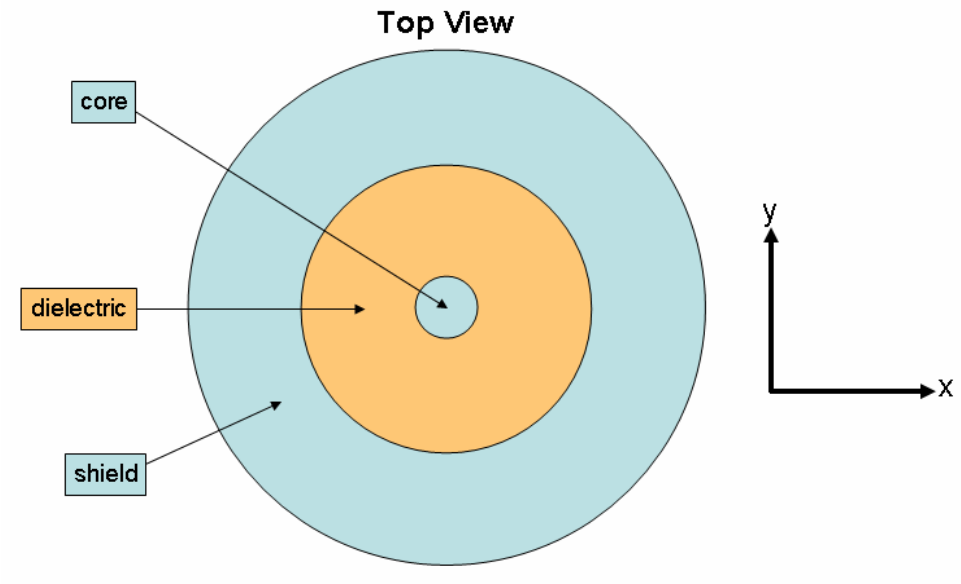
- Starting coordinates: coordinates (in grid points) of the sphere's center
- Sphere radius
- Material type



Monopole Antenna

The monopole antenna, commonly used in ground penetrating radar, is essentially a series of three concentric cylinders. The inner cylinder, the "core", is made of metal. The middle cylinder, the "dielectric", is made of dielectric. The code uses material number 5 (dielectric modeled as dispersive soil) for this. The outer cylinder, the "shield", is also made of metal. The core and dielectric have the same height; the shield's height is smaller. The tops of the three cylinders are all along the same plane. Immediately above this top layer is a layer of metal one grid cell thick:

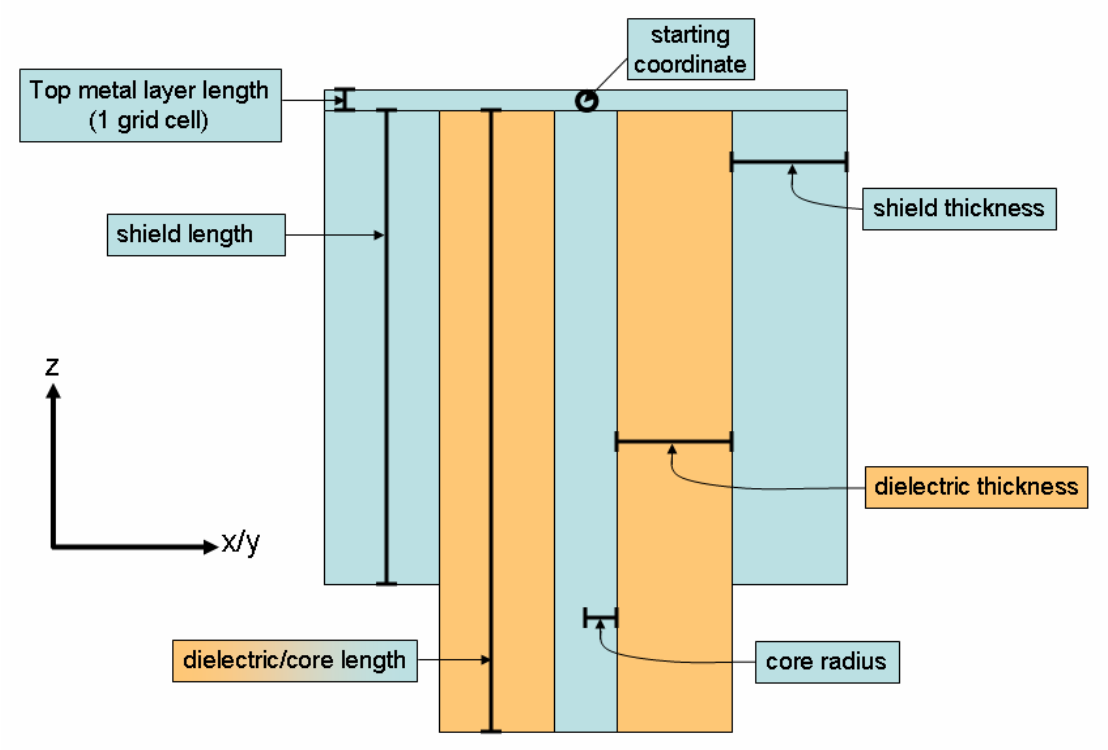




Monopole Antenna Input

To build a monopole antenna, the following input is required:

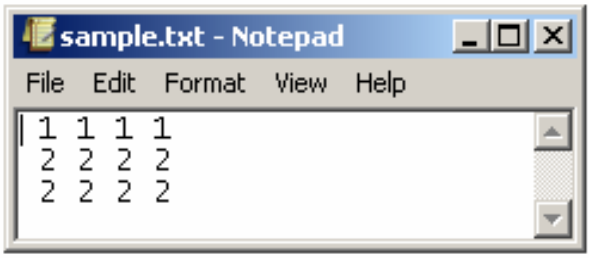
- coordinates (in grid points) of antenna top-center
    - z-direction coordinate is of top metal layer
- core radius, dielectric thickness, and shield thickness, in grid points
- dielectric/core length and shield length, in grid points



Monopole antennae objects can be used in the excitation. Only one monopole antenna can be used in the excitation for any given simulation. When building multiple monopole antenna objects, the last one to be built is the one that can be used in the excitation. If a monopole excitation is selected, that last monopole antenna is automatically excited in the appropriate fashion.

