Understanding the Finite-Difference Time-Domain Method

John B. Schneider

June 22, 2015

Contents

Chapter 1

Numeric Artifacts

1.1 Introduction

Virtually all solutions to problems in electromagnetics require the use of a computer. Even when an analytic or "closed form" solution is available which is nominally exact, one typically must use a computer to translate that solution into numeric values for a given set of parameters. Because of inherent limitations in the way numbers are stored in computers, some errors will invariably be present in the resulting solution. These errors will typically be small but they are an artifact about which one should be aware. Here we will discuss a few of the consequences of finite precision.

Later we will be discussing numeric solutions to electromagnetic problems which are based on the finite-difference time-domain (FDTD) method. The FDTD method makes approximations that force the solutions to be approximate, i.e., the method is inherently approximate. The results obtained from the FDTD method would be approximate even if we used computers that offered infinite numeric precision. The inherent approximations in the FDTD method will be discussed in subsequent chapters.

With numerical methods there is one note of caution which one should always keep in mind. Provided the implementation of a solution does not fail catastrophically, a computer is always willing to give you a result. You will probably find there are times when, to get your program simply to run, the debugging process is incredibly arduous. When your program does run, the natural assumption is that all the bugs have been fixed. Unfortunately that often is not the case. Getting the program to run is one thing, getting correct results is another. And, in fact, getting accurate results is yet another thing—your solution may be correct for the given implementation, but the implementation may not be one which is capable of producing sufficiently accurate results. Therefore, the more ways you have to test your implementation and your solution, the better. For example, a solution may be obtained at one level of discretization and then another solution using a finer discretization. If the two solutions are not sufficiently close, one has not yet converged to the "true" solution and a finer discretization must be used or, perhaps, there is some systemic error in the implementation. The bottom line: just because a computer gives you an answer does not mean that answer is correct.

Lecture notes by John Schneider. `numerical-issues.tex`

1.2 Finite Precision

If we sum one-eleventh eleven times we know that the result is one, i.e., $1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 = 1$. But is that true on a computer? Consider the C program shown in Program 1.1.

Program 1.1 `oneEleventh.c`: Test if $1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11 + 1/11$ equals 1.

```
1  /* Is summing 1./11. ten times == 1.0? */
2  #include <stdio.h>
3
4  int main() {
5    float a;
6
7    a = 1.0 / 11.0;
8
9    if (a + a + a + a + a + a + a + a + a + a + a == 1.0)
10     printf("Equal.\n");
11   else
12     printf("Not equal.\n");
13
14   return 0;
15 }
```

In this program the float variable `a` is set to one-eleventh. In line 9 the sum of eleven `a`'s is compared to one. If they are equal, the program prints "Equal" but prints "Not equal" otherwise. The output of this program is "Not equal." Thus, to a computer (at least one running a language typically used in the solution of electromagnetics problems), the sum of one-eleventh eleven times is not equal to one. It is worth noting that had line 9 been written `a=1/11;`, `a` would have been set to zero since integer math would be used to evaluate the division. By using `a = 1.0 / 11.0;`, the computer uses floating-point math.

The floating-point data types in C or FORTRAN can only store a finite number of digits. On most machines four bytes (32 binary digits or bits) are used for single-precision numbers and eight bytes (64 digits) are used for double precision. Returning to the sum of one-elevenths, as an extreme example, assumed that a computer can only store two decimal digits. One eleventh is equal to 0.09090909... Thus, to two decimal places one-eleventh would be approximated by 0.09. Summing this eleven times yields

$$0.09 + 0.09 + 0.09 + 0.09 + 0.09 + 0.09 + 0.09 + 0.09 + 0.09 + 0.09 + 0.09 = 0.99$$

which is clearly not equal to one. If the number is stored with more digits, the result becomes closer to one, but it never gets there. Both the decimal and binary floating-point representation of one-eleventh have an infinite number of digits. Thus, when attempting to store one-eleventh

in a computer the number has to be truncated so that the computer stores an approximation of one-eleventh. Because of this truncation summing one-eleventh eleven times does not yield one.

Since $1/10$ is equal to 0.1, it might appear this number can be stored with a finite number of digits. Although one-tenth has a finite number of digits when written in base ten (decimal representation), it has an infinite number of digits when written in base two (binary representation).

In a floating-point decimal number each digit represents the number of a particular power of ten. Letting a blank represent a digit, a decimal number can be thought of in the follow way:

$$\cdots \underset{10^3}{\_\_} \; \underset{10^2}{\_\_} \; \underset{10^1}{\_\_} \; \underset{10^0}{\_\_} \; . \; \underset{10^{-1}}{\_\_} \; \underset{10^{-2}}{\_\_} \; \underset{10^{-3}}{\_\_} \; \underset{10^{-4}}{\_\_} \; \cdots$$

Each digits tells how many of a particular power of 10 there is in a number. The decimal point serves as the dividing line between negative and non-negative exponents. Binary numbers are similar except each digit represents a power or two:

$$\cdots \underset{2^3}{\_\_} \; \underset{2^2}{\_\_} \; \underset{2^1}{\_\_} \; \underset{2^0}{\_\_} \; . \; \underset{2^{-1}}{\_\_} \; \underset{2^{-2}}{\_\_} \; \underset{2^{-3}}{\_\_} \; \underset{2^{-4}}{\_\_} \; \cdots$$

The base-ten number 0.1_{10} is simply 1×10^{-1}. To obtain the same value using binary numbers we have to take $2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + \ldots$, i.e., an infinite number of binary digits. Another way of writing this is

$$0.1_{10} = 0.0001100110011001100110011\ldots_2 .$$

As before, when this is stored in a computer, the number has to be truncated. The stored value is no longer precisely equal to one-tenth. Summing ten of these values does not yield one (although the difference is very small).

The details of how floating-point values are stored in a computer are not a primary concern. However, it is helpful to know how bits are allocated. Numbers are stored in exponential form and the standard allocation of bits is:

	total bits	sign	mantissa	exponent
single precision	32	1	23	8
double precision	64	1	52	11

Essentially the exponent gives the magnitude of the number while the mantissa gives the digits of the number—the mantissa determines the precision. The more digits available for the mantissa, the more precisely a number can be represented. Although a double-precision number has twice as many total bits as a single-precision number, it uses 52 bits for the mantissa whereas a single-precision number uses 23. Therefore double-precision numbers actually offer more than twice the precision of single-precision numbers. A mantissa of 23 binary digits corresponds to a little less than seven decimal digits. This is because 2^{23} is $8{,}388{,}608$, thus 23 binary digits can represent numbers between 0 and $8{,}388{,}607$. On the other hand, a mantissa of 52 binary digits corresponds to a value with between 15 and 16 decimal digits ($2^{52} = 4{,}503{,}599{,}627{,}370{,}496$).

For the exponent, a double-precision number has three more bits than a single-precision number. It may seem as if the double-precision exponent has been short-changed as it does not have twice as many bits as a single-precision number. However, keep in mind that the exponent represents the size of a number. Each additional bit essentially doubles the number of values that can be represented. If the exponent had nine bits, it could represent numbers which were twice as large

as single-precision numbers. The three additional bits that a double-precision number possesses allows it to represent exponents which are eight times larger than single-precision numbers. This translates into numbers which are 256 times larger (or smaller) in magnitude than single-precision numbers.

Consider the following equation

$$a + b = a.$$

From mathematics we know this equation can only be satisfied if b is zero. However, using computers this equation can be true, i.e., b makes no contribution to a, even when b is non-zero.

When numbers are added or subtracted, their mantissas are shifted until their exponents are equal. At that point the mantissas can be directly added or subtracted. However, if the difference in the exponents is greater than the length of the mantissa, then the smaller number will not have any affect when added to or subtracted from the larger number. The code fragment shown in Fragment 1.2 illustrates this phenomenon.

Fragment 1.2 Code fragment to test if a non-zero b can satisfy the equation $a + b = a$.

```
1   float a = 1.0, b = 0.5, c;
2
3   c = a + b;
4
5   while(c != a) {
6     b = b / 2.0;
7     c = a + b;
8   }
9
10  printf("%12g %12g %12g\n",a,b,c);
```

Here a is initialized to one while b is set to one-half. The variable c holds the sum of a and b. The while-loop starting on line 5 will continue as long as c is not equal to a. In the body of the loop, b is divided by 2 and c is again set equal to $a + b$. If the computer had infinite precision, this would be an infinite loop. The value of b would become vanishingly small, but it would never be zero and hence $a + b$ would never equal a. However, the loop does terminate and the output of the `printf()` statement in line 10 is:

```
           1   5.96046e-08   1
```

This shows that both a and c are unity while b has a value of 5.96046×10^{-8}. Note that this value of b corresponds to 1×2^{-24}. When b has this value, the exponents of a and b differ by more than 23 (a is 1×2^0).

One more example serves to illustrate the less-than-obvious ways in which finite precision can corrupt a calculation. Assume the variable a is set equal to 2. Taking the square root of a and then squaring a should yield a result which is close to 2 (ideally it would be 2, but since $\sqrt{2}$ has an infinite number of digits, some accuracy will be lost). However, what happens if the square root is

taken 23 times and then the number is squared 23 times? We would hope to get a result close to two, but that is not the case. The program shown in Program 1.3 allows us to test this scenario.

Program 1.3 `rootTest.c`: Take the square root of a number repeatedly and then squaring the number an equal number of times.

```
/* Square-root test. */
#include <math.h>  // needed for sqrt()
#include <stdio.h>

#define COUNT 23

int main() {
  float a = 2.0;
  int i;

  for (i = 0; i < COUNT; i++)
    a = sqrt(a);  // square root of a

  for (i = 0; i < COUNT; i++)
    a = a * a;       // a squared

  printf("%12g\n",a);

  return 0;
}
```

The program output is one, i.e., the result is $a = 1.0$. Each time the square root is taken, the value gets closer and closer to unity. Eventually, because of truncation error, the computer thinks the number is unity. At that point no amount of squaring the number will change it.

1.3 Symbolic Manipulation

When using languages which are typically used in numerical analysis (such as C, C++, FORTRAN, or even Matlab), truncation error is unavoidable. The ratio of the circumference of a circle to its diameter is the number $\pi = 3.141592\ldots$ This is an irrational number with an infinite number of digits. Thus one cannot store the exact numeric value of π in a computer. Instead, one must use an approximation consisting of a finite number of digits. However, there are software packages, such a Mathematica, that allow one to manipulate symbols. Within Mathematica, if a person writes `Pi`, Mathematica "knows" symbolically what that means. For example, the cosine of `10000000001*Pi` is identically negative one. Similarly, one could write `Sqrt[2]`. Mathematica knows that the square of this is identically 2. Unfortunately, though, such symbolic manipulations are incredibly expensive in terms of computational resources. Many cutting-edge

problems in electromagnetics can involve hundreds of thousand or even millions of unknowns. To deal with these large amounts of data it is imperative to be as efficient—both in terms of memory and computation time—as possible. Mathematica is wonderful for many things, but it is not the right tool for solving large numeric problems.

In Matlab one can write `pi` as a shorthand representation of π. However, this representation of π is different from that used in Mathematica. In Matlab, `pi` is essentially the same as the numeric representation—it is just more convenient to write `pi` than all the numeric digits. In C, provided you have included the header file `math.h`, you can use `M_PI` as a shorthand for π. Looking in `math.h` reveals the following statement:

```
# define M_PI           3.14159265358979323846  /* pi */
```

This is similar to what is happening in Matlab. Matlab only knows what the numeric value of `pi` is and that numeric value is a truncated version of the true value. Thus, taking the cosine of `10000000001*pi` yields -0.99999999999954 instead of the exact value of -1 (but, of course, the difference is trivial in this case).

Chapter 2

Brief Review of Electromagnetics

2.1 Introduction

The specific equations on which the finite-difference time-domain (FDTD) method is based will be considered in some detail later. The goal here is to remind you of the physical significance of the equations to which you have been exposed in previous courses on electromagnetics.

In some sense there are just a few simple premises which underlie all electromagnetics. One could argue that electromagnetics is simply based on the following:

1. Charge exerts force on other charge.
2. Charge in motion exerts a force on other charge in motion.
3. All material is made up of charged particles.

Of course translating these premises into a corresponding mathematical framework is not trivial. However one should not lose sight of the fact that the math is trying to describe principles that are conceptually rather simple.

2.2 Coulomb's Law and Electric Field

Coulomb studied the electric force on charged particles. As depicted in Fig. 2.1, given two discrete particles carrying charge Q_1 and Q_2, the force experienced by Q_2 due to Q_1 is along the line joining Q_1 and Q_2. The force is proportional to the charges and inversely proportional to the square of the distance between the charges. A proportionality constant is needed to obtain Coulomb's law which gives the equation of the force on Q_2 due to Q_1:

$$\mathbf{F}_{12} = \hat{\mathbf{a}}_{12} \frac{1}{4\pi\epsilon_0} \frac{Q_1 Q_2}{R_{12}^2} \tag{2.1}$$

where $\hat{\mathbf{a}}_{12}$ is a unit vector pointing from Q_1 to Q_2, R_{12} is the distance between the charges, and $1/4\pi\epsilon_0$ is the proportionality constant. The constant ϵ_0 is known as the permittivity of free space

Lecture notes by John Schneider. `em-review.tex`

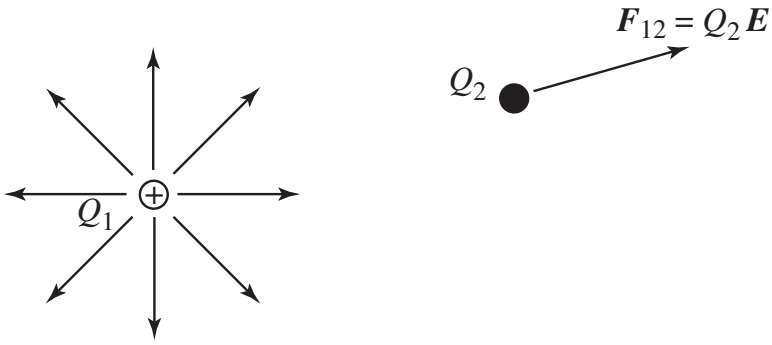


Figure 2.1: The force experienced by charge Q_2 due to charge Q_1 is along the line which pass through both charges. The direction of the force is dictate by the signs of the charges. Electric field is assumed to point radially away from positive charges as is indicated by the lines pointing away from Q_1 (which is assumed here to be positive).

and equals approximately 8.854×10^{-12} F/m. Charge is expressed in units of Coulombs (C) and can be either negative or positive. When the two charges have like signs, the force will be repulsive: $\mathbf{F}_{12}$ will be parallel to $\hat{\mathbf{a}}_{12}$. When the charges are of opposite sign, the force will be attractive so that $\mathbf{F}_{12}$ will be anti-parallel to $\hat{\mathbf{a}}_{12}$.

There is a shortcoming with (2.1) in that it implies action at a distance. It appears from this equation that the force $\mathbf{F}_{12}$ is established instantly. From this equation one could assume that a change in the distance R_{12} results in an instantaneous change in the force $\mathbf{F}_{12}$, but this is not the case. A finite amount of time is required to communicate the change in location of one charge to the other charge (similarly, it takes a finite amount of time to communicate a change in the quantity of one charge to the other charge). To overcome this shortcoming it is convenient to employ the concept of fields. Instead of Q_1 producing a force directly on Q_2, Q_1 is said to produce a field. This field then produces a force on Q_2. The field produced by Q_1 is independent of Q_2—it exists whether or not Q_2 is there to experience it.

In the static case, the field approach does not appear to have any advantage over the direct use of Coulomb's law. This is because for static charges Coulomb's law is correct. Fields must be time-varying for the distinction to arise. Nevertheless, to be consistent with the time-varying case, fields are used in the static case as well. The electric field produced by the point charge Q_1 is

$$\mathbf{E}_1 = \hat{\mathbf{a}}_r \frac{Q_1}{4\pi\epsilon_0 r^2} \tag{2.2}$$

where $\hat{\mathbf{a}}_r$ is a unit vector which points radially away from the charge and r is the distance from the charge. The electric field has units of volts per meter (V/m).

To find the force on Q_2, one merely takes the charge times the electric field: $\mathbf{F}_{12} = Q_2\mathbf{E}_1$. In general, the force on any charge Q is the product of the charge and the electric field at which the charge is present, i.e., $\mathbf{F} = Q\mathbf{E}$.

2.3 Electric Flux Density

All material is made up of charged particles. The material may be neutral overall because it has as many positive charges as negative charges. Nevertheless, there are various ways in which the positive and negative charges may shift slightly within the material, perhaps under the influence of an electric field. The resulting charge separation will have an effect on the overall electric field. Because of this it is often convenient to introduce a new field known as the electric flux density, $\mathbf{D}$, which has units of Coulombs per square meter (C/m^2).* Essentially the $\mathbf{D}$ field ignores the local effects of charge which is bound in a material.

In free space, the electric field and the electric flux density are related by

$$\mathbf{D} = \epsilon_0 \mathbf{E}. \tag{2.3}$$

Gauss's law states that integrating $\mathbf{D}$ over a closed surface yields the enclosed free charge

$$\oint_S \mathbf{D} \cdot \mathbf{ds} = Q_{\text{enc}} \tag{2.4}$$

where S is the closed surface, $\mathbf{ds}$ is an incremental surface element whose normal is directed radially outward, and Q_{enc} is the enclosed charge. As an example, consider the electric field given in (2.2). Taking S to be a spherical surface with the charge at the center, it is simple to perform the integral in (2.4):

$$\oint_S \mathbf{D} \cdot \mathbf{ds} = \int_{\theta=0}^{\pi} \int_{\phi=0}^{2\pi} \epsilon_0 \frac{Q_1}{4\pi\epsilon_0 r^2} \hat{\mathbf{a}}_r \cdot \hat{\mathbf{a}}_r r^2 \sin\theta \, d\phi \, d\theta = Q_1. \tag{2.5}$$

The result is actually independent of the surface chosen (provided it encloses the charge), but the integral is especially easy to perform for a spherical surface.

We want the integral in (2.4) always to equal the enclosed charge as it does in free space. However, things are more complicated when material is present. Consider, as shown in Fig. 2.2, two large parallel plates which carry uniformly distributed charge of equal magnitude but opposite sign. The dashed line represents an integration surface S which is assumed to be sufficiently far from the edges of the plate so that the field is uniform over the top of S. This field is identified as $\mathbf{E}_0$. The fields are zero outside of the plates and are tangential to the sides of S within the plates. Therefore the only contribution to the integral would be from the top of S. The result of the integral $\oint_S \epsilon_0 \mathbf{E} \cdot \mathbf{ds}$ is the negative charge enclosed by the surface (i.e., the negative charge on the bottom plate which falls within S).

Now consider the same plates, carrying the same charge, but with a material present between the plates. Assume this material is "polarizable" such that the positive and negative charges can shift slightly. The charges are not completely free to move—they are bound charges. The positive charges will be repelled by the top plate and attracted to the bottom plate. Conversely, the negative charges will be repelled by the bottom plate and attracted to the top plate. This scenario is depicted in Fig. 2.3.

*Note that not everybody advocates using the $\mathbf{D}$ field. See for example Volume II of *The Feynman Lectures on Physics*, R. P. Feynman, R. B. Leighton, and M. Sands, Addison-Wesley, 1964. Feynman only uses $\mathbf{E}$ and never resorts to $\mathbf{D}$.

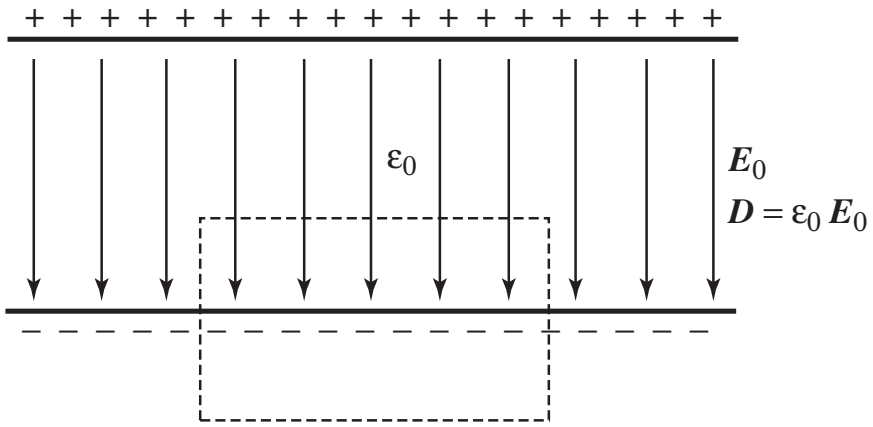


Figure 2.2: Charged parallel plates in free space. The dashed line represents the integration surface S.

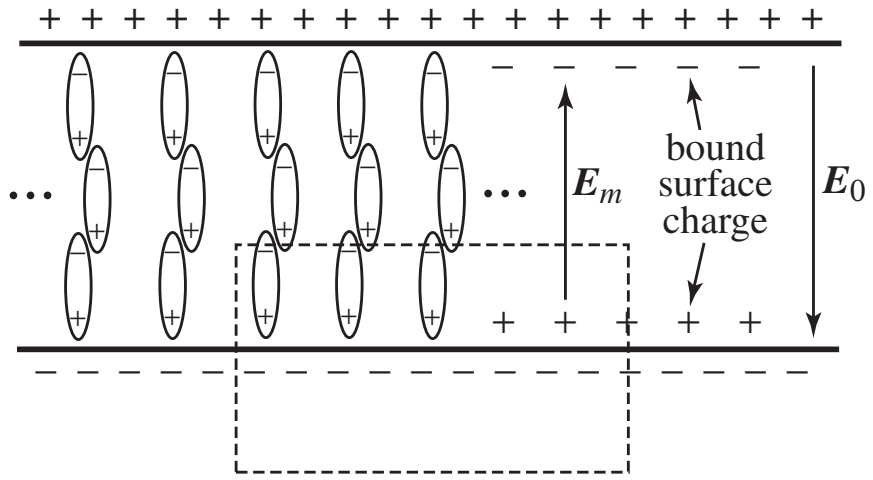


Figure 2.3: Charged parallel plates with a polarizable material present between the plates. The elongated objects represent molecules whose charge orientation serves to produce a net bound negative charge layer at the top plate and a bound positive charge layer at the bottom plate. In the interior, the positive and negative bound charges cancel each other. It is only at the surface of the material where one must account for the bound charge. Thus, the molecules are not drawn throughout the figure. Instead, as shown toward the right side of the figure, merely the bound charge layer is shown. The free charge on the plates creates the electric field $\mathbf{E}_0$. The bound charge creates the electric field $\mathbf{E}_m$ which opposes $\mathbf{E}_0$ and hence diminishes the total electric field. The dashed line again represents the integration surface S.