Material Input File

The material input file is a special object type that allows the user to easily define arbitrary material distributions. A material input file is a text file containing a 2D grid of numbers, located in the same directory as the simulation. The numbers must be separated by spaces. The grid must be rectangular, i.e. all rows and all columns must have the same size. (Rows need not have the same size as columns.) The numbers in the file must correspond to material type numbers. However, the code does not check this. It is up to the user to make sure that the correct numbers have been entered.



The above file, “sample.txt”, could be a 4x3 material file containing two rows of material number 2 (metal) below one row of material number 1 (free space).

Material input files have the following specifications:

- Filename: the name of the file, including its extension (e.g., sample.txt)
- Orientation: direction (x, y, or z) for the file to be placed.
- Height: the coordinate for the file to be placed in its direction of orientation.
- Starting point: the two coordinates in the non-height directions of the bottom corner of where the material input file should be placed, i.e. the corner closest to (1,1,1)
- Width: the number of points in the two non-height directions to read. The width cannot be larger than the number of the points in the computer file, but it can be fewer. If it is fewer, the code will start from the beginning of the file and stop reading before reaching the end. The width also cannot be so big that it will not fit in the simulation grid, given the grid size and the material file starting point.

Material Output Files Input

Once the material distribution has been built, parts or all of it can be checked by printing "material output files". These files contain slices of the final material distribution appear in the same directory as the simulation. For example, one could print out the materials at x=19 or z=1. The numbers in the files correspond to the material number labels at the grid points along the slice.

The first input needed is

- number of material output files to print

Then, for each material output file, the following input is needed:

- the direction and location of the slice (i.e., x=23, etc)

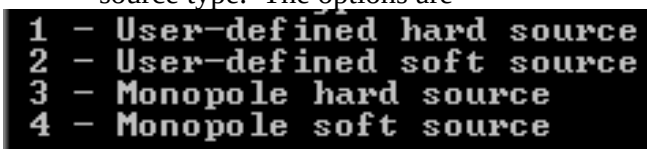
The material distribution then gets printed to a text file in the same directory as the simulation. The file name begins with the letters "mt" (for "material"), followed by an underscore, followed by a letter (x, y, or z) representing the slice direction, followed by a three-digit number representing the slice location, followed by the extension ".dat". Thus, the slice at x=23 will have the file name "mt_x023.dat".

Excitation Input

The code initializes all electric and magnetic field to zero, and without any non-zero field values added, its calculations will produce all zero values. These added values are the "excitation".

The first input needed is

- source type. The options are



In a hard source, the field values at the excitation points are fixed to the values given by the pulse. In a soft source, the field values are the sum of the pulse values and the values previously calculated at the excitation points. Either one may give better results in different circumstances.

User-defined sources are generally chosen when the antenna used for the excitation is built with material input files. In order for a monopole sources to function properly, a monopole antenna must have already been built using the monopole antenna option in the "foreground objects" section. If more than one monopole antenna was built, the last one to be built will be the one used for the excitation.

If a user-defined source (options 1 or 2) is chosen, then the following input is needed:

- number of excitation points

For each excitation point, the following input is needed:

- coordinates. The grid location (i,j,k) of the excitation point.
- directional excitation strengths. The magnitude of the excitation in each direction (x, y, z). These define the magnitude of the x, y, and z components of the excitation's electric field. Generally, numbers in the interval [-1,1] are chosen, although this need not be the case.

Excitation Input: Pulse Shape

Then, for all source types, the following input is needed:

- pulse shape. The options are

```
1 - Narrow-Width Gaussian
2 - Cosine-Modulated Gaussian
3 - Narrow-Width Half-Gaussian
4 - Cosine-Modulated Half-Gaussian
```

For each pulse shape, the following input is needed:

- pulse width, in time steps
- pulse peak time, in time steps

For the cosine-modulated pulse shapes, the following input is also needed:

- pulse frequency, in Hertz

Excitation Input: Pulse Shape Continued

The narrow-width Gaussian is a discretized version of the standard Gaussian pulse:

$$f(n) = e^{-\left(\frac{n-p}{w}\right)^2}$$

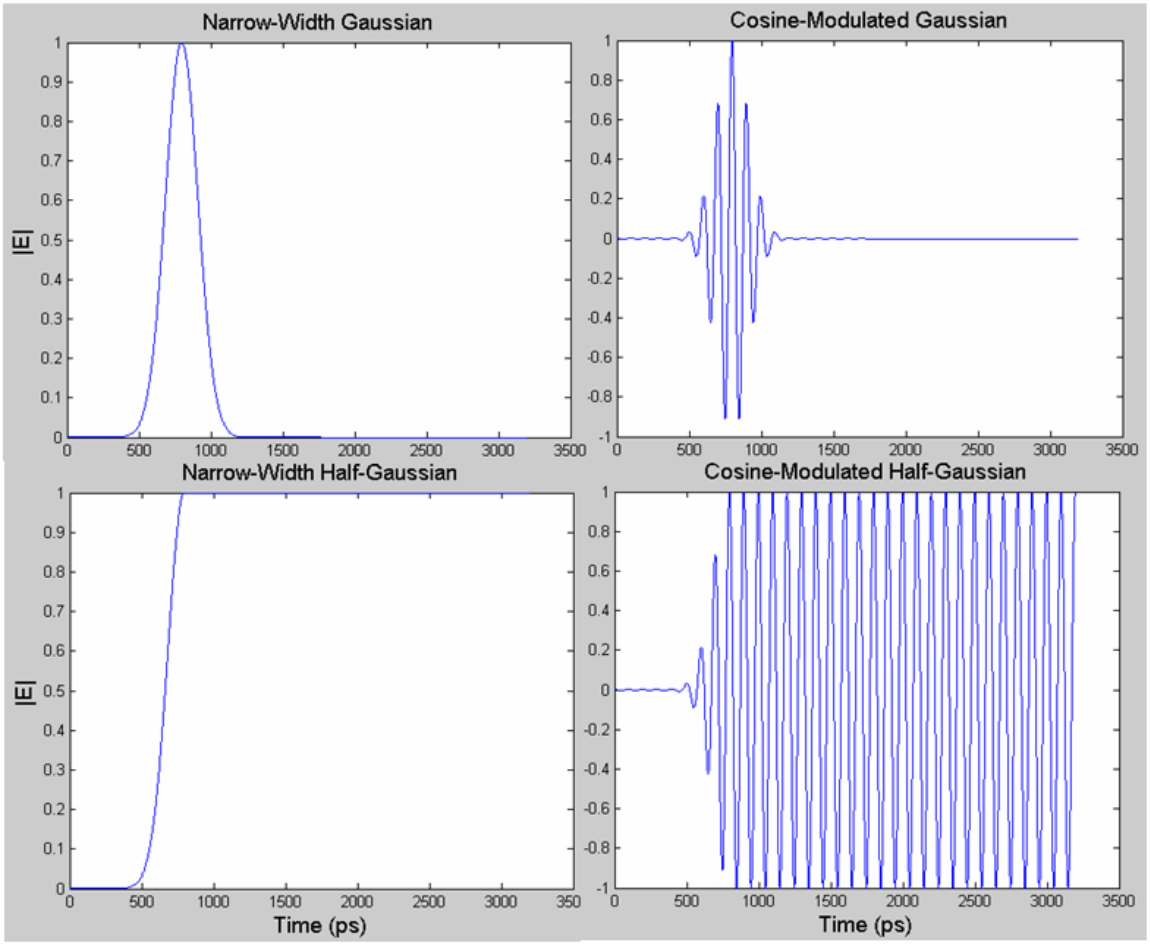
where n is time step number, e≈2.7183, p is the pulse peak time, and w is the pulse width.

The cosine-modulated Gaussian is a discretized version of the standard Gaussian multiplied by a cosine:

$$f(n) = e^{-\left(\frac{n-p}{w}\right)^2} \cos(2\pi f n \Delta t)$$

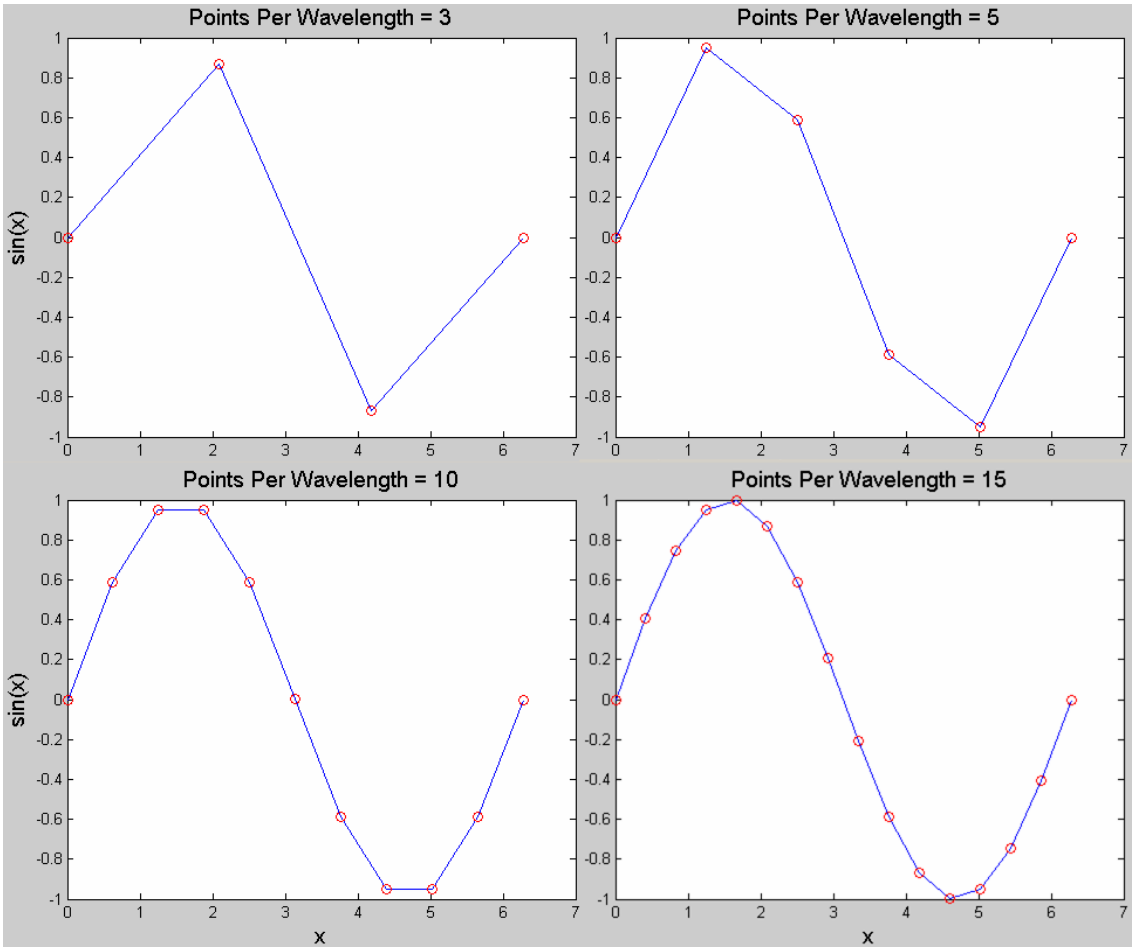
where f is the pulse frequency and Δt is the time step size.