With the material present the electric field due to the charge on the plates is still $\mathbf{E}_0$, i.e., the same field as existed in Fig. (2.2). However, there is another field present due to the displacement of the bound charge in the polarizable material between the plates. The polarized material effectively acts to establish a layer of positive charge adjacent to the bottom plate and a layer of negative charge adjacent to the top plate. The field due to these layers of charge is also uniform but it is in the opposite direction of the field caused by the "free charge" on the plates. The field due to bound charge is labeled $\mathbf{E}_m$ in Fig. (2.3). The total field is the sum of the fields due to the bound and free charges, i.e., $\mathbf{E} = \mathbf{E}_0 + \mathbf{E}_m$. Because $\mathbf{E}_0$ and $\mathbf{E}_m$ are anti-parallel, the magnitude of the total electric field $\mathbf{E}$ will be less than $\mathbf{E}_0$.

Since the material is neutral, we would like the integral of the electric flux over the surface S to yield just the enclosed charge on the bottom plate—not the bound charge due to the material. In some sense this implies that the integration surface cannot separate the positive and negative bound charge of any single molecule. Each molecule is either entirely inside or outside the integration surface. Since each molecule is neutral, the only contribution to the integral will be from the free charge on the plate.

With the material present, the integral of $\oint_S \epsilon_0 \mathbf{E} \cdot \mathbf{ds}$ yields too little charge. This is because, as stated above, the total electric field $\mathbf{E}$ is less than it would be if only free space were present. To correct for the reduced field and to obtain the desired result, the electric flux density is redefined so that it accounts for the presence of the material. The more general expression for the electric flux density is

$$\mathbf{D} = \epsilon_r \epsilon_0 \mathbf{E} = \epsilon \mathbf{E} \tag{2.6}$$

where ϵ_r is the relative permittivity and ϵ is called simply the permittivity. By accounting for the permittivity of a material, Gauss's law is always satisfied.

In (2.6), $\mathbf{D}$ and $\mathbf{E}$ are related by a scalar constant. This implies that the $\mathbf{D}$ and $\mathbf{E}$ fields are related by a simple proportionality constant for all frequencies, all orientations, and all field strengths. Unfortunately the real world is not so simple. Clearly if the electric field is strong enough, it would be possible to tear apart the bound positive and negative charges. Since charges have some mass, they do not react the same way at all frequencies. Additionally, many materials may have some structure, such as crystals, where the response in one direction is not the same in other directions. Nevertheless, Gauss's law is the law and thus always holds. When things get more complicated one must abandon a simple scalar for the permittivity and use an appropriate form to ensure Gauss's law is satisfied. So, for example, it may be necessary to use a tensor for permittivity that is directionally dependent. However, with the exception of frequency-dependent behavior (i.e., dispersive materials), we will not be pursuing those complications. A scalar permittivity will suffice.

2.4 Static Electric Fields

Ignoring possible nonlinear behavior of material, superposition holds for electromagnetic fields. Therefore we can think of any distribution of charges as a collection of point charges. We can get the total field by summing the contributions from all the charges (and this summing will have to be in the form of an integration if the charge is continuously distributed).

Note from (2.2) that the field associated with a point charge merely points radially away from the charge. There is no "swirling" of the field. If we have more than a single charge, the total

field may bend, but it will not swirl. Imagine a tiny wheel with positive charge distributed around its circumference. The wheel hub of the wheel is held at a fixed location but the wheel is free to spin about its hub. For static electric fields, no matter where we put this wheel, there would be no net force on the wheel to cause it to spin. There may be a net force pushing the entire wheel in a particular direction (a translational force), but the forces which are pushing the wheel to spin in the clockwise direction are balanced by the forces pushing the wheel to spin in the counterclockwise direction.

Another property of electrostatic fields is that the electric flux density only begins or terminates on free charge. If there is no charge present, the lines of flux continue.

The lack of swirl in the electric field and the source of electric flux density are fairly simple concepts. However, to be able to analyze the fields properly, one needs a mathematical statement of these concepts. The appropriate statements are

$$\nabla \times \mathbf{E} = 0 \tag{2.7}$$

and

$$\nabla \cdot \mathbf{D} = \rho_v \tag{2.8}$$

where ∇ is the del or nabla operator and ρ_v is the electric charge density (with units of C/m^3). Equation (2.7) is the curl of the electric field and (2.8) is the divergence of the electric flux density. These two equations are discussed further in the following section.

2.5 Gradient, Divergence, and Curl

The del operator is independent of the coordinate system used—naturally the behavior of the fields should not depend on the coordinate system used to describe the field. Nevertheless, the del operator can be expressed in different coordinates systems. In Cartesian coordinates del is

$$\nabla \equiv \hat{\mathbf{a}}_x \frac{\partial}{\partial x} + \hat{\mathbf{a}}_y \frac{\partial}{\partial y} + \hat{\mathbf{a}}_z \frac{\partial}{\partial z} \tag{2.9}$$

where the symbol $\equiv$ means "defined as."

Del acting on a scalar field produces the gradient of the field. Assuming f is a some scalar field, ∇f produces the vector field given by

$$\nabla f = \hat{\mathbf{a}}_x \frac{\partial f}{\partial x} + \hat{\mathbf{a}}_y \frac{\partial f}{\partial y} + \hat{\mathbf{a}}_z \frac{\partial f}{\partial z}. \tag{2.10}$$

The gradient of f points in the direction of greatest change and is proportional to the rate of change. Assume we wish to find the amount of change in f for a small movement dx in the x direction. This can be obtained via $\nabla f \cdot \hat{\mathbf{a}}_x dx$, to wit

$$\nabla f \cdot \hat{\mathbf{a}}_x dx = \frac{\partial f}{\partial x} dx = \text{(rate of change in } x \text{ direction)} \times \text{(movement in } x \text{ direction)}. \tag{2.11}$$

This can be generalized for movement in an arbitrary direction. Letting an incremental small length be given by

$$\mathbf{d}\ell = \hat{\mathbf{a}}_x dx + \hat{\mathbf{a}}_y dy + \hat{\mathbf{a}}_z dz, \tag{2.12}$$

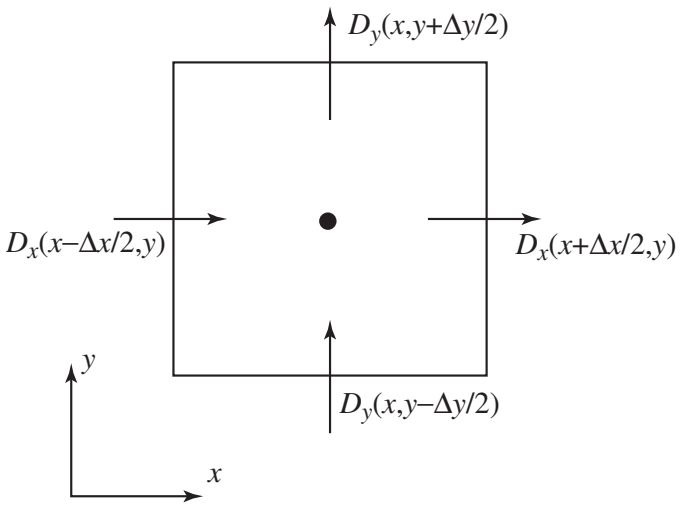


Figure 2.4: Discrete approximation to the divergence taken in the xy-plane.

the change in the field realized by moving an amount $\mathbf{d}\ell$ is

$$\nabla f \cdot \mathbf{d}\ell = \frac{\partial f}{\partial x}dx + \frac{\partial f}{\partial y}dy + \frac{\partial f}{\partial z}dz. \tag{2.13}$$

Returning to (2.8), when the del operator is dotted with a vector field, one obtains the divergence of that field. Divergence can be thought of as a measure of "source" or "sink" strength of the field at a given point. The divergence of a vector field is a scalar field given by

$$\nabla \cdot \mathbf{D} = \frac{\partial D_x}{\partial x} + \frac{\partial D_y}{\partial y} + \frac{\partial D_z}{\partial z}. \tag{2.14}$$

Let us consider a finite-difference approximation of this divergence in the xy-plane as shown in Fig. 2.4. Here the divergence is measured over a small box where the field is assumed to be constant over each edge of the box. The derivatives can be approximated by central differences:

$$\frac{\partial D_x}{\partial x} + \frac{\partial D_y}{\partial y} \approx \frac{D_x\left(x+\frac{\Delta_x}{2},y\right) - D_x\left(x-\frac{\Delta_x}{2},y\right)}{\Delta_x} + \frac{D_y\left(x,y+\frac{\Delta_y}{2}\right) - D_y\left(x,y-\frac{\Delta_y}{2}\right)}{\Delta_y} \tag{2.15}$$

where this is exact as Δ_x and Δ_y go to zero. Letting $\Delta_x = \Delta_y = \delta$, (2.15) can be written

$$\frac{\partial D_x}{\partial x} + \frac{\partial D_y}{\partial y} \approx \frac{1}{\delta}\left(D_x\left(x+\frac{\delta}{2},y\right) - D_x\left(x-\frac{\delta}{2},y\right) + D_y\left(x,y+\frac{\delta}{2}\right) - D_y\left(x,y-\frac{\delta}{2}\right)\right). \tag{2.16}$$

Inspection of (2.16) reveals that the divergence is essentially a sum of the field over the faces with the appropriate sign changes. Positive signs are used if the field is assumed to point out of the box and negative signs are used when the field is assumed to point into the box. If the sum of these values is positive, that implies there is more flux out of the box than into it. Conversely, if the sum is negative, that means more flux is flowing into the box than out. If the sum is zero, there must

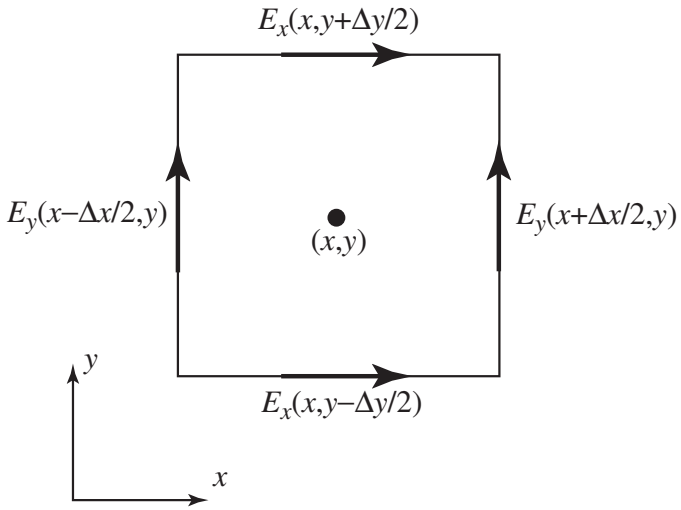


Figure 2.5: Discrete approximation to the curl taken in the xy-plane.

be as much flux flowing into the box as out of it (that does not imply necessarily that, for instance, $D_x\left(x+\delta/2,y\right)$ is equal to $D_x\left(x-\delta/2,y\right)$, but rather that the sum of all four fluxes must be zero).

Equation (2.8) tells us that the electric flux density has zero divergence except where there is charge present (as specified by the charge-density term ρ_v). If the charge density is zero, the total flux entering some small enclosure must also leave it. If the charge density is positive at some point, more flux will leave a small enclosure surrounding that point than will enter it. On the other hand, if the charge density is negative, more flux will enter the enclosure surrounding that point than will leave it.

Finally, let us consider (2.7) which is the curl of the electric field. In Cartesian coordinates it is possible to treat this operation as simply the cross product between the vector operator ∇ and the vector field $\mathbf{E}$:

$$\nabla\times\mathbf{E}=\begin{vmatrix}\hat{\mathbf{a}}_x & \hat{\mathbf{a}}_y & \hat{\mathbf{a}}_z\\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z}\\ E_x & E_y & E_z\end{vmatrix}=\hat{\mathbf{a}}_x\left(\frac{\partial E_z}{\partial y}-\frac{\partial E_y}{\partial z}\right)+\hat{\mathbf{a}}_y\left(\frac{\partial E_x}{\partial z}-\frac{\partial E_z}{\partial x}\right)+\hat{\mathbf{a}}_z\left(\frac{\partial E_y}{\partial x}-\frac{\partial E_x}{\partial y}\right). \tag{2.17}$$

Let us consider the behavior of only the z component of this operator which is dictated by the field in the xy-plane as shown in Fig. 2.5. The z-component of $\nabla\times\mathbf{E}$ can be written as

$$\frac{\partial E_y}{\partial x}-\frac{\partial E_x}{\partial y}\approx\frac{E_y\left(x+\frac{\Delta_x}{2},y\right)-E_y\left(x-\frac{\Delta_x}{2},y\right)}{\Delta_x}-\frac{E_x\left(x,y+\frac{\Delta_y}{2}\right)-E_x\left(x,y-\frac{\Delta_y}{2}\right)}{\Delta_y}. \tag{2.18}$$

The finite-difference approximations of the derivatives are again based on the fields on the edges of a box surrounding the point of interest. However, in this case the relevant fields are tangential to the edges rather than normal to them. Again letting $\Delta_x=\Delta_y=\delta$, (2.18) can be written

$$\frac{\partial E_y}{\partial x}-\frac{\partial E_x}{\partial y}\approx\frac{1}{\delta}\left(E_y\left(x+\frac{\delta}{2},y\right)-E_y\left(x-\frac{\delta}{2},y\right)-E_x\left(x,y+\frac{\delta}{2}\right)+E_x\left(x,y-\frac{\delta}{2}\right)\right). \tag{2.19}$$

In the sum on the right side the sign is positive if the vector component points in the counter-clockwise direction (relative to rotations about the center of the box) and is negative if the vector points in the clockwise direction. Thus, if the sum of these vector components is positive, that implies that the net effect of these electric field vectors is to tend to push a positive charge in the counterclockwise direction. If the sum were negative, the vectors would tend to push a positive charge in the clockwise direction. If the sum is zero, there is no tendency to push a positive charge around the center of the square (which is not to say there would not be a translation force on the charge—indeed, if the electric field is non-zero, there has to be some force on the charge).

2.6 Laplacian

In addition to the gradient, divergence, and curl, there is one more vector operator to consider. There is a vector identity that the curl of the gradient of any function is identically zero

$$\nabla \times \nabla f = 0. \tag{2.20}$$

This is simple to prove by merely performing the operations in Cartesian coordinates. One obtains several second-order partial derivatives which cancel if the order of differentiation is switched. Recall that for a static distribution of charges, $\nabla \times \mathbf{E} = 0$. Since the curl of the electric field is zero, it should be possible to represent the electric field as the gradient of some scalar function

$$\mathbf{E} = -\nabla V. \tag{2.21}$$

The scalar function V is the electric potential and the negative sign is used to make the electric field point from higher potential to lower potential (by historic convention the electric field points away from positive charge and toward negative charge). By expressing the electric field this way, the curl of the electric field is guaranteed to be zero.

Another way to express the relationship between the electric field and the potential is via integration. Consider movement from an arbitrary point a to an arbitrary point b. The change in potential between these two points can be expressed as

$$V_b - V_a = \int_a^b \nabla V \cdot \mathbf{d}\ell. \tag{2.22}$$

The integrand represent the change in the potential for a movement $\mathbf{d}\ell$ and the integral merely sums the changes over the path from a to b. However, the change in potential must also be commensurate with the movement in the direction of, or against, the electric field. If we move against the electric field, potential should go up. If we move along the electric field, the potential should go down. In other words, the incremental change in potential for a movement $\mathbf{d}\ell$ should be $dV = -\mathbf{E} \cdot \mathbf{d}\ell$ (if the movement $\mathbf{d}\ell$ is orthogonal to the electric field, there should be no change in the potential). Summing change in potential over the entire path yields

$$V_b - V_a = -\int_a^b \mathbf{E} \cdot \mathbf{d}\ell. \tag{2.23}$$

The integrals in (2.22) and (2.23) can be equated. Since the equality holds for any two arbitrary points, the integrands must be equal and we are again left with $\mathbf{E} = -\nabla V$.

The electric flux density can be related to the electric field via $\mathbf{D} = \epsilon\mathbf{E}$ and the behavior of the flux density $\mathbf{D}$ is dictated by $\nabla \cdot \mathbf{D} = \rho_v$. Combining these with (2.21) yields

$$\mathbf{E} = \frac{1}{\epsilon}\mathbf{D} = -\nabla V. \tag{2.24}$$

Taking the divergence of both sides yields

$$\frac{1}{\epsilon}\nabla \cdot \mathbf{D} = \frac{1}{\epsilon}\rho_v = -\nabla \cdot \nabla V. \tag{2.25}$$

Rearranging this yields Poisson's equation given by

$$\nabla^2 V = -\frac{\rho_v}{\epsilon} \tag{2.26}$$

where ∇^2 is the Laplacian operator

$$\nabla^2 \equiv \nabla \cdot \nabla = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}. \tag{2.27}$$

Note that the Laplacian is a scalar operator. It can act on a scalar field (such as the potential V as shown above) or it can act on a vector field as we will see later. When it acts on a vector field, the Laplacian acts on each component of the field.

In the case of zero charge density, (2.26) reduces to Laplace's equation

$$\nabla^2 V = 0. \tag{2.28}$$

We have a physical intuition about what gradient, divergence, and curl are telling us, but what about the Laplacian? To answer this, consider a function of a single variable.

Given the function $V(x)$, we can ask if the function at some point is greater than, equal to, or less than the average of its neighboring values. The answer can be expressed in terms of the value of the function at the point of interest and the average of samples to either side of that central point:

$$\frac{V(x+\delta)+V(x-\delta)}{2} - V(x) = \begin{cases} \text{positive} & \text{if center point less than average of neighbors} \\ \text{zero} & \text{if center point equals average of neighbors} \\ \text{negative} & \text{if center point greater than average of neighbors} \end{cases} \tag{2.29}$$

Here the left-most term represents the average of the neighboring values and δ is some displacement from the central point. Equation (2.29) can be normalized by $\delta^2/2$ without changing the basic tenants of this equation. Performing that normalization and rearranging yields

$$\begin{aligned} \frac{1}{\delta^2/2}\left\{\frac{V(x+\delta)+V(x-\delta)}{2} - V(x)\right\} &= \frac{1}{\delta^2}\left\{(V(x+\delta)-V(x)) - (V(x)-V(x-\delta))\right\} \\ &= \frac{\frac{V(x+\delta)-V(x)}{\delta} - \frac{V(x)-V(x-\delta)}{\delta}}{\delta} \\ &\approx \frac{\frac{\partial V(x+\delta/2)}{\partial x} - \frac{\partial V(x-\delta/2)}{\partial x}}{\delta} \\ &\approx \frac{\partial^2 V(x)}{\partial x^2}. \end{aligned} \tag{2.30}$$

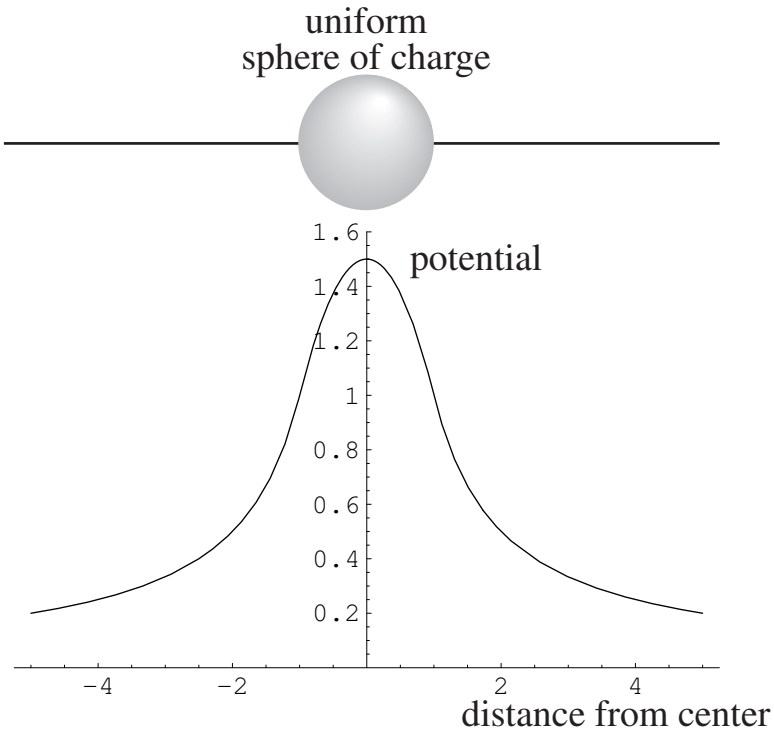


Figure 2.6: Potential along a path which passes through a uniform sphere of positive charge (arbitrary units).

Thus the second partial derivative can be thought of as a way of measuring the field at a point relative to its neighboring points. You should already have in mind that if the second derivative is negative, a function is tending to curve downward. Second derivatives are usually discussed in the context of curvature. However, you should also think in terms of the field at a point and its neighbors. At points where the second derivative is negative those points are higher than the average of their neighboring points (at least if the neighbors are taken to be an infinitesimally small distance away).

In lieu of these arguments, Poisson's equation (2.26) should have physical significance. Where the charge density is zero, the potential cannot have a local minima or maxima. The potential is always equal to the average of the neighboring points. If one neighbor is higher, the other must be lower (and this concept easily generalizes to higher dimensions). Conversely, if the charge density is positive over some region, the potential should increase as one moves deeper into that region but the rate of increase must be such that at any point the average of the neighbors is less than the center point. This behavior is illustrated in Fig. 2.6 which depicts the potential along a path through the center of a uniform sphere of charge.

2.7 Gauss's and Stokes' Theorems

Equation (2.4) presented Gauss's law which stated the flux of $\mathbf{D}$ through a closed surface S is equal to the enclosed charge. There is an identity in vector calculus, known as Gauss's theorem, which states that the integral of the flux of any vector field through a closed surface equals the integral of the divergence of the field over the volume enclosed by the surface. This holds for any vector field, but using the $\mathbf{D}$ field, Gauss's theorem states

$$\oint_S \mathbf{D} \cdot \mathbf{ds} = \int_V \nabla \cdot \mathbf{D} dv \tag{2.31}$$

where V is the enclosed volume and dv is a differential volume element. Note that the left-hand side of (2.31) is the left-hand side of (2.4).

The right-hand side of (2.4) is the enclosed charge Q_{enc} which could be determined either by evaluating the left-hand side of (2.4) or by integration of the charge density ρ_v over the volume enclosed by S. (This is similar to determining the mass of an object by integrating its mass density over its volume.) Thus,

$$Q_{\text{enc}} = \int_V \rho_v dv. \tag{2.32}$$

Equating the right-hand sides of (2.31) and (2.32) yields

$$\int_V \nabla \cdot \mathbf{D} dv = \int_V \rho_v dv. \tag{2.33}$$

Since this must hold over an arbitrary volume, the integrands must be equal which yields (2.8).

Another useful identity from vector calculus is Stokes' theorem which states that the integral of a vector field over any closed path is equal to the integral of the curl of that field over a surface which has that path as its border. Again, this holds for any vector field, but using the electric field as an example one can write

$$\oint_L \mathbf{E} \cdot \mathbf{d}\ell = \int_S \nabla \times \mathbf{E} \cdot \mathbf{ds}. \tag{2.34}$$

The surface normal is assumed to follow the right-hand convention so that when the fingers of the right hand are oriented along the path of the loop, the thumb points in the positive direction of the surface normal.

Static electric fields are conservative which means that the net work required to move a charge in a closed path is always zero. Along some portion of the path positive work would have to be done to push the charge against the field, but this amount of work would be given back by the field as the charge travels along the remaining portions of the path. The integrand on the left-hand side in (2.34) is the field dotted with an incremental length. If the integrand were multiplied by a unit positive charge, the integrand would represent work, since charge times field is force and force times distance is work. Because the electric field is conservative, the integral on the left-hand side of (2.34) must be zero. Naturally this implies that the integral on the right-hand side must also be zero. Since this holds for any loop L (or, similarly, any surface S), the integrand itself must be zero. Equating the integrand to zero yields (2.7).

2.8 Electric Field Boundary Conditions

Consider an interface between two homogeneous regions. Because electric flux density only begins or ends on charge, the normal component of $\mathbf{D}$ can only change at the interface if there is charge on the interface, i.e., surface charge is present. This can be stated mathematically as

$$\hat{\mathbf{n}} \cdot (\mathbf{D}_1 - \mathbf{D}_2) = \rho_s \tag{2.35}$$

where ρ_s is a surface charge density (C/m^2), $\hat{\mathbf{n}}$ is a unit vector normal to the surface, and $\mathbf{D}_1$ and $\mathbf{D}_2$ are the field to either side of the interface. One should properly argue this boundary condition by an application of Gauss's law for a small volume surrounding the surface, but such details are left to other classes (this is just a review!). If no charge is present, the normal components must be equal

$$\hat{\mathbf{n}} \cdot \mathbf{D}_1 = \hat{\mathbf{n}} \cdot \mathbf{D}_2. \tag{2.36}$$

The boundary conditions on the tangential component of the electric field can be determined by integrating the electric field over a closed loop which is essentially a rectangle which encloses a portion of the interface. By letting the sides shrink to zero and keeping the "top" and "bottom" of the rectangle small but finite (so that they are tangential to the surface), one essentially has that the field over the top must be the same as the field over the bottom (owing to the fact that total integral must be zero since the field is conservative). Stated mathematical, the boundary condition is

$$\hat{\mathbf{n}} \times (\mathbf{E}_1 - \mathbf{E}_2) = 0. \tag{2.37}$$

2.9 Conductivity and Perfect Electric Conductors

It is possible for the charge in materials to move under the influence of an electric field such that currents flow. If the material has a non-zero conductivity σ, the current density is given by

$$\mathbf{J} = \sigma \mathbf{E}. \tag{2.38}$$

The current density has units of A/m^2 and the conductivity has units of S/m.

If charge is building up or decaying in a particular region, the divergence of the current density must be non-zero. If the divergence is zero, that implies as much current leaves a point as enters it and there is no build-up or decay of charge. This can be stated as

$$\nabla \cdot \mathbf{J} = -\frac{\partial \rho_v}{\partial t}. \tag{2.39}$$

If the divergence is positive, the charge density must be decreasing with time (so the negative sign will bring the two into agreement). This equation is a statement of charge conservation.

Perfect electric conductors (PECs) are materials where it is assumed that the conductivity approaches infinity. If the fields were non-zero in a PEC, that would imply the current was infinite. Since infinite currents are not allowed, the fields inside a PEC are required to be zero. This subsequently requires that the tangential electric field at the surface of a PEC is zero (since tangential fields are continuous across an interface and the fields inside the PEC are zero). Correspondingly, the normal component of the electric flux density $\mathbf{D}$ at the surface of a PEC must equal the charge density at the surface of the PEC. Since the fields inside a PEC are zero, all points of the PEC must be at the same potential.

2.10 Magnetic Fields

Magnetic fields circulate around, but they do not terminate on anything—there is no (known) magnetic charge. Nevertheless, it is often convenient to define magnetic charge and magnetic current. These fictions allow one to simplify various problems such as integral formulations of scattering problems. However for now we will stick to reality and say they do not exist.

The magnetic flux density $\mathbf{B}$ is somewhat akin to the electric field in that the force on a charge in motion is related to $\mathbf{B}$. If a charge Q is moving with velocity $\mathbf{u}$ in a field $\mathbf{B}$, it experiences a force

$$\mathbf{F} = Q\mathbf{u} \times \mathbf{B}. \tag{2.40}$$

Because $\mathbf{B}$ determines the force on a charge, it must account for all sources of magnetic field. When material is present, the charge in the material can have motion (or rotation) which influences the magnetic flux density.

Alternatively, similar to the electric flux density, we define the magnetic field $\mathbf{H}$ which ignores the local effects of material. These fields are related by

$$\mathbf{B} = \mu_r \mu_0 \mathbf{H} = \mu \mathbf{H} \tag{2.41}$$

where μ_r is the relatively permeability, μ_0 is the permeability of free space equal to $4\pi \times 10^{-7}$ H/m, and μ is simply the permeability. Typically the relative permeability is greater than unity (although usually only by a small amount) which implies that when a material is present the magnetic flux density is larger than when there is only free space.

Charge in motion is the source of magnetic fields. If a current I flows over an incremental distance $\mathbf{d}\ell$, it will produce a magnetic field given by:

$$\mathbf{H} = \frac{I\mathbf{d}\ell \times \mathbf{a}_r}{4\pi r^2} \tag{2.42}$$

where $\mathbf{a}_r$ points from the location of the filament of current to the observation point and r is the distance between the filament and the observation point. Equation (2.42) is known as the Biot-Savart equation. Of course, because of the conservation of charge, a current cannot flow over just a filament and then disappear. It must flow along some path. Thus, the magnetic field due to a loop of current would be given by

$$\mathbf{H} = \oint_L \frac{I\mathbf{d}\ell \times \mathbf{a}_r}{4\pi r^2}. \tag{2.43}$$

If the current was flowing throughout a volume or over a surface, the integral would be correspondingly changed to account for the current wherever it flowed.

From (2.43) one sees that currents (which are just another way of saying charge in motion) are the source of magnetic fields. Because of the cross-product in (2.42) and (2.43), the magnetic field essentially swirls around the current. If one integrates the magnetic field over a closed path, the result is the current enclosed by that path

$$\oint_L \mathbf{H} \cdot \mathbf{d}\ell = I_{\text{enc}}. \tag{2.44}$$

The enclosed current I_{enc} is the current that passes through the surface S which is bound by the loop L.

The left-hand side of (2.44) can be converted to a surface integral by employing Stokes' theorem while the right-hand side can be related to the current density by integrating over the surface of the loop. Thus,

$$\oint_L \mathbf{H} \cdot \mathbf{d}\ell = \int_S \nabla \times \mathbf{H} \cdot \mathbf{ds} = I_{\text{enc}} = \int_S \mathbf{J} \cdot \mathbf{ds}. \tag{2.45}$$

Since this must be true for every loop (and surface), the integrands of the second and fourth terms can be equated. This yields

$$\nabla \times \mathbf{H} = \mathbf{J}. \tag{2.46}$$

The last equation needed to characterize static fields is

$$\nabla \cdot \mathbf{B} = 0. \tag{2.47}$$

This is the mathematical equivalent of saying there is no magnetic charge.

2.11 Magnetic Field Boundary Conditions

Note that the equation governing $\mathbf{B}$ is similar to the equation which governed $\mathbf{D}$. In fact, since the right-hand side is always zero, the equation for $\mathbf{B}$ is simpler. The arguments used to obtain the boundary condition for the normal component of the $\mathbf{D}$ field can be applied directly to the $\mathbf{B}$ field. Thus,

$$\hat{\mathbf{n}} \cdot (\mathbf{B}_1 - \mathbf{B}_2) = 0. \tag{2.48}$$

For the magnetic field, an integration path is constructed along the same lines as the one used to determine the boundary condition on the electric field. Note that the equations governing $\mathbf{E}$ and $\mathbf{H}$ are similar except that the one for $\mathbf{H}$ has a non-zero right-hand side. If the current density is zero over the region of interest, then there is really no distinction between the two and one can say that the tangential magnetic fields must be equal across a boundary. However, if a surface current exists on the interface, there may be a discontinuity in the tangential fields. The boundary condition is given by

$$\hat{\mathbf{n}} \times (\mathbf{H}_1 - \mathbf{H}_2) = \mathbf{K} \tag{2.49}$$

where $\mathbf{K}$ is the surface current density (with units of A/m).

2.12 Summary of Static Fields

When a system is not changing with respect to time, the governing equations are

$$\nabla \cdot \mathbf{D} = \rho_v, \tag{2.50}$$

$$\nabla \cdot \mathbf{B} = 0, \tag{2.51}$$

$$\nabla \times \mathbf{E} = 0, \tag{2.52}$$

$$\nabla \times \mathbf{H} = \mathbf{J}. \tag{2.53}$$

If a loop carries a current but is otherwise neutral, it will produce a magnetic field and only a magnetic field. If a charge is stationary, it will produce an electric field and only an electric field. The charge will not "feel" the loop current and the current loop will not feel the stationary charge (at least approximately). The magnetic field and electric field are decoupled. If a charge Q moves with velocity $\mathbf{u}$ in the presence of both an electric field and a magnetic field, the force on the charge is the sum of the forces due to the electric and magnetic fields

$$\mathbf{F} = Q(\mathbf{E} + \mathbf{u} \times \mathbf{B}). \tag{2.54}$$

2.13 Time Varying Fields

What happens when a point charge moves? We know that charge in motion gives rise to a magnetic field, but if the charge is moving, its associated electric field must also be changing. Thus, when a system is time-varying the electric and magnetic fields must be coupled.

There is a vector identity that the divergence of the curl of any vector field is identically zero. Taking the divergence of both sides of (2.53) yields

$$\nabla \cdot \nabla \times \mathbf{H} = \nabla \cdot \mathbf{J} = -\frac{\partial \rho_v}{\partial t} \tag{2.55}$$

where the conservation of charge equation was used to write the last equality. Since the first term must be zero, this implies that $\partial \rho_v / \partial t$ must also be zero. However, that is overly restrictive. In general, for a time-varying system, the charge density will change with respect to time. Therefore something must be wrong with (2.53) as it pertains to time-varying fields. It was Maxwell who recognized that by adding the temporal derivative of the electric flux density to the right-hand side of (2.53) the equation would still be valid for the time-varying case. The correct equation is given by

$$\nabla \times \mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t} \tag{2.56}$$

The term $\partial \mathbf{D} / \partial t$ is known as the displacement current while $\mathbf{J}$ is typically called the conduction current. Equation (2.56) is known as Ampere's law.

Taking the divergence of the right-hand side of (2.56) yields

$$\nabla \cdot \mathbf{J} + \frac{\partial \nabla \cdot \mathbf{D}}{\partial t} = -\frac{\partial \rho_v}{\partial t} + \frac{\partial \nabla \cdot \mathbf{D}}{\partial t} = -\frac{\partial \rho_v}{\partial t} + \frac{\partial \rho_v}{\partial t} = 0 \tag{2.57}$$

where use was made of (2.8) and the conservation of charge equation (2.39).

The electromotive force (EMF) is the change in potential over some path. It has been observed experimentally that when a magnetic field is time-varying there is a non-zero EMF over a closed path which encloses the varying field (i.e., the electric field is no longer conservative).

The symbol λ is often use to represent total magnetic flux through a given surface, i.e.,

$$\lambda = \int_S \mathbf{B} \cdot \mathbf{ds}. \tag{2.58}$$

For time-varying fields, the EMF over a closed path L can be written

$$V_{\text{emf}} = \frac{d\lambda}{dt}, \tag{2.59}$$

$$-\oint_L \mathbf{E}\cdot\mathbf{d}\ell = \frac{d}{dt}\int_S \mathbf{B}\cdot\mathbf{ds}, \tag{2.60}$$

$$-\int_S \nabla\times\mathbf{E}\cdot\mathbf{ds} = \int_S \frac{\partial\mathbf{B}}{\partial t}\cdot\mathbf{ds}, \tag{2.61}$$

where Stokes' theorem was used to write the last equation. Since this equality holds over any surface, the integrands must be equal. This yields

$$\nabla\times\mathbf{E} = -\frac{\partial\mathbf{B}}{\partial t} \tag{2.62}$$

which is known as Faraday's law.

2.14 Summary of Time-Varying Fields

When a system is changing with respect to time, the governing equations are

$$\nabla\cdot\mathbf{D} = \rho_v, \tag{2.63}$$

$$\nabla\cdot\mathbf{B} = 0, \tag{2.64}$$

$$\nabla\times\mathbf{E} = -\frac{\partial\mathbf{B}}{\partial t}, \tag{2.65}$$

$$\nabla\times\mathbf{H} = \mathbf{J}+\frac{\partial\mathbf{D}}{\partial t}. \tag{2.66}$$

Note that the divergence equations are unchanged from the static case. The two curl equations have picked up terms which couple the electric and magnetic fields. Since the additional terms both involve temporal derivatives, they go to zero in the static case and the equations reduce to those which governed static fields.

For time-varying fields the same boundary conditions hold as in the static case.

2.15 Wave Equation in a Source-Free Region

Equations (2.63)–(2.66) provide a set of coupled differential equations. In the FDTD method we will be dealing directly with the two curl equations. We will stick to the coupled equations and solve them directly. However, it is also possible to decouple these equations. As an example, taking the curl of both sides of (2.65) yields

$$\nabla\times\nabla\times\mathbf{E} = -\nabla\times\frac{\partial\mathbf{B}}{\partial t} = -\mu\nabla\times\frac{\partial\mathbf{H}}{\partial t}. \tag{2.67}$$

There is a vector identity that the curl of the curl of any field is given by

$$\nabla\times\nabla\times\mathbf{E} = \nabla(\nabla\cdot\mathbf{E}) - \nabla^2\mathbf{E}. \tag{2.68}$$

(This is true for any field, not just the electric field as shown here.) In a source-free region, there is no free charge present, $\rho_v = 0$, and hence the divergence of the electric field is zero ($\nabla \cdot \mathbf{D} = \epsilon \nabla \cdot \mathbf{E} = 0$). Thus (2.67) can be written

$$\nabla^2 \mathbf{E} = \mu \frac{\partial}{\partial t}(\nabla \times \mathbf{H}). \tag{2.69}$$

Keeping in mind that we are considering a source-free region so that J would be zero, we now use (2.66) to write

$$\begin{aligned} \nabla^2 \mathbf{E} &= \mu \frac{\partial}{\partial t} \frac{\partial \mathbf{D}}{\partial t}, & (2.70) \\ &= \mu\epsilon \frac{\partial^2 \mathbf{E}}{\partial t^2}. & (2.71) \end{aligned}$$

Equation (2.71) is the wave equation for the electric field and is often written

$$\nabla^2 \mathbf{E} - \frac{1}{c^2} \frac{\partial^2 \mathbf{E}}{\partial t^2} = 0 \tag{2.72}$$

where $c = 1/\sqrt{\mu\epsilon}$. Had we decoupled the equations to solve $\mathbf{H}$ instead of $\mathbf{E}$, we would still obtain the same equation (except with $\mathbf{H}$ replacing $\mathbf{E}$).

2.16 One-Dimensional Solutions to the Wave Equation

The wave equation which governs either the electric or magnetic field in one dimension in a source-free region can be written

$$\frac{\partial^2 f(x,t)}{\partial x^2} - \mu\epsilon \frac{\partial^2 f(x,t)}{\partial t^2} = 0. \tag{2.73}$$

We make the claim that any function $f(\xi)$ is a solution to this equation provided that f is twice differentiable and ξ is replaced with $t \pm x/c$ where $c = 1/\sqrt{\mu\epsilon}$. Thus, we have

$$f(\xi) = f(t \pm x/c) = f(x,t). \tag{2.74}$$

The first derivatives of this function can be obtained via the chain rule. Keeping in mind that

$$\begin{aligned} \frac{\partial \xi}{\partial t} &= 1, & (2.75) \\ \frac{\partial \xi}{\partial x} &= \pm \frac{1}{c}, & (2.76) \end{aligned}$$

the first derivatives can be written

$$\begin{aligned} \frac{\partial f(\xi)}{\partial t} &= \frac{\partial f(\xi)}{\partial \xi} \frac{\partial \xi}{\partial t} = \frac{\partial f(\xi)}{\partial \xi}, & (2.77) \\ \frac{\partial f(\xi)}{\partial x} &= \frac{\partial f(\xi)}{\partial \xi} \frac{\partial \xi}{\partial x} = \pm \frac{1}{c} \frac{\partial f(\xi)}{\partial \xi}. & (2.78) \end{aligned}$$

Employing the chain rule in a similar fashion, the second derivatives can be written as

$$\frac{\partial}{\partial t}\left(\frac{\partial f(\xi)}{\partial t}\right) = \frac{\partial}{\partial t}\left(\frac{\partial f(\xi)}{\partial \xi}\right) = \frac{\partial^2 f(\xi)}{\partial \xi^2}\frac{\partial \xi}{\partial t} = \frac{\partial^2 f(\xi)}{\partial \xi^2}, \tag{2.79}$$

$$\frac{\partial}{\partial x}\left(\frac{\partial f(\xi)}{\partial x}\right) = \frac{\partial}{\partial x}\left(\pm\frac{1}{c}\frac{\partial f(\xi)}{\partial \xi}\right) = \pm\frac{1}{c}\frac{\partial^2 f(\xi)}{\partial \xi^2}\frac{\partial \xi}{\partial x} = \frac{1}{c^2}\frac{\partial^2 f(\xi)}{\partial \xi^2}. \tag{2.80}$$

Thus, (2.79) and (2.80) show that

$$\frac{\partial^2 f}{\partial t^2} = \frac{\partial^2 f}{\partial \xi^2} \tag{2.81}$$

$$\frac{\partial^2 f}{\partial x^2} = \frac{1}{c^2}\frac{\partial^2 f}{\partial \xi^2}. \tag{2.82}$$

Substituting these into (2.73) yields

$$\frac{1}{c^2}\frac{\partial^2 f}{\partial \xi^2} - \frac{1}{c^2}\frac{\partial^2 f}{\partial \xi^2} = 0. \tag{2.83}$$

The two terms on the left-hand side cancel, thus satisfying the equation.

Consider a constant argument of f, say $t - x/c = 0$. Assume this argument is obtain by simultaneously having $t = 0$ and $x = 0$. Now, let time advance by one second, i.e., $t = 1$ s. How must the position x change to maintain an argument of zero? Solving for x yields $x = c(1\text{ s})$. In other words to move along with the field so as to maintain the value $f(0)$ (whatever that value happens to be), at time zero, we would be at position zero. At time one second, we would have to have moved to the location $x = c(1\text{ s})$. Speed is change in position over change in time. Thus the speed with which we are moving is $x/t = c(1\text{ s})/(1\text{ s}) = c$.

In these notes c will typically be used to represent the speed of light in free space. Using the permittivity and permeability of free space, we obtain $c = 1/\sqrt{\epsilon_0\mu_0} \approx 3 \times 10^8$ m/s.

Chapter 3

Introduction to the Finite-Difference Time-Domain Method: FDTD in 1D

3.1 Introduction

The finite-difference time-domain (FDTD) method is arguably the simplest, both conceptually and in terms of implementation, of the full-wave techniques used to solve problems in electromagnetics. It can accurately tackle a wide range of problems. However, as with all numerical methods, it does have its share of artifacts and the accuracy is contingent upon the implementation. The FDTD method can solve complicated problems, but it is generally computationally expensive. Solutions may require a large amount of memory and computation time. The FDTD method loosely fits into the category of "resonance region" techniques, i.e., ones in which the characteristic dimensions of the domain of interest are somewhere on the order of a wavelength in size. If an object is very small compared to a wavelength, quasi-static approximations generally provide more efficient solutions. Alternatively, if the wavelength is exceedingly small compared to the physical features of interest, ray-based methods or other techniques may provide a much more efficient way to solve the problem.