The half-Gaussian pulses are equivalent to their regular Gaussian counterparts until the pulse peak time is reached. At that point, the narrow-width half-Gaussian's value is fixed to one and the cosine-modulated half-Gaussian's values are fixed to the values produced by the cosine term.



Excitation Input: Points Per Wavelength

In order for an FDTD to properly simulate wave propagation, there must be enough spatial grid points per wavelength to accurately model the wave. Using more points per wavelength increases both model resolution, which is desirable, and memory requirements, which is undesirable. Thus, in general, the minimum number of points per wavelength that will give sufficient accuracy is sought. 10 points per wavelength is a common choice.



If a pulse shape with cosine modulation has been selected, then the code calculates and displays points per wavelength after pulse frequency has been entered. This display can be used to determine if a satisfactory number of points per wavelength are being used. However, the code will not make such judgments on its own- the user is able to run simulations with any number of points per wavelength.

If a narrow-width pulse shape has been selected, then points per wavelength is not calculated because there is no frequency entered.

Field Component Output Input

Electric and magnetic field values can be printed to "field output files" on the computer's hard drive. Field output files are text files containing the field values of a particular field component (Ex, Ey, Ez, Hx, Hy, or Hz) along a particular slice through the computational domain and at a specific time step.

In order to avoid clogging hard disks with excessive amounts of field component output files, users can choose to write field component output files at regular intervals of time steps. The time steps in which field component files are written are called "output time steps". Simulations then produce series of field component output files, each containing the same field component slice at successive output time steps. For one simulation, all field component slices will be printed at all output time steps. Thus, the following input is needed:

- number of time steps between outputs
- number of output files per output time step: the number of field component slices to print at each output time step

For each field component slice, the following input is needed:

- field component slice variables (field, direction, location)
    - field: field to output. Uses the labels (1 - Hx; 2- Hy; 3 - Hz; 4 - Ex; 5 - Ey; 6 - Ez)
    - direction: direction of slice. Uses the labels (1 - x; 2 - y; 3 - z)
    - location: location (coordinate) of slice

Field output files will appear in the same directory as the simulation. The file name begins with two letters representing the field component, followed by an underscore, followed by a letter (x, y, or z) representing the slice direction, followed by a three-digit number representing the slice location, followed by a second underscore, followed by the letter "t" and a three-digit number representing the output time step number, followed by the extension ".dat". Thus, the Ex field component at the slice x=23 and at output time step 5 will have the file name "ex_x023_t005.dat".

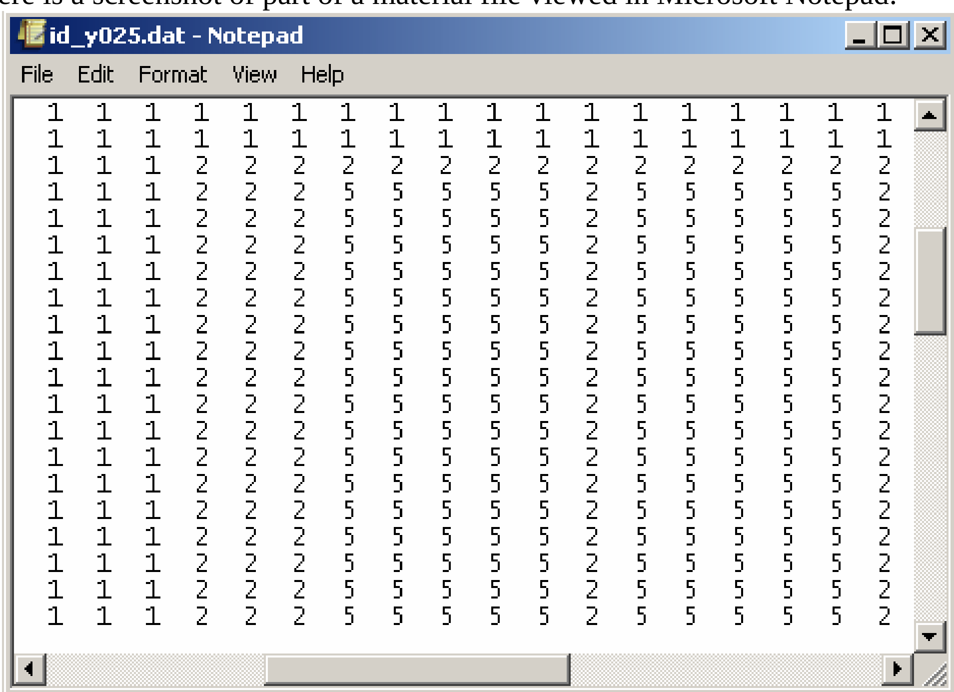
Viewing Output Files

Viewing Output Files Directly

Material or field component output files can be viewed directly by opening them up in a text editor, word processor, spreadsheet, or other program. The material or field component points will be oriented in the scheme described in the "Coordinate Systems" section, with the origin in the bottom-left corner.