The FDTD method employs finite differences as approximations to both the spatial and temporal derivatives that appear in Maxwell's equations (specifically Ampere's and Faraday's laws). Consider the Taylor series expansions of the function $f(x)$ expanded about the point x_0 with an offset of $\pm\delta/2$:

$$f\left(x_0+\frac{\delta}{2}\right) = f(x_0)+\frac{\delta}{2}f'(x_0)+\frac{1}{2!}\left(\frac{\delta}{2}\right)^2 f''(x_0)+\frac{1}{3!}\left(\frac{\delta}{2}\right)^3 f'''(x_0)+\ldots, \quad (3.1)$$

$$f\left(x_0-\frac{\delta}{2}\right) = f(x_0)-\frac{\delta}{2}f'(x_0)+\frac{1}{2!}\left(\frac{\delta}{2}\right)^2 f''(x_0)-\frac{1}{3!}\left(\frac{\delta}{2}\right)^3 f'''(x_0)+\ldots \quad (3.2)$$

where the primes indicate differentiation. Subtracting the second equation from the first yields

$$f\left(x_0+\frac{\delta}{2}\right)-f\left(x_0-\frac{\delta}{2}\right)=\delta f'(x_0)+\frac{2}{3!}\left(\frac{\delta}{2}\right)^3 f'''(x_0)+\ldots \quad (3.3)$$

Lecture notes by John Schneider. `fdtd-intro.tex`

Dividing by δ produces

$$\frac{f\left(x_0+\frac{\delta}{2}\right)-f\left(x_0-\frac{\delta}{2}\right)}{\delta}=f'(x_0)+\frac{1}{3!}\frac{\delta^2}{2^2}f'''(x_0)+\ldots \tag{3.4}$$

Thus the term on the left is equal to the derivative of the function at the point x_0 plus a term which depends on δ^2 plus an infinite number of other terms which are not shown. For the terms which are not shown, the next would depend on δ^4 and all subsequent terms would depend on even higher powers of δ. Rearranging slightly, this relationship is often stated as

$$\left.\frac{df(x)}{dx}\right|_{x=x_0}=\frac{f\left(x_0+\frac{\delta}{2}\right)-f\left(x_0-\frac{\delta}{2}\right)}{\delta}+O(\delta^2). \tag{3.5}$$

The "big-Oh" term represents all the terms that are not explicitly shown and the value in parentheses, i.e., δ^2, indicates the lowest order of δ in these hidden terms. If δ is sufficiently small, a reasonable approximation to the derivative may be obtained by simply neglecting all the terms represented by the "big-Oh" term. Thus, the central-difference approximation is given by

$$\left.\frac{df(x)}{dx}\right|_{x=x_0}\approx\frac{f\left(x_0+\frac{\delta}{2}\right)-f\left(x_0-\frac{\delta}{2}\right)}{\delta}. \tag{3.6}$$

Note that the central difference provides an approximation of the derivative of the function at x_0, but the function is not actually sampled there. Instead, the function is sampled at the neighboring points $x_0+\delta/2$ and $x_0-\delta/2$. Since the lowest power of δ being ignored is second order, the central difference is said to have second-order accuracy or second-order behavior. This implies that if δ is reduced by a factor of 10, the error in the approximation should be reduced by a factor of 100 (at least approximately). In the limit as δ goes to zero, the approximation becomes exact.

One can construct higher-order central differences. In order to get higher-order behavior, more terms, i.e., more sample points, must be used. Appendix A presents the construction of a fourth-order central difference. The use of higher-order central differences in FDTD schemes is certainly possible, but there are some complications which arise because of the increased "stencil" of the difference operator. For example, when a PEC is present, it is possible that the difference operator will extend into the PEC prematurely or it may extend to the other side of a PEC sheet. Because of these types of issues, we will only consider the use of second-order central difference.

3.2 The Yee Algorithm

The FDTD algorithm as first proposed by Kane Yee in 1966 employs second-order central differences. The algorithm can be summarized as follows:

1. Replace all the derivatives in Ampere's and Faraday's laws with finite differences. Discretize space and time so that the electric and magnetic fields are staggered in both space and time.

2. Solve the resulting difference equations to obtain "update equations" that express the (unknown) future fields in terms of (known) past fields.

3. Evaluate the magnetic fields one time-step into the future so they are now known (effectively they become past fields).

4. Evaluate the electric fields one time-step into the future so they are now known (effectively they become past fields).

5. Repeat the previous two steps until the fields have been obtained over the desired duration.

At this stage, the summary is probably a bit too abstract. One really needs an example to demonstrate the simplicity of the method. However, developing the full set of three-dimensional equations would be overkill and thus the algorithm will first be presented in one-dimension. As you will see, the extension to higher dimensions is quite simple.

3.3 Update Equations in 1D

Consider a one-dimensional space where there are only variations in the x direction. Assume that the electric field only has a z component. In this case Faraday's law can be written

$$-\mu\frac{\partial \mathbf{H}}{\partial t} = \nabla \times \mathbf{E} = \begin{vmatrix} \hat{\mathbf{a}}_x & \hat{\mathbf{a}}_y & \hat{\mathbf{a}}_z \\ \frac{\partial}{\partial x} & 0 & 0 \\ 0 & 0 & E_z \end{vmatrix} = -\hat{\mathbf{a}}_y \frac{\partial E_z}{\partial x}. \tag{3.7}$$

Thus H_y must be the only non-zero component of the magnetic field which is time varying. (Since the right-hand side of this equation has only a y component, the magnetic field may have non-zero components in the x and z directions, but they must be static. We will not be concerned with static fields here.) Knowing this, Ampere's law can be written

$$\epsilon\frac{\partial \mathbf{E}}{\partial t} = \nabla \times \mathbf{H} = \begin{vmatrix} \hat{\mathbf{a}}_x & \hat{\mathbf{a}}_y & \hat{\mathbf{a}}_z \\ \frac{\partial}{\partial x} & 0 & 0 \\ 0 & H_y & 0 \end{vmatrix} = \hat{\mathbf{a}}_z \frac{\partial H_y}{\partial x}. \tag{3.8}$$

The two scalar equations obtained from (3.7) and (3.8) are

$$\mu\frac{\partial H_y}{\partial t} = \frac{\partial E_z}{\partial x}, \tag{3.9}$$

$$\epsilon\frac{\partial E_z}{\partial t} = \frac{\partial H_y}{\partial x}. \tag{3.10}$$

The first equation gives the temporal derivative of the magnetic field in terms of the spatial derivative of the electric field. Conversely, the second equation gives the temporal derivative of the electric field in terms of the spatial derivative of the magnetic field. As will be shown, the first equation will be used to advance the magnetic field in time while the second will be used to advance the electric field. A method in which one field is advanced and then the other, and then the process is repeated, is known as a leap-frog method.

The next step is to replace the derivatives in (3.9) and (3.10) with finite differences. To do this, space and time need to be discretized. The following notation will be used to indicate the location where the fields are sampled in space and time

$$E_z(x,t) = E_z(m\Delta_x, q\Delta_t) = E_z^q[m]\,, \tag{3.11}$$

$$H_y(x,t) = H_y(m\Delta_x, q\Delta_t) = H_y^q[m]\,, \tag{3.12}$$

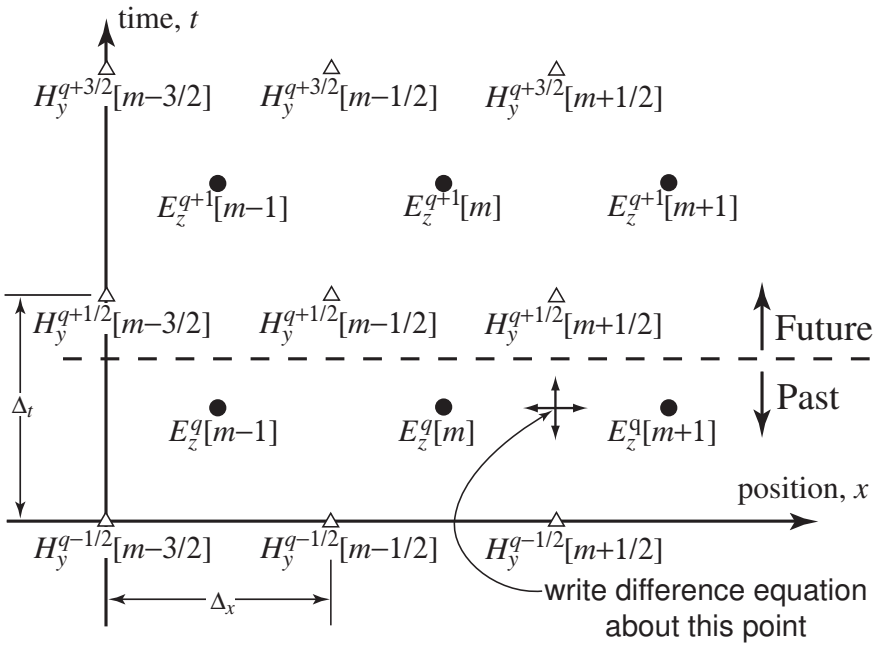


Figure 3.1: The arrangement of electric- and magnetic-field nodes in space and time. The electric-field nodes are shown as circles and the magnetic-field nodes as triangles. The indicated point is where the difference equation is expanded to obtain an update equation for H_y.

where Δ_x is the spatial offset between sample points and Δ_t is the temporal offset. The index m corresponds to the spatial step, effectively the spatial location, while the index q corresponds to the temporal step. When written as a superscript q still represents the temporal step—it is not an exponent. When implementing FDTD algorithms we will see that the spatial indices are used as array indices while the temporal index, which is essentially a global parameter, is not explicitly specified for each field location. Hence, it is reasonable to keep the spatial indices as an explicit argument while indicating the temporal index separately.

Although we only have one spatial dimension, time can be thought of as another dimension. Thus this is effectively a form of two-dimensional problem. The question now is: How should the electric and magnetic field sample points, also known as nodes, be arranged in space and time? The answer is shown in Fig. 3.1. The electric-field nodes are shown as circles and the magnetic-field nodes as triangles. Assume that all the fields below the dashed line are known—they are considered to be in the past—while the fields above the dashed line are future fields and hence unknown. The FDTD algorithm provides a way to obtain the future fields from the past fields.

As indicated in Fig. 3.1, consider Faraday's law at the space-time point $((m+1/2)\Delta_x, q\Delta_t)$

$$\mu \frac{\partial H_y}{\partial t}\bigg|_{(m+1/2)\Delta_x, q\Delta_t} = \frac{\partial E_z}{\partial x}\bigg|_{(m+1/2)\Delta_x, q\Delta_t}. \tag{3.13}$$

The temporal derivative is replaced by a finite difference involving $H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}\right]$ and $H_y^{q-\frac{1}{2}}\left[m+\frac{1}{2}\right]$ (i.e., the magnetic field at a fixed location but two different times) while the spatial derivative is replaced by a finite difference involving $E_z^q[m+1]$ and $E_z^q[m]$ (i.e., the electric field at two different

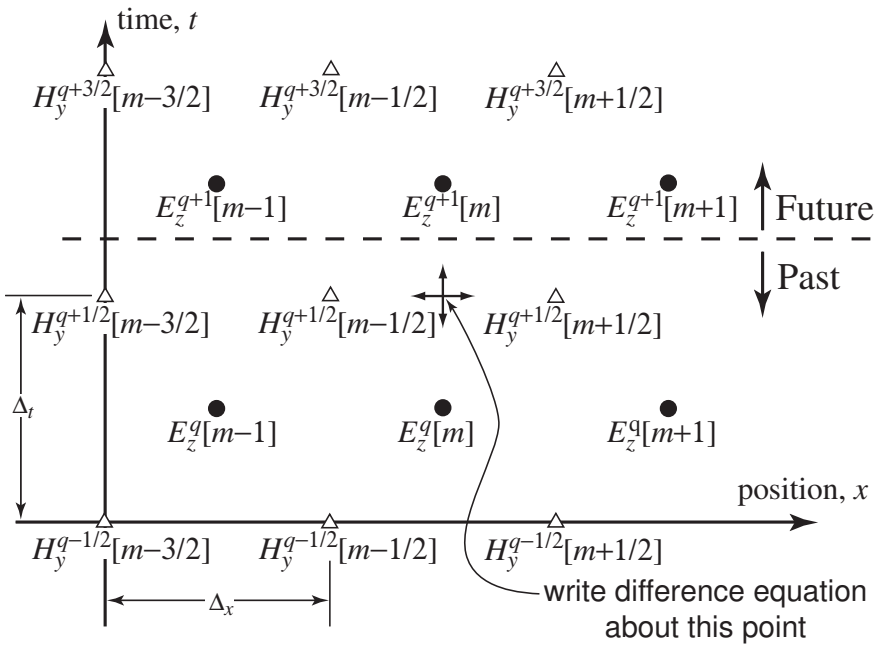


Figure 3.2: Space-time after updating the magnetic field. The dividing line between future and past values has moved forward a half temporal step. The indicated point is where the difference equation is written to obtain an update equation for E_z.

locations but one time). This yields

$$\mu \frac{H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}\right] - H_y^{q-\frac{1}{2}}\left[m+\frac{1}{2}\right]}{\Delta_t} = \frac{E_z^q[m+1] - E_z^q[m]}{\Delta_x}. \tag{3.14}$$

Solving this for $H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}\right]$ yields

$$H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}\right] = H_y^{q-\frac{1}{2}}\left[m+\frac{1}{2}\right] + \frac{\Delta_t}{\mu\Delta_x}\left(E_z^q[m+1] - E_z^q[m]\right). \tag{3.15}$$

This is known as an update equation, specifically the update equation for the H_y field. It is a generic equation which can be applied to any magnetic-field node. It shows that the future value of H_y depends on only its previous value and the neighboring electric fields. After applying (3.15) to all the magnetic-field nodes, the dividing line between future and past values has advanced a half time-step. The space-time grid thus appears as shown in Fig. 3.2 which is identical to Fig. 3.1 except for the advancement of the past/future dividing line.

Now consider Ampere's law (3.10) applied at the space-time point $(m\Delta_x, (q+1/2)\Delta_t)$ which is indicated in Fig. 3.2:

$$\epsilon \left.\frac{\partial E_z}{\partial t}\right|_{m\Delta_x,(q+1/2)\Delta_t} = \left.\frac{\partial H_y}{\partial x}\right|_{m\Delta_x,(q+1/2)\Delta_t}. \tag{3.16}$$

Replacing the temporal derivative on the left with a finite difference involving $E_z^{q+1}[m]$ and $E_z^q[m]$ and replacing the spatial derivative on the right with a finite difference involving $H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}\right]$ and

$H_y^{q+\frac{1}{2}}\left[m-\frac{1}{2}\right]$ yields

$$\epsilon\frac{E_z^{q+1}[m]-E_z^q[m]}{\Delta_t}=\frac{H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}\right]-H_y^{q+\frac{1}{2}}\left[m-\frac{1}{2}\right]}{\Delta_x}. \tag{3.17}$$

Solving for $E_z^{q+1}[m]$ yields

$$E_z^{q+1}[m]=E_z^q[m]+\frac{\Delta_t}{\epsilon\Delta_x}\left(H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}\right]-H_y^{q+\frac{1}{2}}\left[m-\frac{1}{2}\right]\right). \tag{3.18}$$

Equation (3.18) is the update equation for the E_z field. The indices in this equation are generic so that the same equation holds for every E_z node. Similar to the update equation for the magnetic field, here we see that the future value of E_z depends on only its past value and the value of the neighboring magnetic fields.

After applying (3.18) to every electric-field node in the grid, the dividing line between what is known and what is unknown moves forward another one-half temporal step. One is essentially back to the situation depicted in Fig. 3.1—the future fields closest to the dividing line between the future and past are magnetics fields. They would be updated again, then the electric fields would be updated, and so on.

It is often convenient to represent the update coefficients $\Delta_t/\epsilon\Delta_x$ and $\Delta_t/\mu\Delta_x$ in terms of the ratio of how far energy can propagate in a single temporal step to the spatial step. The maximum speed electromagnetic energy can travel is the speed of light in free space $c=1/\sqrt{\epsilon_0\mu_0}$ and hence the maximum distance energy can travel in one time step is $c\Delta_t$ (in all the remaining discussions the symbol c will be reserved for the speed of light in free space). The ratio $c\Delta_t/\Delta_x$ is often called the Courant number which we label S_c. It plays an important role in determining the stability of a simulation (hence the use of S) and will be considered further later. Letting $\mu=\mu_r\mu_0$ and $\epsilon=\epsilon_r\epsilon_0$, the coefficients in (3.18) and (3.15) can be written

$$\frac{1}{\epsilon}\frac{\Delta_t}{\Delta_x}=\frac{1}{\epsilon_r\epsilon_0}\frac{\sqrt{\epsilon_0\mu_0}}{\sqrt{\epsilon_0\mu_0}}\frac{\Delta_t}{\Delta_x}=\frac{\sqrt{\epsilon_0\mu_0}}{\epsilon_r\epsilon_0}\frac{c\Delta_t}{\Delta_x}=\frac{1}{\epsilon_r}\sqrt{\frac{\mu_0}{\epsilon_0}}\frac{c\Delta_t}{\Delta_x}=\frac{\eta_0}{\epsilon_r}\frac{c\Delta_t}{\Delta_x}=\frac{\eta_0}{\epsilon_r}S_c \tag{3.19}$$

$$\frac{1}{\mu}\frac{\Delta_t}{\Delta_x}=\frac{1}{\mu_r\mu_0}\frac{\sqrt{\epsilon_0\mu_0}}{\sqrt{\epsilon_0\mu_0}}\frac{\Delta_t}{\Delta_x}=\frac{\sqrt{\epsilon_0\mu_0}}{\mu_r\mu_0}\frac{c\Delta_t}{\Delta_x}=\frac{1}{\mu_r}\sqrt{\frac{\epsilon_0}{\mu_0}}\frac{c\Delta_t}{\Delta_x}=\frac{1}{\mu_r\eta_0}\frac{c\Delta_t}{\Delta_x}=\frac{1}{\mu_r\eta_0}S_c \tag{3.20}$$

where $\eta_0=\sqrt{\mu_0/\epsilon_0}$ is the characteristic impedance of free space.

In FDTD simulations there are restrictions on how large a temporal step can be. If it is too large, the algorithm produces unstable results (i.e., the numbers obtained are completely meaningless and generally tend quickly to infinity). At this stage we will not consider a rigorous analysis of stability. However, thinking about the way fields propagate in an FDTD grid, it seems logical that energy should not be able to propagate any further than one spatial step for each temporal step, i.e., $c\Delta_t\le\Delta_x$. This is because in the FDTD algorithm each node only affects its nearest neighbors. In one complete cycle of updating the fields, the furthest a disturbance could propagate is one spatial step. It turns out that the optimum ratio for the Courant number (in terms of minimizing numeric errors) is also the maximum ratio. Hence, for the one-dimensional simulations considered initially, we will use

$$S_c=\frac{c\Delta_t}{\Delta_x}=1. \tag{3.21}$$

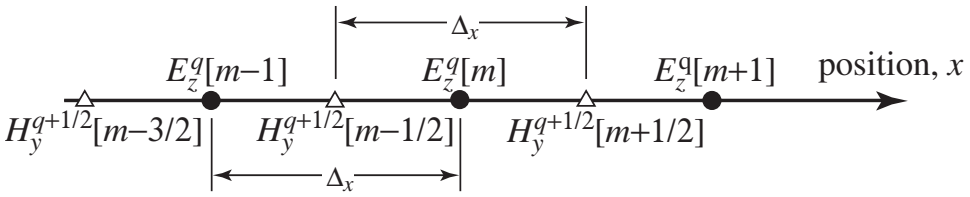


Figure 3.3: A one-dimensional FDTD space showing the spatial offset between the magnetic and electric fields.

When first obtaining the update equations for the FDTD algorithm, it is helpful to think in terms of space-time. However, treating time as an additional dimension can be awkward. Thus, in most situations it is more convenient to think in terms of a single spatial dimension where the electric and magnetic fields are offset a half spatial step from each other. This is depicted in Fig. 3.3. The temporal offset between the electric and magnetic field is always understood whether explicitly shown or not.

3.4 Computer Implementation of a One-Dimensional FDTD Simulation

Our goal now is to translate the update equations (3.15) and (3.18) into a usable computer program. The first step is to discard, at least to a certain extent, the superscripts—time is a global parameter and will be recorded in a single integer variable. Time is not something about which each node needs to be concerned.

Next, keep in mind that in most computer languages the equal sign is used as "the assignment operator." In C, the following is a perfectly valid statement

```
a = a+b;
```

In the usual mathematical sense, this statement is only true if b were zero. However, to a computer this statement means take the value of b, add it to the old value of a, and place the result back in the variable a. Essentially we are updating the value of a. In C this statement can be written more tersely as

```
a += b;
```

When writing a computer program to implement the FDTD algorithm, one does not bother trying to construct a program that explicitly uses offsets of one-half. Nodes are stored in arrays and, as is standard practice, individual array elements are specified with integer indices. Thus, the computer program (or, perhaps more correctly, the *author* of the computer program) implicitly incorporates the fact that electric and magnetic fields are offset while using only integer indices to specify location. As you will see, spatial location and the array index will be virtually synonymous.

For example, assume two arrays, `ez` and `hy`, are declared which will contain the E_z and H_y fields at 200 nodes

```
double ez[200], hy[200], imp0=377.0;
```

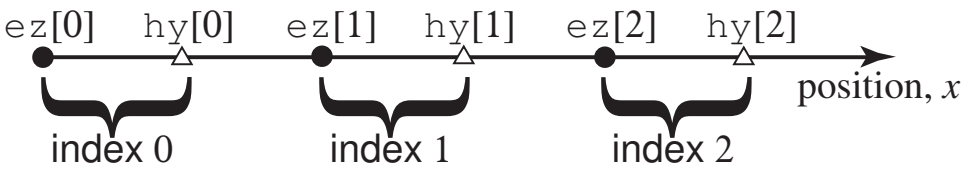


Figure 3.4: A one-dimensional FDTD space showing the assumed spatial arrangement of the electric- and magnetic-field nodes in the arrays `ez` and `hy`. Note that an electric-field node is assumed to exist to the left of the magnetic-field node with the same index.

The variable `imp0` is the characteristic impedance of free space and will be used in the following discussion (it is initialized to a value of 377.0 in this declaration). One should think of the elements in the `ez` and `hy` arrays as being offset from each other by a half spatial step even though the array values will be accessed using an integer index.

It is arbitrary whether one initially wishes to think of an `ez` array element as existing to the right or the left of an `hy` element with the same index (we assume "left" corresponds to descreasing values of x while "right" corresponds to increasing values). Here we will assume `ez` nodes are to the left of `hy` nodes with the same index. This is illustrated in Fig. 3.4 where `ez[0]` is to the left of `hy[0]`, `ez[1]` is to the left of `hy[1]`, and so on. In general, when a Courier font is used, e.g., `hy[m]`, we are considering an array and any offsets of one-half associated with that array are implicitly understood. When Times-Italic font is use, e.g., $H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}\right]$ we are discussing the field itself and offsets will be given explicitly.

Assuming a Courant number of unity ($S_c = 1$), the node `hy[1]` could be updated with a statement such as

```
hy[1] = hy[1] + (ez[2] - ez[1]) / imp0;
```

In general, any magnetic-field node can be updated with

```
hy[m] = hy[m] + (ez[m + 1] - ez[m]) / imp0;
```

For the electric-field nodes, the update equation can be written

```
ez[m] = ez[m] + (hy[m] - hy[m - 1]) * imp0;
```

These two update equations, placed in appropriate loops, are the engines that drive an FDTD simulation. However, there are a few obvious pieces missing from the puzzle before a useful simulation can be performed. These missing pieces include

1. Nodes at the end of the physical space do not have neighboring nodes to one side. For example, there is no `hy[-1]` node for the `ez[0]` node to use in its update equation. Similarly, if the arrays are declared with 200 element, there is no `ez[200]` available for `hy[199]` to use in its update equation (recall that the index of the last element in a C array is one less than the total number of elements—the array index represents the offset from the first element of the array). Therefore a standard update equation cannot be used at these nodes.

2. Only a constant impedance is used so only a homogeneous medium can be modeled (in this case free space).

3. As of yet there is no energy present in the field. If the fields are initially zero, they will remain zero forever.

The first issue can be addressed using absorbing boundary conditions (ABC's). There are numerous implementations one can use. In later material we will be consider only a few of the more popular techniques.

The second restriction can be removed by allowing the permittivity and permeability to change from node to node. However, in the interest of simplicity, we will continue to use a constant impedance for a little while longer.

The third problem can be overcome by initializing the fields to a non-zero state. However, this is cumbersome and typically not a good approach. Better solutions are to introduce energy via either a hardwired source, an additive source, or a total-field/scattered-field (TFSF) boundary. We will consider implementation of each of these approaches.

3.5 Bare-Bones Simulation

Let us consider a simulation of a wave propagating in free space where there are 200 electric- and magnetic-field nodes. The code is shown in Program 3.1.

Program 3.1 `1DbareBones.c`: Bare-bones one-dimensional simulation with a hard source.

```
1  /* Bare-bones 1D FDTD simulation with a hard source. */
2
3  #include <stdio.h>
4  #include <math.h>
5
6  #define SIZE 200
7
8  int main()
9  {
10   double ez[SIZE] = {0.}, hy[SIZE] = {0.}, imp0 = 377.0;
11   int qTime, maxTime = 250, mm;
12
13   /* do time stepping */
14   for (qTime = 0; qTime < maxTime; qTime++) {
15
16     /* update magnetic field */
17     for (mm = 0; mm < SIZE - 1; mm++)
18       hy[mm] = hy[mm] + (ez[mm + 1] - ez[mm]) / imp0;
19
20     /* update electric field */
21     for (mm = 1; mm < SIZE; mm++)
```

```
22        ez[mm] = ez[mm] + (hy[mm] - hy[mm - 1]) * imp0;
23
24      /* hardwire a source node */
25      ez[0] = exp(-(qTime - 30.) * (qTime - 30.) / 100.);
26
27      printf("%g\n", ez[50]);
28    } /* end of time-stepping */
29
30    return 0;
31  }
```

In the declaration of the field arrays in line 10, "=$\{$`0.`$\}$" has been added to ensure that these arrays are initialized to zero. (For larger arrays this is not an efficient approach for initializing the arrays and we will address this fact later.) The variable `qTime` is an integer counter that serves as the temporal index or time step. The total number of time steps in the simulation is dictated by the variable `maxTime` which is set to 250 in line 11 (250 was chosen arbitrarily—it can be any value desired).

Time-stepping is accomplished with the for-loop that begins on line 14. Embedded within this time-stepping loop are two additional (spatial) loops—one to update the magnetic field and the other to update the electric field. The magnetic-field update loop starting on line 17 excludes the last magnetic-field node in the array, `hy[199]`, since this node lacks one neighboring electric field. For now we will leave this node zero. The electric-field update loop in line 21 starts with a spatial index `m` of 1, i.e., it does not include `ez[0]` which is the first E_z node in the grid. The value of `ez[0]` is dictated by line 25 which is a Gaussian function that will have a maximum value of unity when the time counter `qTime` is 30. The first time through the loop, when `qTime` is zero, `ez[0]` will be set to $\exp(-9) \approx 1.2341 \times 10^{-4}$ which is small relative to the maximum value of the source. Line 27 prints the value of `ez[50]` to the screen, once for each time step. A plot of the output generated by this program is shown in Fig. 3.5.

Note that the output is a Gaussian. The excitation is introduced at `ez[0]` but the field is recorded at `ez[50]`. Because $c\Delta_t = \Delta_x$ in this simulation (i.e., the Courant number is unity), the field moves one spatial step for every time step. The separation between the source point and the observation point results in the observed signal being delayed by 50 time steps from what it was at the source. The source function has a peak at 30 time steps but, as can be seen from Fig. 3.5, the field at the observation point is maximum at time step 80.

Consider a slight modification to Program 3.1 where the simulation is run for 1000 time steps instead of 250 (i.e., `maxTime` is set to 1000 in line 11 instead of 250). The output obtained in this case is shown in Fig. 3.6. Why are there multiple peaks here and why are they both positive and negative?

The last magnetic-field node in the grid is initially zero and remains zero throughout the simulation. When the field encounters this node it essentially see a perfect magnetic conductor (PMC). To satisfy the boundary condition at this node, i.e., that the total magnetic field go to zero, a reflected wave is created which reverses the sign of the magnetic field but preserves the sign of the electric field. This phenomenon is considered in more detail in the next section. The second peak in Fig. 3.6 is this reflected wave. The reflected wave continues to travel in the negative direction

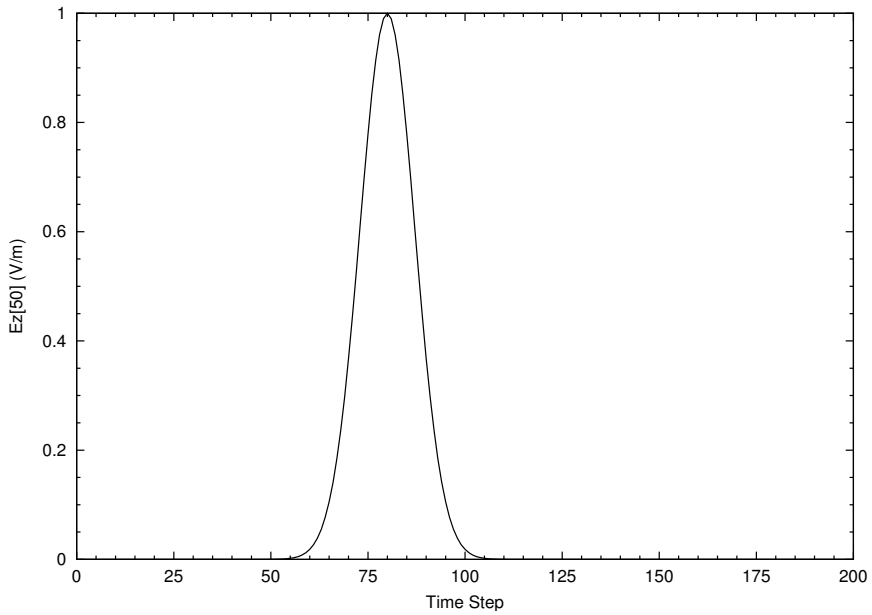


Figure 3.5: Output generated by Program 3.1.

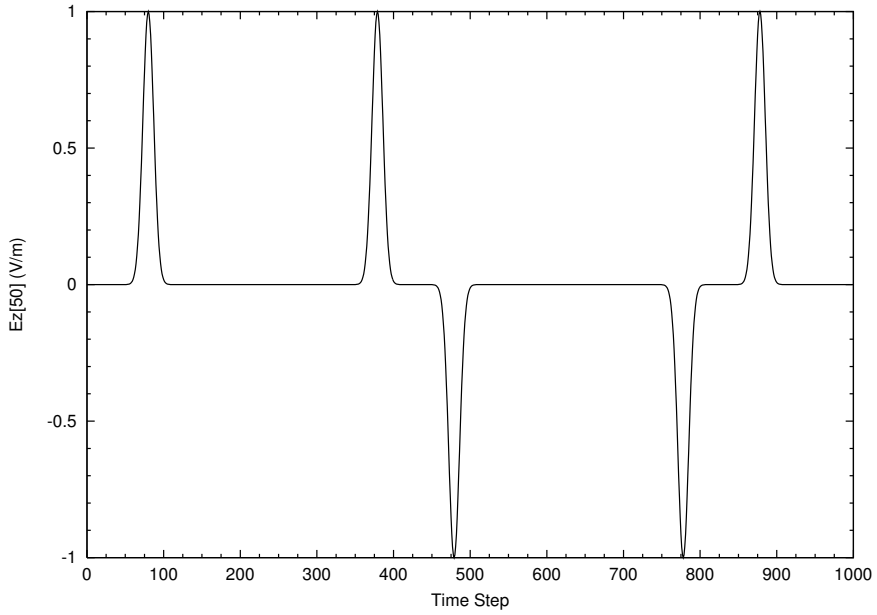


Figure 3.6: Output generated by Program 3.1 but with `maxTime` set to 1000.

until it encounters the first electric-field node `ez[0]`. This node has its value set by the source function and is oblivious to what is happening in the interior of the grid. In this particular case, by the time the reflected field reaches the left end of the grid, the source function has essentially gone to zero and nothing is going to change that. Thus the node `ez[0]` behaves like a perfect electric conductor (PEC). To satisfy the boundary conditions at this node, the wave is again reflected, but this time the electric field changes sign while the sign of the magnetic field is preserved. In this way the field which was introduced into the grid continues to bounce back and forth until the simulation is terminated. The simulation is of a resonator with one PMC wall and one PEC wall. (Note that the boundary condition at `ez[0]` is the same whether or not the source function has gone to zero. Any incoming field cannot change the value at `ez[0]` and hence a reflected wave must be generated which has equal magnitude but opposite sign from the incoming field.)

3.6 PMC Boundary in One Dimension

In Program 3.1 one side of the grid (the "right side") is terminated by a magnetic field which is always zero. It was observed that this node acts as a perfect magnetic conductor (PMC) which produces a reflected wave where the electric field is not inverted while the magnetic field is inverted. To understand fully why this is the case, let us consider the right side of a one-dimensional domain where 200 electric- and magnetic-field nodes are used to model free space. Assume the Courant number is unity and the impedance of free space is 377. The last node in the grid is `hy[199]` and it will always remain zero. The other nodes in the grid are updated using, in C notation:

```
ez[m] = ez[m] + (hy[m] - hy[m - 1]) * 377;        (3.22)
hy[m] = hy[m] + (ez[m + 1] - ez[m]) / 377;        (3.23)
```

Assume that a Dirac delta pulse, i.e., a unit amplitude pulse existing at a single electric-field node in space-time, is nearing the end of the grid. Table 3.1 shows the fields at progressive time-steps starting at a time q when the pulse has reached node `ez[198]`.

At time q node `ez[198]` is unity while `hy[197]` was set to $-1/377$ at the previous update of the magnetic fields. When the magnetic fields are updated at time $q+1/2$ the update equation (3.23) dictates that `hy[197]` be set to zero (the "old" value of the magnetic field cancels the contribution from the electric field). Meanwhile, `hy[198]` becomes $-1/377$. All other magnetic-field nodes will be zero.

Updating the electric field at time-step $q + 1$ results in `ez[198]` being set to zero while `ez[199]` is set to one—the pulse advances one spatial step to the right. If the normal update equation could be used at node `hy[199]`, at time $q + 3/2$ it would be set to $-1/377$. However, because there is no neighboring electric field to the right of `hy[199]`, the update equation cannot be used and, lacking an alternative way of calculating its value, `hy[199]` is left as zero. Thus at time $q + 3/2$ *all* the magnetic-field nodes in the grid are zero.

When the electric field is updated at time $q + 2$ essentially nothing happens. The electric fields are updated from their old values and the difference of surrounding magnetic fields. However all magnetic fields are zero. Thus the new electric field is the same as the old electric field.

At time $q + 5/2$ the unit pulse which exists at `ez[199]` causes `hy[198]` to become $1/377$ which is the negative of what it was two times steps ago. From this time forward, the pulse propagates back to the left with the electric field maintaining unit amplitude.

		node					
		ez[197]	hy[197]	ez[198]	hy[198]	ez[199]	hy[199]
time step	$q-1/2$		$-1/377$		0		0
	q	0		1		0	
	$q+1/2$		0		$-1/377$		0
	$q+1$	0		0		1	
	$q+3/2$		0		0		0
	$q+2$	0		0		1	
	$q+5/2$		0		$1/377$		0
	$q+3$	0		1		0	
	$q+5/2$		$1/377$		0		0
	$q+4$	1		0		0	

Table 3.1: Electric- and magnetic-field nodes at the "end" of arrays which have 200 elements, i.e., the last node is `hy[199]` which is always set to zero. A pulse of unit amplitude is propagating to the right and has arrived at `ez[198]` at time-step q. Time is advancing as one reads down the columns.

This discussion is for a single pulse, but any incident field could be treated as a string of pulses and then one would merely have to superimpose their values. This dicussion further supposes the Courant number is unity. When the Courant number is not unity the termination of the grid still behaves as a PMC wall, but the pulse will not propagate without distortion (it suffers dispersion because of the properties of the grid itself as will be discussed in more detail in Sec. 7.4).

If the grid were terminated on an electric-field node which was always set to zero, that node would behave as a perfect electric conductor. In that case the reflected electric field would have the opposite sign from the incident field while the magnetic field would preserve its sign. This is what happens to any field incident on the left side of the grid in Program 3.1.

3.7 Snapshots of the Field

In Program 3.1 the field at a single point is recorded to a file. Alternatively, it is often useful to view the fields over the entire computational domain at a single instant of time, i.e., take a "snapshot" that shows the field throughout space. Here we describe one way in which this can be conveniently implemented in C.

The approach adopted here will open a separate file for each snapshot. Each file will have a common base name, then a dot, and then a sequence number which will be called the frame number. So, for example, the files might be called `sim.0`, `sim.1`, `sim.2`, and so on. To accomplish this, the fragments shown in Fragments 3.2 and 3.3 would be added to a program (such as Program 3.1).

Fragment 3.2 Declaration of variables associated with taking snapshots. The base name is stored in the character array `basename` and the complete file name for each frame is stored in `filename`. Here the base name is initialized to `sim` but, if desired, the user could be prompted for the base

name. The integer `frame` is the frame number for each snapshot and is initialized to zero.

```
1  char basename[80] = "sim", filename[100];
2  int frame = 0;
3  FILE *snapshot;
```

Fragment 3.3 Code to generate the snapshots. This would be placed inside the time-stepping loop. The initial if statement ensures the electric field is recorded every tenth time-step.

```
1     /* write snapshot if time-step is a multiple of 10 */
2     if (qTime % 10 == 0) {
3       /* construct complete file name and increment frame counter */
4       sprintf(filename, "%s.%d", basename, frame++);
5
6       /* open file */
7       snapshot = fopen(filename, "w");
8
9       /* write data to file */
10      for (mm = 0; mm < SIZE; mm++)
11        fprintf(snapshot, "%g\n", ez[mm]);
12
13      /* close file */
14      fclose(snapshot);
15    }
```

In Fragment 3.2 the base name is initialized to `sim` but the user could be prompted for this. The integer variable `frame` is the frame (or snapshot) counter that will be incremented each time a snapshot is taken. It is initialized to zero. The file pointer `snapshot` is used for the output files.

The code shown in Fragment 3.3 would be placed inside the time-stepping loop of Program 3.1. Line 2 checks, using the modulo operator (`%`) if the time step is a multiple of 10. (10 was chosen somewhat arbitrarily. If snapshots were desired more frequently, a smaller value would be used. If snapshots were desired less frequently, a larger value would be used.) If the time step is a multiple of 10, the complete output-file name is constructed in line 4 by writing the file name to the string variable `filename`. (Since zero is a multiple of 10, the first snapshot that is taken corresponds to the fields at time zero. This data would be written to the file `sim.0`. Note that in Line 4 the frame number is incremented each time a file name is created. The file is opened in line 7 and the data is written using the loop starting in line 10. Finally, the file is closed in line 14.

Fig. 3.7 shows the snapshots of the field at time steps 20, 30, and 40 using essentially the same code as Program 3.1—the only difference being the addition of the code to take the snapshots. The corresponding files are `sim.2`, `sim.3`, and `sim.4`. In these snapshots the field can be seen entering the computational domain from the left and propagating to the right.

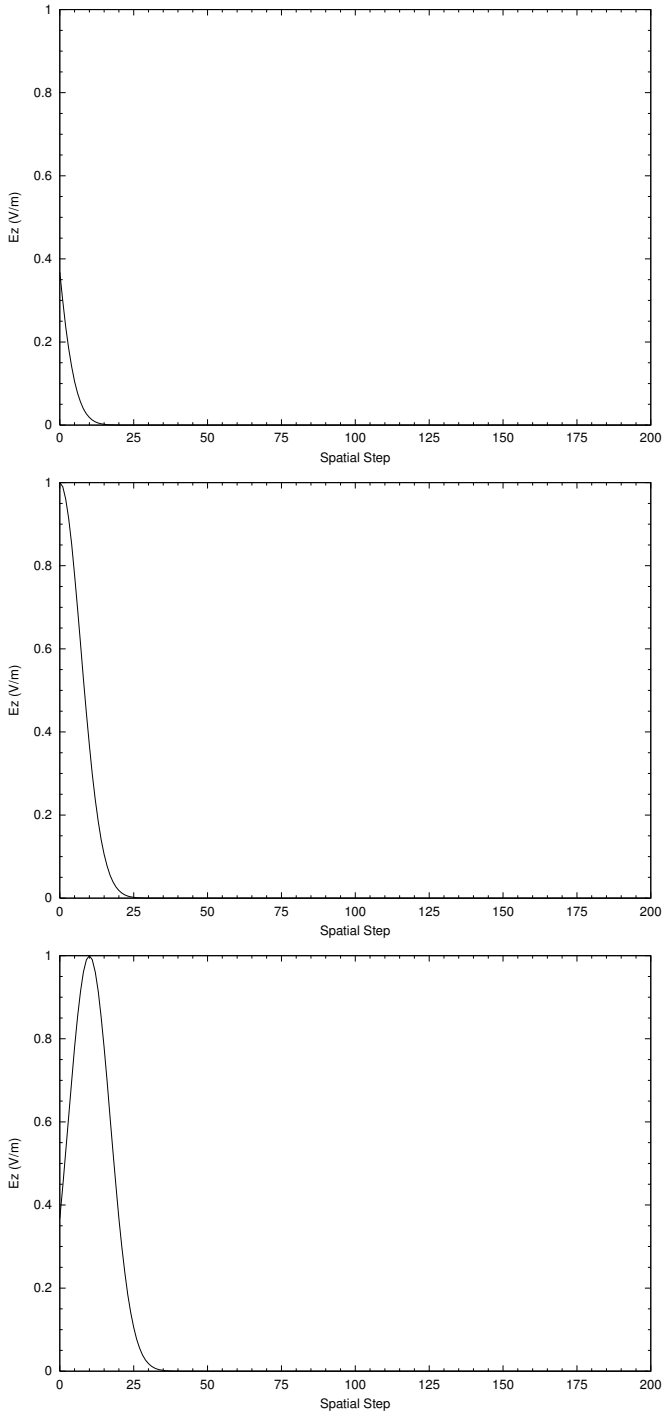


Figure 3.7: Snapshots taken at time-steps 20, 30, and 40 of the E_z field generated by Program 3.1. The field is seen to be propagating away from the hardwired source at the left end of the grid.

3.8 Additive Source

Hardwiring the source, as was done in Program 3.1, has the severe shortcoming that no energy can pass through the source node. This problem can be rectified by using an additive source. Consider Ampere's law with the current density term:

$$\nabla \times \mathbf{H} = \mathbf{J} + \epsilon \frac{\partial \mathbf{E}}{\partial t}. \tag{3.24}$$

The current density $\mathbf{J}$ can represent both the conduction current due to flow of charge in a material under the influence of the electric field, i.e., current given by $\sigma\mathbf{E}$, as well as the current associated with any source, i.e., an "impressed current." At this point we are just interested in the source aspect of $\mathbf{J}$ and will return to the issue of finite conductivity in Sec. 3.12 and Sec. 5.7. Rearranging (3.24) slightly yields

$$\frac{\partial \mathbf{E}}{\partial t} = \frac{1}{\epsilon}\nabla \times \mathbf{H} - \frac{1}{\epsilon}\mathbf{J}. \tag{3.25}$$

This equation gives the temporal derivative of the electric field in terms of the spatial derivative of the magnetic field—which is as before—and an additional term which can be thought of as the forcing function for the system. This current can be specified to be whatever is desired.