If the program includes a system for assigning coordinates to points in the file, it may be possible to use these coordinates to determine what grid points values in the file are from. However, there may be several complications to this procedure. First, many programs place the origin in the top-left corner, in which case the values it gives for the y/z (vertical) direction will be wrong. Also, while spreadsheets will typically place each value in a cell, word processors generally count in the x/y (horizontal) direction by column, character by character, meaning that column number will not correspond to grid coordinate. At best, the column number may be a multiple of the grid coordinate. Finally, line wrapping will occur if there are more points in the x/y (horizontal) direction than can fit on one line in the text editor, further complicating the relationship between grid points and points in the program in use.

Here is a screenshot of part of a material file viewed in Microsoft Notepad:



1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
1	1	1	2	2	2	5	5	5	5	5	2	5	5	5	5	5	2
1	1	1	2	2	2	5	5	5	5	5	2	5	5	5	5	5	2
1	1	1	2	2	2	5	5	5	5	5	2	5	5	5	5	5	2
1	1	1	2	2	2	5	5	5	5	5	2	5	5	5	5	5	2
1	1	1	2	2	2	5	5	5	5	5	2	5	5	5	5	5	2
1	1	1	2	2	2	5	5	5	5	5	2	5	5	5	5	5	2
1	1	1	2	2	2	5	5	5	5	5	2	5	5	5	5	5	2
1	1	1	2	2	2	5	5	5	5	5	2	5	5	5	5	5	2
1	1	1	2	2	2	5	5	5	5	5	2	5	5	5	5	5	2
1	1	1	2	2	2	5	5	5	5	5	2	5	5	5	5	5	2
1	1	1	2	2	2	5	5	5	5	5	2	5	5	5	5	5	2
1	1	1	2	2	2	5	5	5	5	5	2	5	5	5	5	5	2
1	1	1	2	2	2	5	5	5	5	5	2	5	5	5	5	5	2
1	1	1	2	2	2	5	5	5	5	5	2	5	5	5	5	5	2
1	1	1	2	2	2	5	5	5	5	5	2	5	5	5	5	5	2
1	1	1	2	2	2	5	5	5	5	5	2	5	5	5	5	5	2
1	1	1	2	2	2	5	5	5	5	5	2	5	5	5	5	5	2

The astute observer may recognize this as being from a vertical slice through a monopole antenna.

Viewing Output Files In MatLab

Loading material or field component output files into a program that can effectively plot them offers significant advantages over viewing the files directly. All data in a file can be interpreted by observing the plots' colors, which is generally easier and more effective than reading the underlying numbers in a direct viewing. Some programs can even create animations to view all or parts of a field component slice series or some other sequence of output files. Images can be saved for use in papers and other reports.

A collection of MatLab files designed to read and plot the code's output files has been included with the main Fortran code to help users visualize output files. Use of these MatLab files is explained here and should be fairly easy, even for those with limited MatLab expertise. These files include:

- fc_read.m: read in field components output files
- fc_animate.m: animate field component matrices
- mt_readplot.m: read in & plot material files
- mx_getval.m: get Cartesian coordinate value of a 2D matrix

Further information about these files is available in the files themselves or through MatLab's help command. (For example, to view information about fc_read.m, first go to the file's directory using the cd (change directory), then type "help fc_read").

Sample Simulation 1 – Monopole Antenna

Here, we'll walk through the simulation of a simple monopole antenna, from the input through visualization of the results. The first thing to do is run the simulation- either double-click on the executable file (nufdtd3d.exe) or execute it from the code development software (such as Microsoft Developer Studio Fortran PowerStation).