To translate (3.25) into a form suitable for the FDTD algorithm, the spatial derivatives are again expressed in terms of finite differences and then one solves for the future fields in terms of past fields. Recall that for Ampere's law, the update equation for $E_z^q[m]$ was obtained by applying finite differences at the space-time point $(m\Delta_x, (q+1/2)\Delta_t)$. Going through the exact same procedure but adding the source term yields

$$E_z^{q+1}[m] = E_z^q[m] + \frac{\Delta_t}{\epsilon\Delta_x}\left(H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}\right] - H_y^{q+\frac{1}{2}}\left[m-\frac{1}{2}\right]\right) - \frac{\Delta_t}{\epsilon}J_z^{q+\frac{1}{2}}[m]\,. \tag{3.26}$$

The source current could potentially be distributed over a number of nodes, but for the sake of introducing energy to the grid, it suffices to apply it to a single node.

In order to preserve the original update equation (which is sometimes handy when writing loops), (3.26) can be separated into two steps: first the usual update is applied and then the source term is added. For example:

$$E_z^{q+1}[m] = E_z^q[m] + \frac{\Delta_t}{\epsilon\Delta_x}\left(H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}\right] - H_y^{q+\frac{1}{2}}\left[m-\frac{1}{2}\right]\right) \tag{3.27}$$

$$E_z^{q+1}[m] = E_z^{q+1}[m] - \frac{\Delta_t}{\epsilon}J_z^{q+\frac{1}{2}}[m]\,. \tag{3.28}$$

In practice the source current might only exist at a single node in the 1D grid (as will be the case in the examples to come). Thus, (3.28) would be applied only at the node where the source current is non-zero.

Generally the amplitude and the sign of the source function are not a concern. When calculating things such as the scattering cross-section or the reflection coefficient, one always normalizes by the incident field. Therefore we do not need to specify explicitly the value of Δ_t/ϵ in (3.28)—it suffices to merely treat this coefficient as being contained in the source function itself.

A program that implements an additive source and takes snapshots of the electric field is shown in Program 3.4. The changes from Program 3.1 are shown in bold. The source function is exactly

the same as before except now, instead of setting the value of `ez[0]` to the value of this function, the source function is added to `ez[50]`. The source is introduced in line 29 and the update equations are unchanged from before. (Note that in this chapter the programs will be somewhat verbose, simplistic, and repetitive. Once we are comfortable with the FDTD algorithm we will pay more attention to better coding practices.)

Program 3.4 `1Dadditive.c`: One-dimensional FDTD program with an additive source.

```
1  /* 1D FDTD simulation with an additive source. */
2
3  #include <stdio.h>
4  #include <math.h>
5
6  #define SIZE 200
7
8  int main()
9  {
10   double ez[SIZE] = {0.}, hy[SIZE] = {0.}, imp0 = 377.0;
11   int qTime, maxTime = 200, mm;
12
13   char basename[80] = "sim", filename[100];
14   int frame = 0;
15   FILE *snapshot;
16
17   /* do time stepping */
18   for (qTime = 0; qTime < maxTime; qTime++) {
19
20     /* update magnetic field */
21     for (mm = 0; mm < SIZE - 1; mm++)
22       hy[mm] = hy[mm] + (ez[mm + 1] - ez[mm]) / imp0;
23
24     /* update electric field */
25     for (mm = 1; mm < SIZE; mm++)
26       ez[mm] = ez[mm] + (hy[mm] - hy[mm - 1]) * imp0;
27
28     /* use additive source at node 50 */
29     ez[50] += exp(-(qTime - 30.) * (qTime - 30.) / 100.);
30
31     /* write snapshot if time a multiple of 10 */
32     if (qTime % 10 == 0) {
33       sprintf(filename, "%s.%d", basename, frame++);
34       snapshot=fopen(filename, "w");
35       for (mm = 0; mm < SIZE; mm++)
36         fprintf(snapshot, "%g\n", ez[mm]);
37       fclose(snapshot);
38     }
```

```
39    } /* end of time-stepping */
40
41    return 0;
42  }
```

Snapshots of E_z taken at time-steps 20, 30, and 40 are shown in Fig. 3.8. Note that the field originates from node 50 and that it propagates to either side of this node. Also notice that the peak amplitude is half of what it was when the source function was implemented as a hardwired source.

As something of an aside, in Program 3.4 note that the code that takes a snapshot of the electric field was placed in the time-stepping but after the update equation. Thus one might ask: do the contents of snapshot file `sim.0` contain the fields at time zero or at time one? And, do the other snapshots correspond to times that multiples of 10 or do the correspond to one plus a multiple of 10? In nearly all practical cases it won't matter. The precise location of $t = 0$ is rather aribtrary. So, when looking at the snapshots it is usually sufficient to know that the sequence of snapshots start "at the beginning of the simulation" and then are taken every 10 time steps. However, if one wants to be more precise about this, absolute time is usually dictated by the source function. Now, think in terms of the hard-source implementation rather than the additive source. We have implemented a Gaussian source that has a peak amplitude at time-step 30. The way the code is written here, with the source being applied after the update equation and then the snapshot being taken last, we would see the peak at the source node in frame `sim.1`. In other words the snapshots do indeed correspond to times that are multiples of 10. So, in some sense the electric fields start at a time step of -1. The very first update update loop takes them up to time step 0, and then the source function is applied to set the field at the source node at time-step 0. *However*, this is truly a minor point and we will not worry about it in subsequent discussions. Whether the code that introduces the source appears before or after the update loop and whether the code that generates output appears before or after the update loop, often doesn't matter—the important thing is generally just that these things are included in the time-stepping loop.

3.9 Terminating the Grid

In most instances one is interested in modeling a problem which exists in an open domain, i.e., an infinite space. This is true even when the specific region of interest, say the region where a scatterer is present, may be small. That scatterer is in an unbounded space. Thus far the code we have written is only suitable for modeling a resonator since the nodes at the ends of the grid reflect any field incident upon them. We now wish to rectify this shortcoming. Absorbing boundary conditions (ABC's) will be used so that the grid, which will contain only a finite number of nodes, can behave as if it were infinite. In one dimension, when operating at the Courant limit of one, an exact ABC can be realized. Unfortunately in higher dimensions, or even in one dimension when not operating at the Courant limit, ABC's are only approximate. The better the ABC, the less energy it reflects back into the interior of the grid.

Before implementing an ABC, let us again consider the code shown in Program 3.4 but with the maximum number of time steps set to 450. With the FDTD method, the more ways in which

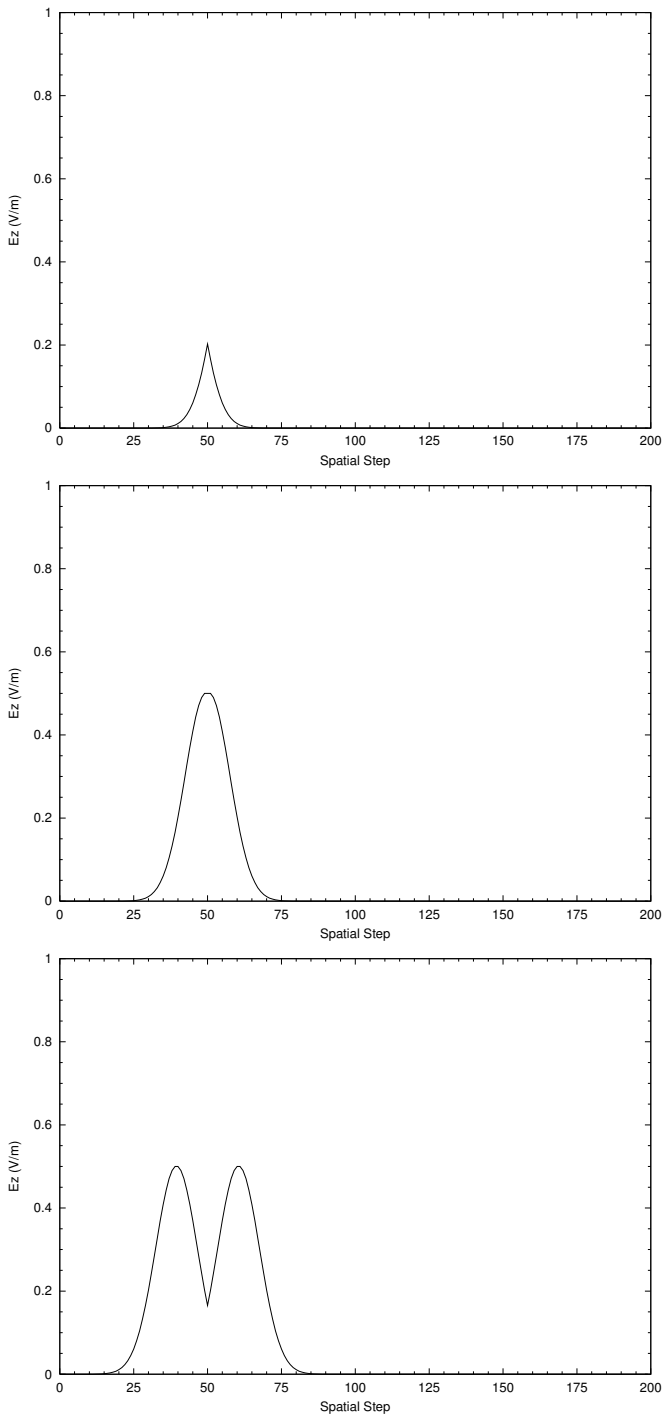


Figure 3.8: Snapshots taken at time-steps 20, 30, and 40 of the E_z field generated by Program 3.4. An additive source is applied to node 50 and the field is seen to propagate away from it to either side.

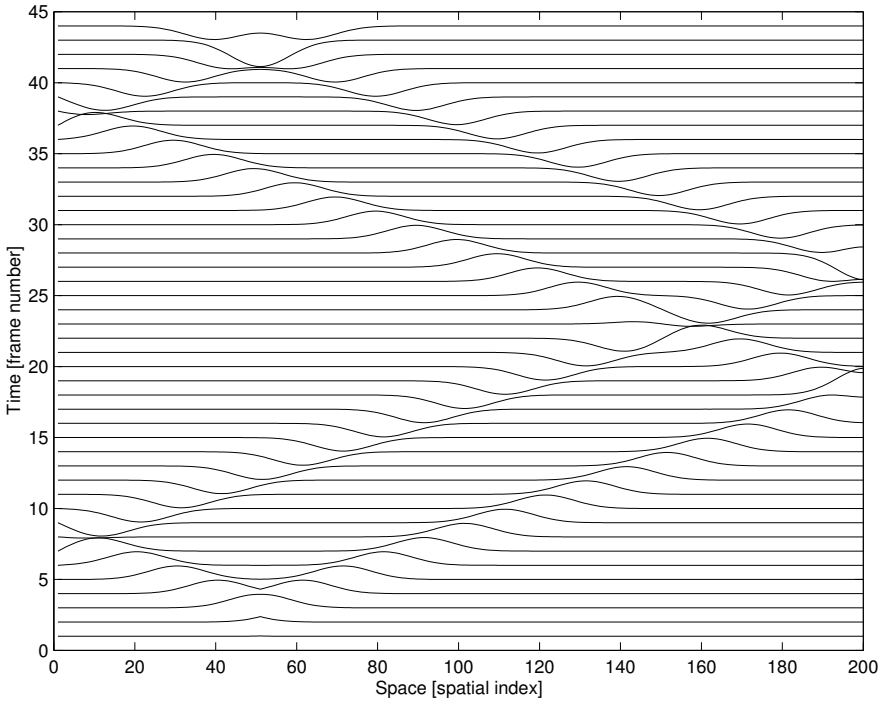


Figure 3.9: Waterfall plot of the electric field produced by Program 3.4. The computational domain has 200 nodes with a PEC boundary on the left and a PMC boundary on the right. The vertical axis gives the frame number. Snapshots, i.e., frames, were recorded every 10 time steps.

the field can be visualized, the better. Watching the field propagate in the time-domain can provide insights into the behavior of a system. Additionally, visualization of the propagation of the fields can be an invaluable aid when debugging FDTD code. Animations of the field are especially useful and different display strategies will be discussed later.

Since we cannot include an animation here, we will use a "waterfall plot" of the electric field in the one-dimensional domain. A waterfall plot is a collection of standard "x vs. y" plots where each plot is offset slightly from the next (a direct vertical offset will be used here). This can be thought of as stacking all the frames of an animation, one above the next.

Figure 3.9 shows the waterfall plot corresponding to the output from Program 3.4 (with a `maxTime` of 450). Each line represents a snapshot of the field throughout the computational domain. One can see that electric field starts to propagate away from the source which is at node 50. The curve/line corresponding to 5 on the vertical axis is the data from the sixth frame (i.e., `sim.5`). Since the frames are recorded every ten time-steps, since `sim.0` corresponds to the field at time zero, this line shows the field at the fiftieth time-step. This line has two peaks. One is traveling to the left and the other to the right. Once the left-going field encounters the end of the grid at node zero, it is both reflected and inverted. It then travels to the right as time progresses. The peak which originally travels to the right from the source encounters the right end of the grid around frame (or curve) 17. In this case, with the PMC boundary that exists there, the electric field is not inverted—instead, the magnetic field, which is not plotted, is inverted. A reflected wave then propagates back to the left. The field propagates back and forth, inverting its sign at the left

boundary and preserving its sign at the right boundary, until the simulation is halted. The Matlab code that was used to generate this waterfall plot is given in Appendix B. Additionally, Appendix B provides Matlab code that can be used to animate snapshots of a one-dimensional domain.

Returning to the issue of grid termination, when the Courant number is unity, the distance the wave travels in one temporal step is equal to one spatial step, i.e., $c\Delta_t = \Delta_x$. We are interested in modeling an open domain where there is no energy entering the grid "from the outside." Therefore, for node `ez[0]`, its updated value should just be the previous value that existed at `ez[1]`. Since no energy is entering the grid from the left, the field at `ez[1]` must be propagating solely to the left. At the next time step the value that was at `ez[1]` should now appear at `ez[0]`. Similar arguments hold at the other end of the grid. The updated value of `hy[199]` should be the previous value of `hy[198]`.

Thus, a simple ABC can be realized by adding the following line to Program 3.4 between lines 23 and 24

```
ez[0] = ez[1];
```

Similarly, the following line would be added between lines 19 and 20

```
hy[SIZE-1] = hy[SIZE-2];
```

The waterfall plot which is obtained for the electric field after making these changes is shown in Fig. 3.10. Note that the reflected fields are no longer present. The left- and right-going pulses reach the end of the grid and then disappear as if they have continued to propagate off to infinity. (However, there is still some persistent field that lingers throughout the grid. This field is small—about five orders of magnitude smaller than the peak when using single precision—and is a consequence of finite precision. These small fields are not visible on the scale of the plot and are not of much practical concern since typically other sources of error will be far larger.)

As mentioned previously, this simple ABC only works in limited situations. However, the basic premise is employed in many of the more complicated ABC's: the future value of the field at the end of the grid depends on some combination of the past and interior fields. We will return to this topic in Chap. 6.

3.10 Total-Field/Scattered-Field Boundary

Note that *any* function $f(\xi)$ which is twice differentiable is a solution to the wave equation. In one dimension all that is required is that the argument ξ be replaced by $t \pm x/c$. A proof was given in Sec. 2.16. Thus far the excitation of the FDTD grids has occurred at a point—either the hardwired source at the left end of the grid, as shown in Program 3.1, or the additive source at node 50, as shown in Program 3.4. Now our goal is to construct a source such that the excitation only propagates in one direction, i.e., the source introduces an incident field that is propagating to the right (the positive x direction). We will accomplish this using what is know as a total-field/scattered-field (TFSF) boundary.

We start by specifying the incident field as a function of space and time. A Gaussian pulse has been used for the excitation in the previous examples. A Gaussian can still be used to specify the

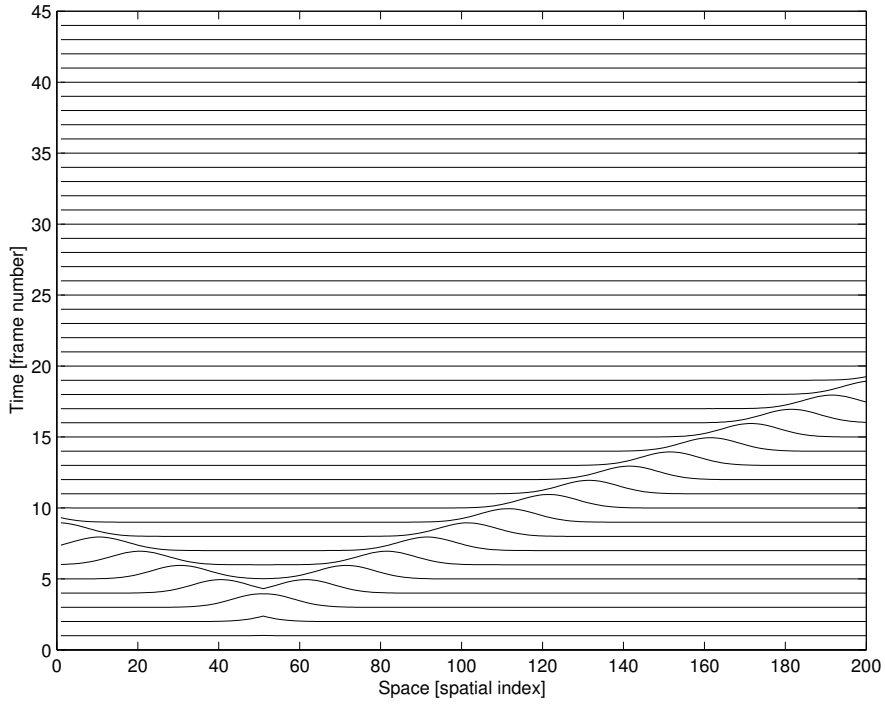


Figure 3.10: Waterfall plot of the electric field using the same computational domain as Fig. 3.9 except a simple ABC has been used to terminate the grid. Note that the field propagates from the additive source at node 50 and merely disappears when it reaches either end of the grid.

excitation, but to obtain a wave propagating to the right, the argument should be $t - x/c$ instead of merely t. Previously the source was given by

$$f(t) = f(q\Delta_t) = e^{-\left(\frac{q\Delta_t - 30\Delta_t}{10\Delta_t}\right)^2} = e^{-\left(\frac{q-30}{10}\right)^2} = f[q] \tag{3.29}$$

where $30\Delta_t$ is a delay and the term in the denominator of the exponent ($10\Delta_t$) controls the width of the pulse. Note that the time-step width Δ_t can be canceled from the numerator and denominator of the exponent.

For the propagating incident field, t in (3.29) is replaced with $t - x/c$. In discretized space-time this argument is given by

$$t - \frac{x}{c} = q\Delta_t - \frac{m\Delta_x}{c} = \left(q - \frac{m\Delta_x}{c\Delta_t}\right)\Delta_t = (q-m)\,\Delta_t \tag{3.30}$$

where the assumption that the Courant number $c\Delta_t/\Delta_x$ is unity has been used to write the last equality. This expression can now be used for the argument in the previous source function to obtain a propagating wave which we will identify as E_z^{inc}

$$E_z^{\text{inc}}[m,q] = e^{-\left(\frac{(q-m)\Delta_t - 30\Delta_t}{10\Delta_t}\right)^2} = e^{-\left(\frac{(q-m)-30}{10}\right)^2} \tag{3.31}$$

This equation essentially assumes that the origin, i.e., the point $x = 0$, corresponds to the index $\mathtt{m} = 0$. However, the origin can be shifted to a different point and this fact will be exploited later. Keep in mind that there is nothing that dictates that we must always think of the origin as corresponding to the left-most point in the grid.

The corresponding magnetic field is obtained by dividing the electric field by the characteristic impedance. Additionally, to ensure that $\mathbf{E}_z^{\text{inc}} \times \mathbf{H}_y^{\text{inc}}$ points in the desired direction of travel, the magnetic field must be negative, i.e.,

$$H_y^{\text{inc}}[m,q] = -\sqrt{\frac{\epsilon}{\mu}}E_z^{\text{inc}}[m,q] = -\frac{1}{\eta}e^{-\left(\frac{(q-m)-30}{10}\right)^2} \tag{3.32}$$

where $\eta = \sqrt{\mu/\epsilon}$ is the characteristic impedance of the medium. Note that the arguments do not need to be integers. If one needs to calculate the magnetic field at the position $m - 1/2$ and time $q - 1/2$, these are perfectly legitimate arguments.

In the total-field/scattered-field (TFSF) formulation, the computational domain is divided into two regions: (1) the total-field region which contains the incident field plus any scattered field and (2) the scattered-field region which contains only scattered field. The incident field is introduced on an fictitious seam, or boundary, between the total-field and the scattered-field regions. The location of this boundary is somewhat arbitrary, but it is typically placed so that any scatterers are contained in the total-field region.

When updating the fields, the update equations must be consistent. This is to say only scattered fields should be used to update a node in the scattered-field region and only total fields should be used to update a node in the total-field region. Figure 3.11 shows a one-dimensional grid where the TFSF boundary is assumed to exist between nodes $H_y\left[49 + \frac{1}{2}\right]$ and $E_z[50]$ (in Fig. 3.11 the nodes are shown in the computer-array form with integer indices). The node $H_y\left[49 + \frac{1}{2}\right]$ is equivalent to $H_y\left[50 - \frac{1}{2}\right]$ and will be written using the latter form in the following discussion. Note that

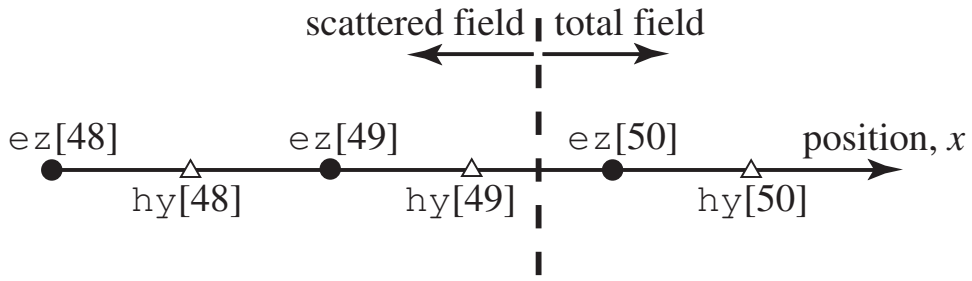


Figure 3.11: Portion of the one-dimensional arrays in the vicinity of a total-field/scattered-field boundary. Scattered field exists to the left of the boundary and total field exists to the right. Note that node `hy[49]` has an index of 49 in a computer program but corresponds logically to the location $(49+\frac{1}{2})\Delta_x$ or, equivalently, $(50-\frac{1}{2})\Delta_x$.

no matter where the boundary is placed, there will only be two nodes adjacent to the boundary—one electric-field node and one magnetic-field node. Furthermore, although the location of this boundary is arbitrary, once its location is selected, it is fixed throughout the simulation. Defining the scattered-field region to be to the left of the boundary and the total-field region to be to the right, we see that `hy[49]` is the last node in the scattered-field region while `ez[50]` is the first node in the total-field region.

When updating the nodes adjacent to the boundary, there is a problem, i.e., an inconsistency, in that a neighbor to one side is not the same type of field as the field being updated. This is to say that a total-field node will depend on a scattered-field node and, conversely, a scattered-field node will depend on a total-field node. The solution to this problem actually provides the way in which fields are introduced into the grid using the TFSF boundary.

Consider the usual update equation of the electric field at location $m = 50$ which was given in (3.18) and is repeated below

$$\overbrace{E_z^{q+1}[50]}^{\text{tot}} = \overbrace{E_z^{q}[50]}^{\text{tot}} + \frac{\Delta_t}{\epsilon\Delta_x}\left(\overbrace{H_y^{q+\frac{1}{2}}\left[50+\frac{1}{2}\right]}^{\text{tot}} - \overbrace{H_y^{q+\frac{1}{2}}\left[50-\frac{1}{2}\right]}^{\text{scat}}\right). \tag{3.33}$$

We have assumed the TFSF boundary is between $E_z^{q+1}[50]$ and $H_y^{q+\frac{1}{2}}\left[50-\frac{1}{2}\right]$ and the labels above the individual components indicate if the field is in the total-field region or the scattered-field region. We see that $E_z^{q+1}[50]$ and $H_y^{q+\frac{1}{2}}\left[50+\frac{1}{2}\right]$ are total-field nodes but $H_y^{q+\frac{1}{2}}\left[50-\frac{1}{2}\right]$ is a scattered-field node—it lacks the incident field. This can be fixed by adding the incident field to $H_y^{q+\frac{1}{2}}\left[50-\frac{1}{2}\right]$ in (3.33). This added field must correspond to the magnetic field which exists at location $50-1/2$ and time step $q+1/2$. Thus, a consistent update equation for $E_z^{q+1}[50]$ which

only involves total fields is

$$\overbrace{E_z^{q+1}[50]}^{\text{tot}} = \overbrace{E_z^{q}[50]}^{\text{tot}} + \frac{\Delta_t}{\epsilon\Delta_x}\left(\overbrace{H_y^{q+\frac{1}{2}}\left[50+\frac{1}{2}\right]}^{\text{tot}} - \overbrace{\left\{\overbrace{H_y^{q+\frac{1}{2}}\left[50-\frac{1}{2}\right]}^{\text{scat}} + \overbrace{\left(-\frac{1}{\eta}E_z^{\text{inc}}\left[50-\frac{1}{2},q+\frac{1}{2}\right]\right)}^{\text{inc}}\right\}}^{\text{tot}}\right). \tag{3.34}$$

The sum of the terms in braces gives the total magnetic field for $H_y^{q+\frac{1}{2}}\left[50-\frac{1}{2}\right]$. Note that here the incident field is assumed to be given. (It might be calculated analytically or, as we will see in higher dimensions where the TFSF boundary involves several points, it might be calculated with an auxilliary FDTD simulation of its own. But, either way, it is known.)

Instead of modifying the update equation, it is usually best to preserve the standard update equation (so that it can be put in a loop that pertains to all nodes), and then apply a correction in a separate step. In this way, $E_z^{q+1}[50]$ is updated in a two-step process:

$$E_z^{q+1}[50] = E_z^{q}[50] + \frac{\Delta_t}{\epsilon\Delta_x}\left(H_y^{q+\frac{1}{2}}\left[50+\frac{1}{2}\right] - H_y^{q+\frac{1}{2}}\left[50-\frac{1}{2}\right]\right), \tag{3.35}$$

$$E_z^{q+1}[50] = E_z^{q+1}[50] + \frac{\Delta_t}{\epsilon\Delta_x}\frac{1}{\eta}E_z^{\text{inc}}\left[50-\frac{1}{2},q+\frac{1}{2}\right]. \tag{3.36}$$

The characteristic impedance η can be written as $\sqrt{\mu_r\mu_0/\epsilon_r\epsilon_0} = \eta_0\sqrt{\mu_r/\epsilon_r}$. Recall from (3.19) that the coefficient $\Delta_t/\epsilon\Delta_x$ can be expressed as $\eta_0 S_c/\epsilon_r$ where S_c is the Courant number. Combining these terms, the correction equation (3.36) can be written

$$E_z^{q+1}[50] = E_z^{q+1}[50] + \frac{S_c}{\sqrt{\epsilon_r\mu_r}}E_z^{\text{inc}}\left[50-\frac{1}{2},q+\frac{1}{2}\right]. \tag{3.37}$$

With a Courant number of unity and free space (where $\epsilon_r = \mu_r = 1$), this reduces to

$$E_z^{q+1}[50] = E_z^{q+1}[50] + E_z^{\text{inc}}\left[50-\frac{1}{2},q+\frac{1}{2}\right]. \tag{3.38}$$

This equation simply says that the incident field that existed one-half a temporal step in the past and one-half a spatial step to the left of $E_z^{q+1}[50]$ is added to this node. This is logical since a field traveling to the right requires one-half of a temporal step to travel half a spatial step.

Now consider the update equation for $H_y^{q+\frac{1}{2}}\left[50-\frac{1}{2}\right]$ which is given by (3.15) (with one subtracted from the spatial offset):

$$\overbrace{H_y^{q+\frac{1}{2}}\left[50-\frac{1}{2}\right]}^{\text{scat}} = \overbrace{H_y^{q-\frac{1}{2}}\left[50-\frac{1}{2}\right]}^{\text{scat}} + \frac{\Delta_t}{\mu\Delta_x}\left(\overbrace{E_z^{q}[50]}^{\text{tot}} - \overbrace{E_z^{q}[49]}^{\text{scat}}\right). \tag{3.39}$$

As was true for the update of the electric field adjacent to the TFSF boundary, this is not a consistent equation since the terms are scattered-field quantities except for $E_z^q[50]$ which is in the total-field region. To correct this, the incident field could be subtracted from $E_z^q[50]$. Rather than modifying (3.39), we choose to give the necessary correction as a separate equation. The correction would be

$$H_y^{q+\frac{1}{2}}\left[50-\frac{1}{2}\right] = H_y^{q+\frac{1}{2}}\left[50-\frac{1}{2}\right] - \frac{\Delta_t}{\mu\Delta_x}E_z^{\text{inc}}[50,q]. \tag{3.40}$$

With a Courant number of unity and free space, this equation becomes

$$H_y^{q+\frac{1}{2}}\left[50-\frac{1}{2}\right] = H_y^{q+\frac{1}{2}}\left[50-\frac{1}{2}\right] - \frac{1}{\eta_0}E_z^{\text{inc}}[50,q]. \tag{3.41}$$

As mentioned previously, there is nothing that requires the origin to be assigned to one particular node in the grid. There is no reason that one has to associate the location $x = 0$ with the left end of the grid. In the TFSF formulation it is usually most convenient to fix the origin relative to the TFSF boundary itself. Let the origin $x = 0$ correspond to the node $E_z[50]$. Such a shift requires that 50 be subtracted from the spatial indices given previously for the incident field. The correction equations thus become

$$H_y^{q+\frac{1}{2}}\left[50-\frac{1}{2}\right] = H_y^{q+\frac{1}{2}}\left[50-\frac{1}{2}\right] - \frac{1}{\eta_0}E_z^{\text{inc}}[0,q], \tag{3.42}$$

$$E_z^{q+1}[50] = E_z^{q+1}[50] + E_z^{\text{inc}}\left[-\frac{1}{2},q+\frac{1}{2}\right]. \tag{3.43}$$

To implement a TFSF boundary, one merely has to translate (3.42) and (3.43) into the necessary statements. A program that implements a TFSF boundary between `hy[49]` and `ez[50]` is shown in Program 3.5.

Program 3.5 `1Dtfsf.c`: One-dimensional simulation with a TFSF boundary between `hy[49]` and `ez[50]`.

```
/* 1D FDTD simulation with a simple absorbing boundary condition
 * and a TFSF boundary between hy[49] and ez[50]. */

#include <stdio.h>
#include <math.h>

#define SIZE 200

int main()
{
  double ez[SIZE] = {0.}, hy[SIZE] = {0.}, imp0 = 377.0;
  int qTime, maxTime = 450, mm;

  char basename[80]="sim", filename[100];
```

```
15   int frame = 0;
16   FILE *snapshot;
17
18   /* do time stepping */
19   for (qTime = 0; qTime < maxTime; qTime++) {
20
21     /* simple ABC for hy[size - 1] */
22     hy[SIZE - 1] = hy[SIZE - 2];
23
24     /* update magnetic field */
25     for (mm = 0; mm < SIZE - 1; mm++)
26       hy[mm] = hy[mm] + (ez[mm + 1] - ez[mm]) / imp0;
27
28     /* correction for Hy adjacent to TFSF boundary */
29     hy[49] -= exp(-(qTime - 30.) * (qTime - 30.) / 100.) / imp0;
30
31     /* simple ABC for ez[0] */
32     ez[0] = ez[1];
33
34     /* update electric field */
35     for (mm = 1; mm<SIZE; mm++)
36       ez[mm] = ez[mm] + (hy[mm] - hy[mm - 1]) * imp0;
37
38     /* correction for Ez adjacent to TFSF boundary */
39     ez[50] += exp(-(qTime + 0.5 - (-0.5) - 30.) *
40                   (qTime + 0.5 - (-0.5) - 30.) / 100.);
41
42     /* write snapshot if time a multiple of 10 */
43     if (qTime % 10 == 0) {
44       sprintf(filename, "%s.%d", basename, frame++);
45       snapshot = fopen(filename, "w");
46       for (mm = 0; mm < SIZE; mm++)
47         fprintf(snapshot, "%g\n", ez[mm]);
48       fclose(snapshot);
49     }
50   } /* end of time-stepping */
51
52   return 0;
53 }
```

Note that this is similar to Program 3.4. Other than the incorporation of the ABC's in line 22 and 32, the only differences are the removal of the additive source (line 29 of Program 3.4) and the addition of the two correction equations in lines 29 and 39. The added code is shown in bold. In line 39, the half-step forward in time is obtained with `qTime+0.5`. The half-step back in space is obtained with the `-0.5` which is enclosed in parentheses.

The waterfall plot of the fields generated by Program 3.5 is shown in Fig. 3.12. Note that the

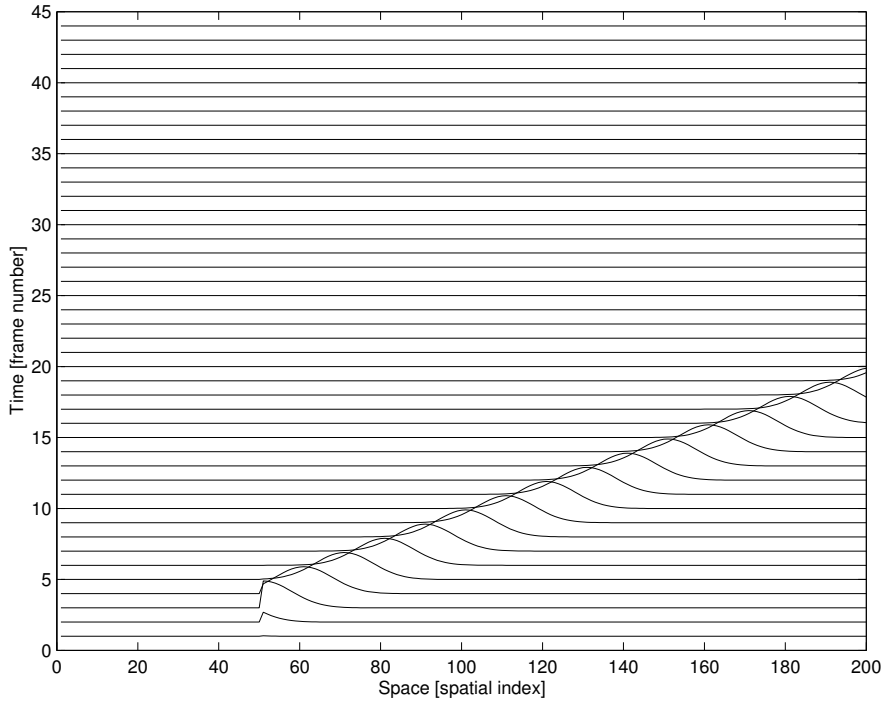


Figure 3.12: Waterfall plot of the electric fields produced by Program 3.5 which has a TFSF boundary between nodes `hy[49]` and `ez[50]`.

field appears at node 50 and travels exclusively to the right—no field propagates to the left from the TFSF boundary. Since there is nothing to scatter the incident field in this simulation, Fig. 3.12 shows only the incident field. The next step is, thus, the inclusion of some inhomogeneity in the grid to generate scattered fields.

3.11 Inhomogeneities

The FDTD update equations were obtained from approximations of Faraday's and Ampere's laws which were themselves differential equations. Differential equations pertain at a point. Thus, the ϵ and μ which appear in these equations are the ones which exist at the location of the corresponding node. It is certainly permissible that these values change from point to point. In fact, it is required that they change when modeling inhomogeneous material.

Recall, from (3.19), that the electric-field update equation had a coefficient of $\eta_0 S_c/\epsilon_r$. Assuming that the Courant number S_c is still unity, but allowing the relative permittivity to be a function of position, the update equation could be implemented with a statement such as

```
ez[m] = ez[m] + (hy[m] - hy[m-1])*imp0/epsR[m];
```

where the array element `epsR[m]` contains the relative permittivity at the point $m\Delta_x$, i.e., at a point collocated with the node `ez[m]`. The size of the `epsR` array would be the same size as the electric-field array and the elements have to be initialized to appropriate values.

The same concept applies to the relative permeability in the updating of the magnetic fields where the update coefficient is given by $S_c/\mu_r\eta_0$ (ref. (3.20)). The relative permeability that exists at the point in space corresponding to the location of a particular magnetic-field node is the one that should be used in the update equation for that node. Assuming an array `muR` has been created and initialized with the values of the relative permeability, the magnetic-fields would be updated with an equation such as

```
hy[m] = hy[m] + (ez[m + 1] - ez[m]) / imp0 / muR[m];
```

A program that models a region of space near the interface between free space and a dielectric with a relative permittivity of nine is shown in Program 3.6 (the permeability is that of free space). The incident field is still introduced via a TFSF boundary, which is in the free-space side of the computational domain, and the ABC on the left hand side is the same as before. However, there are some other minor changes between this program and the program in Program 3.5. The electric and magnetic fields are no longer initialized when they are declared. Instead, two loops are used to set the initial fields to zero. The magnetic field is now declared to have one fewer node than the electric field. This was done so that the computational domain begins and ends on an electric-field node. (There are no truly compelling reasons to have the computational domain begin and end with the same field type, but such symmetry can simplify coding and some aspects of certain problems.) Because the grid now terminates on an electric field, the ABC at the right end of the grid must be applied to this terminal electric-field node. This is accomplished with the statement in line 45.

Program 3.6 `1Ddielectric.c`: One-dimensional FDTD program to model an interface between free-space and a dielectric that has a relative permittivity ϵ_r of 9.

```
1  /* 1D FDTD simulation with a simple absorbing boundary
2   * condition, a TFSF boundary between hy[49] and ez[50], and
3   * a dielectric material starting at ez[100] */
4
5  #include <stdio.h>
6  #include <math.h>
7
8  #define SIZE 200
9
10 int main()
11 {
12   double ez[SIZE], hy[SIZE - 1], epsR[SIZE], imp0 = 377.0;
13   int qTime, maxTime = 450, mm;
14   char basename[80] = "sim", filename[100];
15   int frame = 0;
16   FILE *snapshot;
17
18   /* initialize electric field */
19   for (mm = 0; mm < SIZE; mm++)
20     ez[mm] = 0.0;
21
```

```
22    /* initialize magnetic field */
23    for (mm = 0; mm < SIZE - 1; mm++)
24      hy[mm] = 0.0;
25
26    /* set relative permittivity */
27    for (mm = 0; mm < SIZE; mm++)
28      if (mm < 100)
29        epsR[mm] = 1.0;
30      else
31        epsR[mm] = 9.0;
32
33    /* do time stepping */
34    for (qTime = 0; qTime < maxTime; qTime++) {
35
36      /* update magnetic field */
37      for (mm = 0; mm<SIZE - 1; mm++)
38        hy[mm] = hy[mm] + (ez[mm + 1] - ez[mm]) / imp0;
39
40      /* correction for Hy adjacent to TFSF boundary */
41      hy[49] -= exp(-(qTime - 30.) * (qTime - 30.) / 100.) / imp0;
42
43      /* simple ABC for ez[0] and ez[SIZE - 1] */
44      ez[0] = ez[1];
45      ez[SIZE-1] = ez[SIZE-2];
46
47      /* update electric field */
48      for (mm = 1; mm < SIZE - 1; mm++)
49        ez[mm] = ez[mm] + (hy[mm] - hy[mm - 1]) * imp0 / epsR[mm];
50
51      /* correction for Ez adjacent to TFSF boundary */
52      ez[50] += exp(-(qTime + 0.5 - (-0.5) - 30.)*
53                    (qTime + 0.5 - (-0.5) - 30.) / 100.);
54
55      /* write snapshot if time a multiple of 10 */
56      if (qTime % 10 == 0) {
57        sprintf(filename, "%s.%d", basename, frame++);
58        snapshot = fopen(filename, "w");
59        for (mm = 0; mm < SIZE; mm++)
60          fprintf(snapshot, "%g\n", ez[mm]);
61        fclose(snapshot);
62      }
63    } /* end of time-stepping */
64
65    return 0;
66  }
```

The relative-permittivity array `epsR` is initialize in the loop starting at line 27. If the spatial

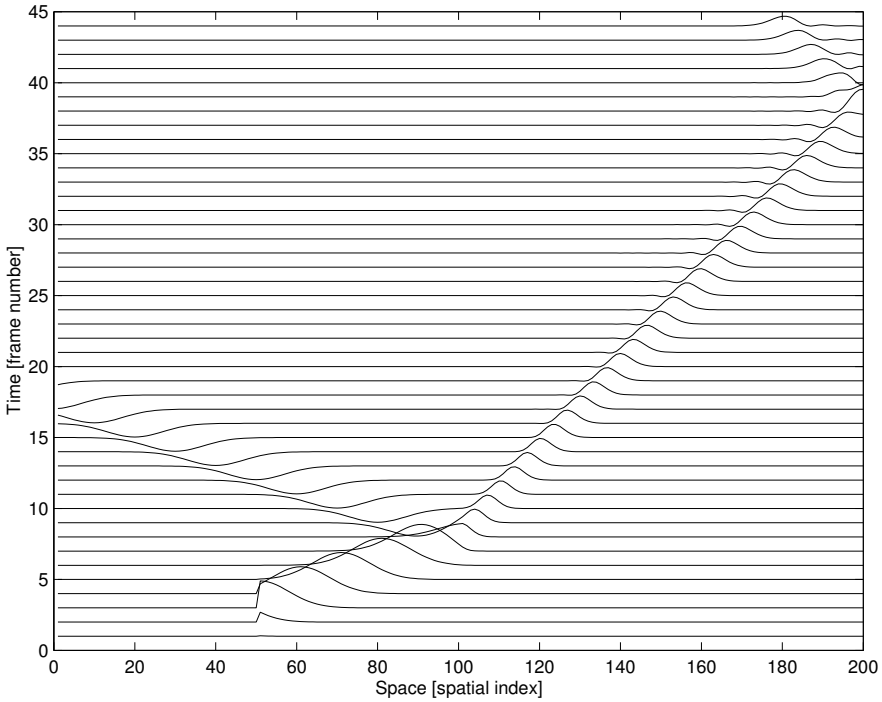


Figure 3.13: Waterfall plot of the electric fields produced by Program 3.6 which has a dielectric with a relative permittivity of 9 starting at node 100. Free space is to the left of that.

index $\mathtt{mm}$ is less than 100, the relative permittivity is set to unity (i.e., free space), otherwise it is set to 9. The characteristic impedance of free space is η_0 while the impedance for the dielectric is $\eta_0/3$. Note that the update equations do not directly incorporate the dielectric impedance. Rather, the coefficient that appears in the equation uses the impedance of free space and the relative permittivity that pertains at that point.