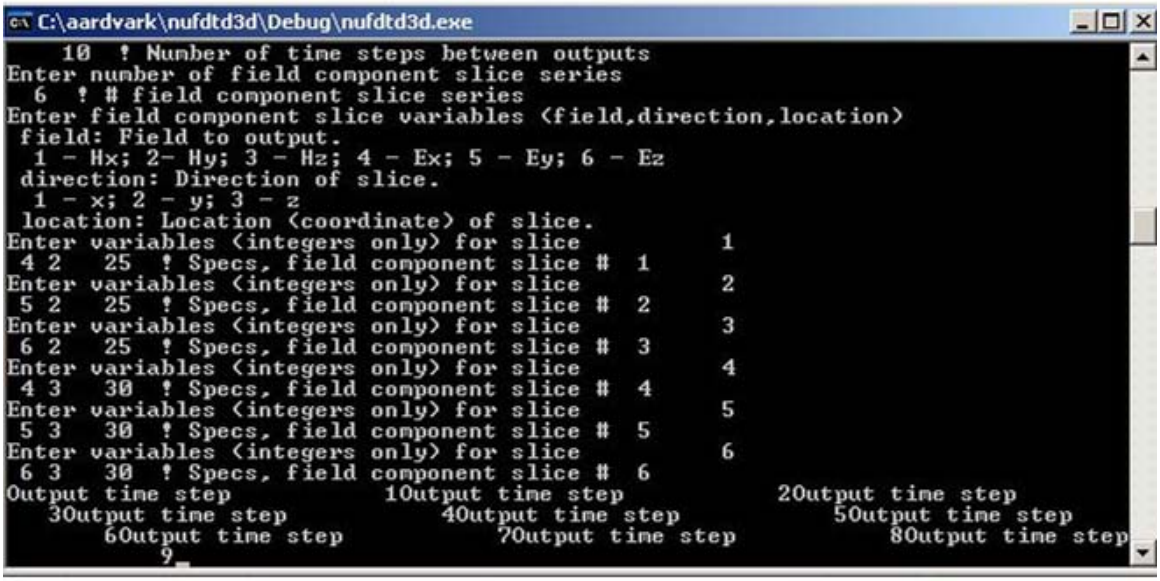
The input file contains the text:

```
  50   50   90  ! grid size
 .089   .089   .089  ! grid cell size in cm
 1000  ! number of time steps
.1000E-11  ! Time step size (sec)
0  ! Number of new materials
 1  ! Number of background layers
 1  ! Material type of layer  1
 1  ! Number of objects (built-in objects & material files)
4  ! Object type
  25   25   70  ! (i,j,k) of antenna top-center
   1    5    3  ! Core radius, dielectric thickness & shield thickness
  50   45  ! Dielectric/core length, shield length
   2  ! Number of material output files printed
 2   25  ! direction, location of material output file
 3   30  ! direction, location of material output file
3  ! Source type
2  ! Pulse shape
 50.0  ! Gaussian pulse width (time steps)
200.0  ! Gaussian peak time (time steps)
.1500E+10  ! Pulse frequency
   10  ! Number of time steps between outputs
 6  ! # field component slice series
4 2   25  ! Specs, field component slice #  1
5 2   25  ! Specs, field component slice #  2
6 2   25  ! Specs, field component slice #  3
4 3   30  ! Specs, field component slice #  4
5 3   30  ! Specs, field component slice #  5
6 3   30  ! Specs, field component slice #  6
```

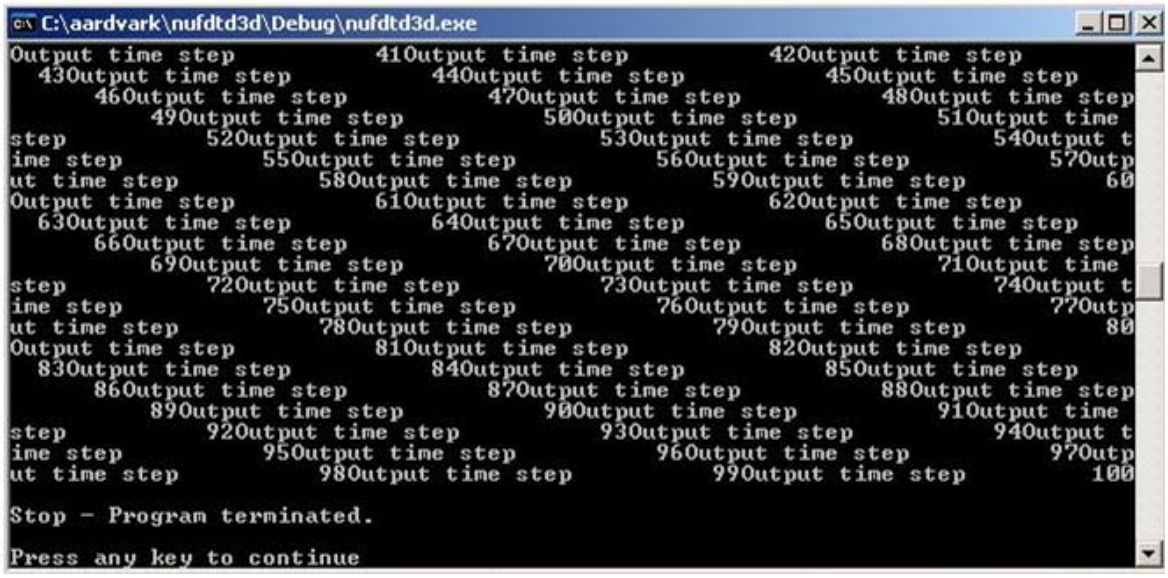
This text can be copied and pasted directly into the command window. If, while pasting it in, the command window closes, check the parameters file to make sure that the grid size allocation (npx,npy,npz) is large enough.

Sample Simulation 1 – Monopole Antenna (Continued)

This is what the command window will look like just after the input has been entered:



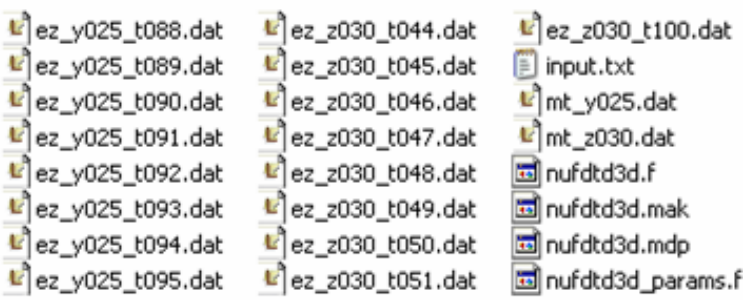
This is what the command window will look like just after the simulation finishes:



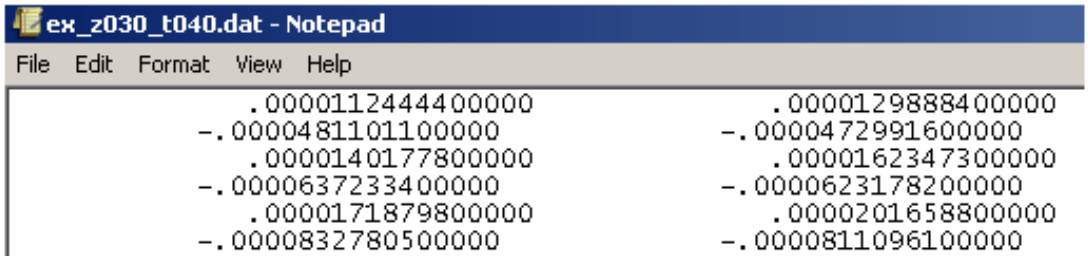
If the messages “Output time step ##” appear on separate lines instead of on the same lines as seen here, that is fine. The messages are simply a convenience for the user to monitor simulation progress and do not affect the simulation.

Sample Simulation 1 – Monopole Antenna (Continued)

This is an image of the part of the project folder after the simulation ran, showing some of the field component output files, the input file, the two material output files, and the code and project files:



This is an image of part of the file ex_z030_t040.dat, which contains the x-direction component of the electric field at the slice z=30 and at the 40th output time step (or the 400th total time step, because the simulation used 10 time steps between outputs, or 4×10^{-10} seconds after the start of the simulation, because the time step size was .1000E-11 seconds):



This is an image of a series of MatLab commands used to create a movie file of the series of field component output files containing the z-direction component of the electric field at the slice y=25:

```
>> fc = fc_read('ez','y',25,100);
>> mv = fc_animate(fc,1,100);
>> movie2avi(mv,'mv_ez_y25.avi');
Warning: The frame height has been padded to be a multiple of four as required by Intel
> In C:\MATLAB6p5\toolbox\matlab\iofun\@avifile\addframe.m at line 139
  In C:\MATLAB6p5\toolbox\matlab\iofun\movie2avi.m at line 67
Warning: The frame width has been padded to be a multiple of four as required by Intel
> In C:\MATLAB6p5\toolbox\matlab\iofun\@avifile\addframe.m at line 145
  In C:\MATLAB6p5\toolbox\matlab\iofun\movie2avi.m at line 67
>>
```

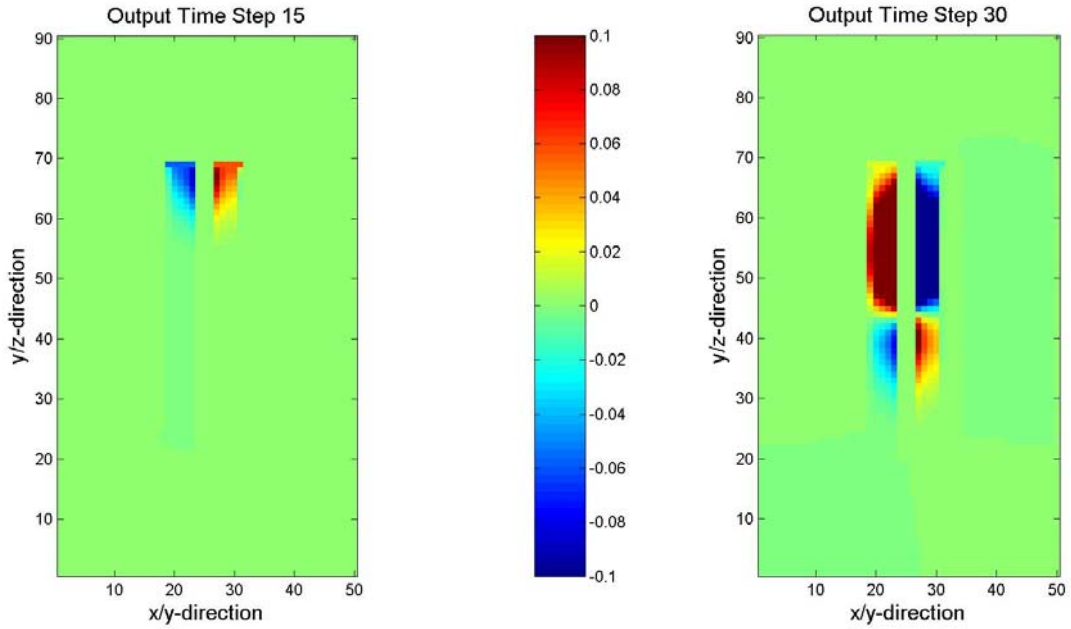
Sample Simulation 1 – Monopole Antenna (Continued)

This is an image of a series of MatLab commands used to print jpeg image files of the field component output files containing the x-direction component of the electric field at the slice y=25 at the output time steps 15 and 30:

```
>> fc = fc_read('ex','y',25,100);
>> fig = fc_jpg(fc,15,'ex_y25_t15');
>> fig = fc_jpg(fc,30,'ex_y25_t30');
```

This creates the files ‘ex_y25_t15.jpg’ & ‘ex_y25_t30.jpg’. The ‘.jpg’ is automatically added to the file name when the image file is created.

Here are the images:



(To save space on the page, the colorbar was cropped from the Output Time Step 30 image.)

Because the slice is at y=25, the horizontal axis corresponds to the x-direction and the vertical axis corresponds to the z-direction.

Sample Simulation 2 – Spiral Antenna

The spiral antenna simulation uses a material input file "psi_spiral.dat" for the antenna. It also uses a user-defined source, which requires the user to define excitation points and directions. Here we will show how to determine what the excitation should be.

The input file contains the text:

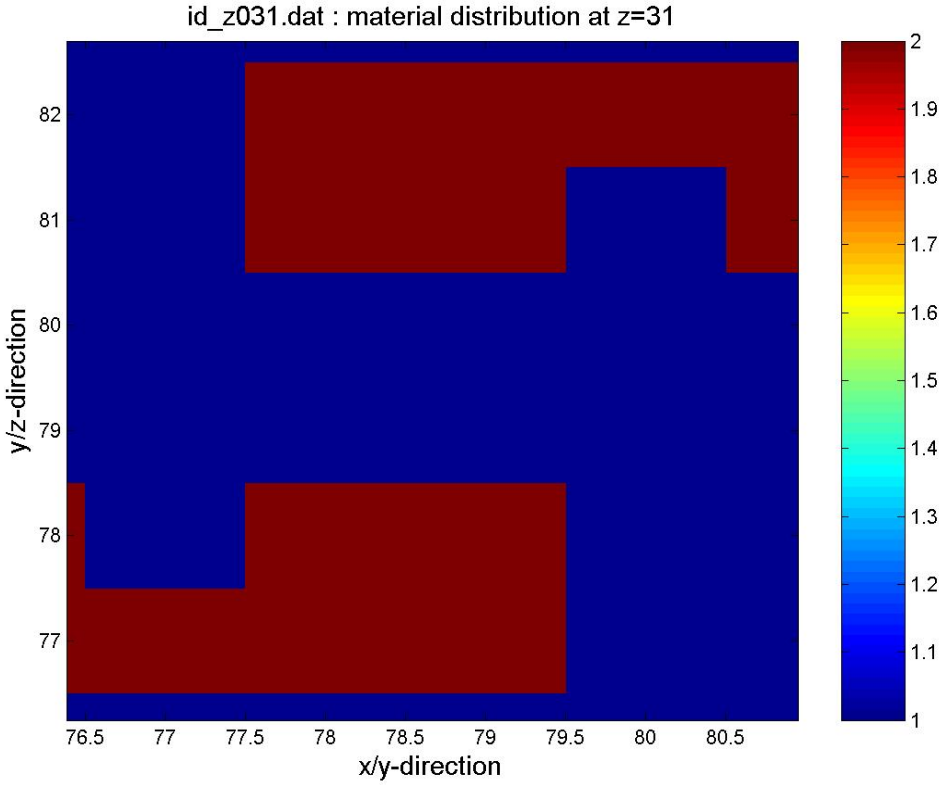
```
   165  165   61  ! grid size
  .089   .089   .089  ! grid cell size in cm
 6000  ! number of time steps
.1000E-11  ! Time step size (sec)
 0  ! Number of new materials
  1  ! Number of background layers
  1  ! Material type of layer  1
  1  ! Number of objects (built-in objects & material files)
0  ! Object type
psi_spiral_frsp.dat   ! Material input filename
  3  ! Orientation
 31  ! Height
  5    5  ! Starting point (x,y)
 157  157  ! Width (x,y)
  2  ! Number of material output files printed
 3   31  ! direction, location of material output file
 1   79  ! direction, location of material output file
2  ! Source type
 4  ! Number of excitation points
  82    80    31  ! Excitation point coords #  1
.00 1.00  .00  ! Directional exn strengths
  82    79    31  ! Excitation point coords #  2
.00 1.00  .00  ! Directional exn strengths
  83    80    31  ! Excitation point coords #  3
.00 1.00  .00  ! Directional exn strengths
  83    79    31  ! Excitation point coords #  4
.00 1.00  .00  ! Directional exn strengths
2  ! Pulse shape
 50.0  ! Gaussian pulse width (time steps)
200.0  ! Gaussian peak time (time steps)
.1500E+10  ! Pulse frequency
   20  ! Number of time steps between outputs
 4  ! # field component slice series
5 1   78  ! Specs, field component slice #  4
5 3   31  ! Specs, field component slice #  2
5 3   29  ! Specs, field component slice #  6
5 3   15  ! Specs, field component slice #  5
```

Sample Simulation 2 – Spiral Antenna (Continued)

If we didn't know the coordinates of the excitation points, we could figure this out from a plot of the antenna. To do this, first run the code and paste the input through the material output files section:

```
*** MATERIAL OUTPUT FILES
How many material output files to print?
    1  ! Number of material output files printed
Enter direction, location of mat. out. file #           1
 First number: direction: 1=x (y-z plane); 2=y; 3=z
 Second number: location: slice coordinate
  3   31  ! direction, location of material output file
```

Then, from MatLab, plot the material output file. Note that the file is of the slice that the material input file (which contains the antenna) was placed. To do this, use the program mt_readplot.m. Then, zoom in on the center, where the excitation is supposed to go:



Using the material i.d. number system, we can tell that blue is free space and red is metal. The excitation goes in the free space between the bottom and top pieces of metal. Because the image shows a z-direction slice, we know that the horizontal axis corresponds to the x-direction and the vertical axis corresponds to the y-direction. Because the slice is at z=31, it's clear that the excitation coordinates are (78,79,31), (78,80,31), (79,79,31), and (79,80,31). Also, we can see that the path between the two pieces of metal is in the y-direction, so the excitation points should all have the directional excitation strengths are all (0,1,0). (0,-1,0) would also work.

The rest of the simulation follows from the previous Sample Simulation and will not be discussed further.