When a wave is normally incident from a medium with a characteristic impedance η_1 to a medium with a characteristic impedance η_2, the reflection coefficient Γ and the transmission coefficient T are given by

$$\Gamma = \frac{\eta_2 - \eta_1}{\eta_2 + \eta_1}, \tag{3.44}$$

$$T = \frac{2\eta_2}{\eta_2 + \eta_1}. \tag{3.45}$$

Therefore the reflection and transmission coefficients that pertain to this example are

$$\Gamma = \frac{\eta_0/3 - \eta_0}{\eta_0/3 + \eta_0} = -\frac{1}{2}, \tag{3.46}$$

$$T = \frac{2\eta_0/3}{\eta_0/3 + \eta_0} = \frac{1}{2}. \tag{3.47}$$

The waterfall plot of the data produced by Program 3.6 is shown in Fig. 3.13. Once the field encounters the interface at node 100, a reflected field (i.e., a scattered field) is created. Although

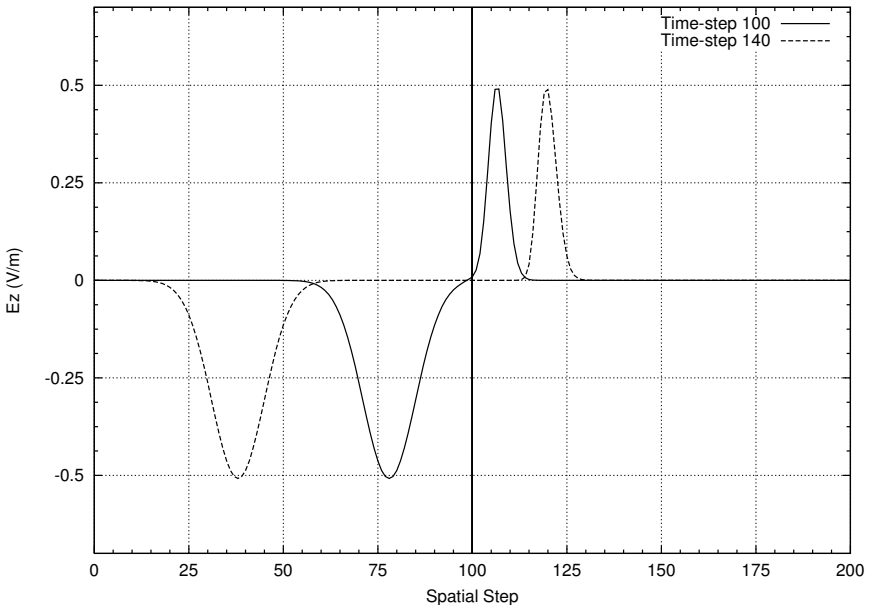


Figure 3.14: Two of the snapshots produced by Program 3.6. The vertical line at node 100 corresponds to the interface between free space and the dielectric. The incident pulse had unit amplitude. Shown in this figure are the transmitted field (to the right of the interface) and the reflected field (to the left).

one cannot easily judge scales from the waterfall plot, it can be seen that the reflected field is negative and appears to have about half the magnitude of the incident pulse (the peak of the incident field spans a vertical space corresponding to nearly two frames while the peak of the reflected field spans about one frame). Similarly, the transmitted pulse is positive and appears to have half the magnitude of the incident field. One can see this more clearly in Fig. 3.14 which shows the field at time-steps 100 and 140. The incident pulse had unit amplitude. At the time-steps shown here, the field has split into the transmitted and reflected pulses, each of which has an magnitude of one-half.

Returning to the waterfall plot of Fig. 3.13, one can also see that the pulse in the dielectric travels more slowly than the pulse in free space. With a relative permittivity of 9, the speed of light should be one-third of that in free space. Thus, from frame to frame the peak in the dielectric has moved one-third of the distance that the peak moves in free space.

There are two numerical artifacts present in Fig. 3.13, one which we need to fix and the other we need to understand. Note that when the reflected field encounters the left boundary it disappears. The ABC does its job and the field is absorbed. On the other hand, when the transmitted wave encounters the right boundary, at approximately frame 37, it is not completely absorbed. A reflected wave is produced which is visible in the upper right-hand corner of Fig. 3.13. Why is the ABC no longer working? The problem is that the simple ABC used so far is based on the assumption that the wave travels one spatial step for every time step. In this dielectric, with a relative permittivity of 9, the speed of light is one-third that of free space and hence the wave does not travel one spatial step per time step—it travels a third of a spatial step. A possible fix might be to

update the electric field on the boundary with the value of the neighboring electric-field node from three time steps in the past. However, what if the relative permittivity of the dielectric were 2? In that case the speed of light would be $1/\sqrt{2}$ times that of free space. There would be no past value of an interior node that could be used directly to update the boundary node. So, it is necessary to rethink the implementation of the ABC so that it can handle these sorts of situations. This will be addressed in another chapter.

The other artifact present in Fig. 3.13 is slightly less obvious. If you look at the trailing edge of the transmitted pulse around frame 33, or so, you will see a slight wiggle. The incident field is a Gaussian pulse which asymptotically goes to zero to either side of the peak value. However, the transmitted pulse does not behave this way—at least not after propagating in the dielectric for a while (initially there are no wiggles visible at the trailing edge of the transmitted pulse). These wiggles are caused by *dispersion* in the FDTD grid. When the Courant number is anything other than unity, the FDTD grid is dispersive, meaning that different frequencies propagate at different speeds. Note that we have defined the Courant number as the $c\Delta_t/\Delta_x$ where c is the speed of light in free space. We will generally maintain the convention that c represents the speed of light in free space. However, one can think of the Courant number as a local quantity that equals the local speed of light multiplied by the ratio Δ_t/Δ_x. Because the speed of light is one-third of that of free space, the local Courant number in the dielectric is not unity. Since the Gaussian pulse consists of a range of frequencies, and these frequencies are propagating at different speeds, the pulse "breaks apart" as it propagates. In practice, one tries to ensure that the amount of dispersion is small, but it is unavoidable in multi-dimensional FDTD analysis. Dispersion will be considered further later.

Because of the discretized nature of the FDTD grid, the location of a material boundary can be somewhat ambiguous. The relatively permittivity that pertains to a particular electric-field node can be assumed to exist over the space that extends from one of its neighboring magnetic-field nodes to the other neighboring magnetic-field node. This idea is illustrated in Fig. 3.15 which shows a portion of the FDTD grid together with the permittivity associated with each node. The permittivities are indicated with the bar along the bottom of the figure.

If there is only a change in permittivity, the location of the interface between the different media seems rather clear. It coincides with the magnetic-field node that has ϵ_1 to one side and ϵ_2 to the other. However, what if there is a change in permeability too? The permeabilities are indicated with a bar along the top of the figure. It is seen that the interface associated with the change in permeabilities is not aligned with the interface associated with the change in permittivities. One way to address this problem is to assume the true interface is aligned with an electric-field node. This node would then use the average of the permittivities of the media to either side. This scenario is depicted in Fig. 3.16. Alternatively, if one wants to have the boundary aligned with a magnetic-field node, then the node located on the boundary would use the average of the permeabilities to either side while the electric-field nodes would use the permittivity of the first medium if they were to the left of the boundary and use the permittivity of the second medium if they were to the right.

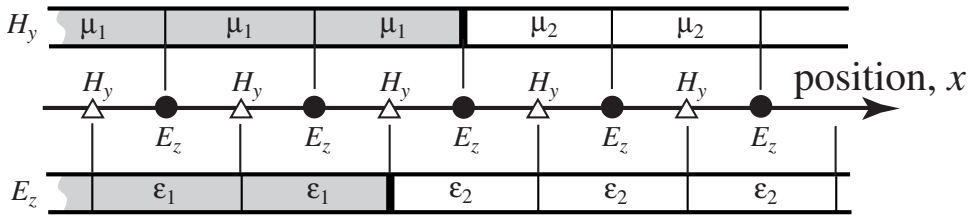


Figure 3.15: One-dimensional grid depicting an abrupt change in both the permittivity and permeability. The actual location of the interface between the two media is ambiguous.

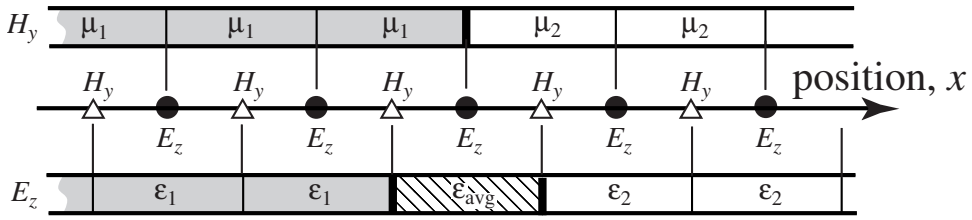


Figure 3.16: One-dimensional grid depicting a change from one medium to another. An electric-field node is assumed to be collocated with the interface, hence the permittivity used there is the average of the permittivities to either side.

3.12 Lossy Material

When a material has a finite conductivity σ, a conduction-current term is added to Ampere's law (which is distinct from the source-current term mentioned in Sec. 3.8). Thus,

$$\sigma \mathbf{E} + \epsilon \frac{\partial \mathbf{E}}{\partial t} = \nabla \times \mathbf{H}. \tag{3.48}$$

The discretized form of Ampere's law provided the update equation for the electric field. As before, assuming only a z component of the field and variation only in the x direction, this equation reduces to

$$\sigma E_z + \epsilon \frac{\partial E_z}{\partial t} = \frac{\partial H_y}{\partial x}. \tag{3.49}$$

As discussed in Sec. 3.3 and detailed in Fig. 3.2, this equation was expanded about the point $(m\Delta_x, (q+1/2)\Delta_t)$ to obtain the electric-field update equation. However, when loss is present, the undifferentiated electric field appears on the left side of the equation. With the assumed arrangement of the nodes shown in Fig. 3.2, there is no electric field at the space-time point $(m\Delta_x, (q+1/2)\Delta_t)$. This problem can be circumvented by using the averaging (in time) of the electric field to either side of the desired point, i.e.,

$$E_z^{q+\frac{1}{2}}[m] \approx \frac{E_z^{q+1}[m] + E_z^q[m]}{2}. \tag{3.50}$$

Thus a suitable discretization of Ampere's law when loss is present is

$$\sigma \frac{E_z^{q+1}[m] + E_z^q[m]}{2} + \epsilon \frac{E_z^{q+1}[m] - E_z^q[m]}{\Delta_t} = \frac{H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}\right] - H_y^{q+\frac{1}{2}}\left[m-\frac{1}{2}\right]}{\Delta_x}. \tag{3.51}$$

As before, this can be solved for $E_z^{q+1}[m]$, which is the "future" field, in terms of purely past fields. The result is

$$E_z^{q+1}[m] = \frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}} E_z^q[m] + \frac{\frac{\Delta_t}{\epsilon\Delta_x}}{1+\frac{\sigma\Delta_t}{2\epsilon}} \left(H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}\right] - H_y^{q+\frac{1}{2}}\left[m-\frac{1}{2}\right] \right). \tag{3.52}$$

When σ is zero this reduces to the previous update equation (3.18).

In previous update equations it was possible to express the coefficients in terms of the Courant number, i.e., the ratio of the temporal step to the spatial step. In (3.52) it appears the term $\sigma\Delta_t/2\epsilon$ requires that the temporal step be specified (together with the conductivity and permittivity). However, there is a way to express this such that the temporal step does not need to be stated explicitly. This will be considered in detail in Sec. 5.7.

As we will see, it is occasionally helpful to incorporate a magnetic conduction current in Faraday's law. Similar to the electric conduction current, the magnetic conduction current is assumed to be the product of the magnetic conductivity σ_m and the magnetic field. Faraday's law becomes

$$-\sigma_m \mathbf{H} - \mu \frac{\partial \mathbf{H}}{\partial t} = \nabla \times \mathbf{E}. \tag{3.53}$$

We again restrict consideration to an H_y component which varies only in the x direction. This reduces to

$$\sigma_m H_y + \mu \frac{\partial H_y}{\partial t} = \frac{\partial E_z}{\partial x}. \tag{3.54}$$

As before, this is expanded/discretized at the space-time point $((m+1/2)\Delta_x, q\Delta_t)$. Since there is no magnetic field available at integer time steps, the magnetic field is averaged in time to get an approximation of the field at time $q\Delta_t$. This yields

$$\sigma_m \frac{H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}\right] + H_y^{q-\frac{1}{2}}\left[m+\frac{1}{2}\right]}{2} + \mu \frac{H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}\right] - H_y^{q-\frac{1}{2}}\left[m+\frac{1}{2}\right]}{\Delta_t} = \frac{E_z^q[m+1] - E_z^q[m]}{\Delta_x}. \tag{3.55}$$

Solving for $H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}\right]$ yields the update equation

$$H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}\right] = \frac{1-\frac{\sigma_m\Delta_t}{2\mu}}{1+\frac{\sigma_m\Delta_t}{2\mu}} H_y^{q-\frac{1}{2}}\left[m+\frac{1}{2}\right] + \frac{\frac{\Delta_t}{\mu\Delta_x}}{1+\frac{\sigma_m\Delta_t}{2\mu}} \left(E_z^q[m+1] - E_z^q[m]\right). \tag{3.56}$$

When σ_m is zero this reduces to (3.15).

Program 3.7 models a lossy dielectric half-space that starts at node 100. As before, the relative permittivity is 9. However there is also an electric loss present such that $\sigma\Delta_t/2\epsilon$ is 0.01. The program uses two coefficient arrays, `ceze` and `cezh`. The terms in the `ceze` array multiply the previous (or "self") term while the `cezh` array contains the terms that multiply the spatial difference of the magnetic fields. Think of these arrays as consisting of coefficients (or constants), hence the "`c`" at the start of their name, that appear in the E_z update equation, hence the `ez` part of the name, and multiplying either the electric field (`ceze`) or the magnetic field (`cezh`). The values of these arrays are set in the loop that starts at line 27. Since the simple ABC previously

employed at the right edge of the grid does not work, it has been removed but the left side of the grid is terminated as before. The magnetic field update is unchanged from before. A waterfall plot of the data produced by Program 3.7 is shown in Fig. 3.17. The pulse decays as it propagates in the lossy region and eventually decays to a rather negligible value. Thus the lack of an ABC at the right side of the grid is not really a concern in this particular instance.

Program 3.7 `1Dlossy.c`: One-dimensional simulation with a lossy dielectric region.

```
1  /* 1D FDTD simulation of a lossy dielectric region. */
2
3  #include <stdio.h>
4  #include <math.h>
5
6  #define SIZE 200
7  #define LOSS 0.01
8
9  int main()
10 {
11   double ez[SIZE], hy[SIZE - 1], ceze[SIZE], cezh[SIZE],
12     imp0 = 377.0;
13   int qTime, maxTime = 450, mm;
14   char basename[80] = "sim", filename[100];
15   int frame = 0;
16   FILE *snapshot;
17
18   /* initialize electric field */
19   for (mm = 0; mm < SIZE; mm++)
20     ez[mm] = 0.0;
21
22   /* initialize magnetic field */
23   for (mm = 0; mm < SIZE - 1; mm++)
24     hy[mm] = 0.0;
25
26   /* set electric-field update coefficients */
27   for (mm = 0; mm < SIZE; mm++)
28     if (mm < 100) { /* free space */
29       ceze[mm] = 1.0;
30       cezh[mm] = imp0;
31     } else {     /* lossy dielectric */
32       ceze[mm] = (1.0 - LOSS) / (1.0 + LOSS);
33       cezh[mm] = imp0 / 9.0 / (1.0 + LOSS);
34     }
35
36   /* do time stepping */
37   for (qTime = 0; qTime < maxTime; qTime++) {
38
```

```
39      /* update magnetic field */
40      for (mm = 0; mm < SIZE - 1; mm++)
41        hy[mm] = hy[mm] + (ez[mm + 1] - ez[mm]) / imp0;
42
43      /* correction for Hy adjacent to TFSF boundary */
44      hy[49] -= exp(-(qTime - 30.) * (qTime - 30.) / 100.) / imp0;
45
46      /* simple ABC for ez[0] */
47      ez[0] = ez[1];
48
49      /* update electric field */
50      for (mm = 1; mm < SIZE - 1; mm++)
51        ez[mm] = ceze[mm] * ez[mm] + cezh[mm] * (hy[mm] - hy[mm - 1]);
52
53      /* correction for Ez adjacent to TFSF boundary */
54      ez[50] += exp(-(qTime + 0.5 - (-0.5) - 30.) *
55                    (qTime + 0.5 - (-0.5) - 30.) / 100.);
56
57      /* write snapshot if time a multiple of 10 */
58      if (qTime % 10 == 0) {
59        sprintf(filename, "%s.%d", basename, frame++);
60        snapshot=fopen(filename, "w");
61        for (mm = 0; mm < SIZE; mm++)
62          fprintf(snapshot, "%g\n", ez[mm]);
63        fclose(snapshot);
64      }
65    } /* end of time-stepping */
66
67    return 0;
68  }
```

When loss is present the characteristic impedance of the medium becomes

$$\eta = \sqrt{\frac{\mu\left(1 - j\frac{\sigma_m}{\omega\mu}\right)}{\epsilon\left(1 - j\frac{\sigma}{\omega\epsilon}\right)}} = \eta_0\sqrt{\frac{\mu_r\left(1 - j\frac{\sigma_m}{\omega\mu}\right)}{\epsilon_r\left(1 - j\frac{\sigma}{\omega\epsilon}\right)}} \tag{3.57}$$

When $\sigma_m/\mu = \sigma/\epsilon$ the terms in parentheses are equal and hence cancel. With those terms canceled, the characteristic impedance is indistinguishable from the lossless case. Therefore

$$\eta\big|_{\frac{\sigma_m}{\mu}=\frac{\sigma}{\epsilon}} = \eta\big|_{\sigma_m=\sigma=0} = \eta_0\sqrt{\frac{\mu_r}{\epsilon_r}}. \tag{3.58}$$

As shown in (3.44), the reflection coefficient for a wave normally incident on a planar boundary is proportional to the difference of the impedances to either side of the interface. If the material on one side is lossless while the material on the other side is lossy with $\sigma_m/\mu = \sigma/\epsilon$, then the impedances are matched provided the ratios of ϵ_r and μ_r are also matched across the boundary.

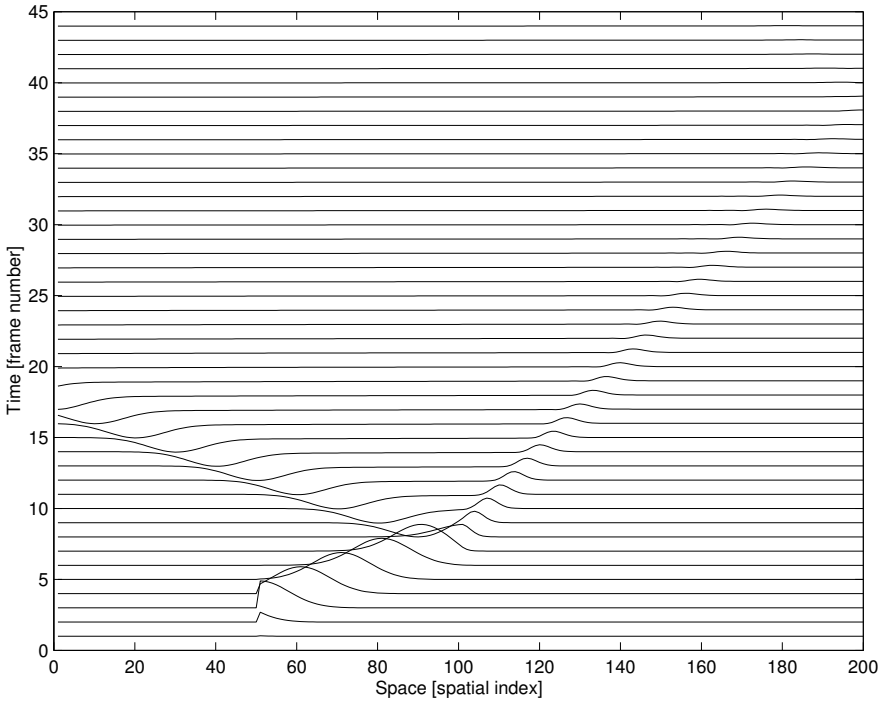


Figure 3.17: Waterfall plot of the electric fields produced by Program 3.7 which has a lossy dielectric region with a relative permittivity of 9 starting at node 100.

With the impedances matched, there will be no reflection from the interface. Therefore a lossy layer could be used to terminate the grid. The fields will dissipate in this lossy region and, if the region is large enough, may be small by the time they encounter the end of the grid. Upon reflection from the end of the grid, the fields would have to propagate back through the lossy layer where they would decay even further. With proper design the reflected fields can be made inconsequentially small when they eventually get back to the lossless portion of the grid.

A lossy layer with the impedance matched to the previous region can be implemented easily in one dimension. Program 3.8 shows a program where a lossless dielectric layer with $\epsilon_r = 9$ starts at node 100. The lossless region extends to node 180. At node 180, and beyond, the material has both a nonzero electric and a magnetic conductivity. The conductivities are matched in the sense that $\sigma_m/\mu = \sigma/\epsilon$. Thus the terms $\sigma_m \Delta_t/2\mu$ and $\sigma \Delta_t/2\epsilon$ in the update-equations are also matched. In this program these terms are set to 0.02. The coefficients used in the magnetic-field update equations are stored in the arrays `chyh` and `chye`.

The waterfall plot of the data generated by Program 3.8 is shown in Fig. 3.18. The fields that enter the lossless region propagate to the right and eventually encounter the lossy region. Because the impedances of the lossless and lossy media are matched, the fields enter the lossy region without reflection (actually, that is true in the continuous world, but only approximately true in the discretized FDTD world—there is some small reflection present). As the fields propagate in the lossy region they dissipate to the point where they are almost negligible when they reenter the lossless region. There is no reflected field evident in the upper right corner of the plot.

Program 3.8 `1Dmatched.c`: Program with a lossless dielectric region followed by a lossy layer that has its impedance matched to the lossless dielectric.

```
/* 1D FDTD simulation of a lossless dielectric region
 * followed by a lossy layer which matches the impedance
 * of the dielectric.  */

#include <stdio.h>
#include <math.h>

#define SIZE 200
#define LOSS 0.02
#define LOSS_LAYER 180

int main()
{
  double ez[SIZE], hy[SIZE - 1], ceze[SIZE], cezh[SIZE],
    chyh[SIZE - 1], chye[SIZE - 1], imp0 = 377.0;
  int qTime, maxTime = 450, mm;
  char basename[80] = "sim", filename[100];
  int frame = 0;
  FILE *snapshot;

  /* initialize electric field */
  for (mm = 0; mm < SIZE; mm++)
    ez[mm] = 0.0;

  /* initialize magnetic field */
  for (mm = 0; mm < SIZE - 1; mm++)
    hy[mm] = 0.0;

  /* set electric-field update coefficients */
  for (mm = 0; mm < SIZE; mm++)
    if (mm < 100) {
      ceze[mm] = 1.0;
      cezh[mm] = imp0;
    } else if (mm < LOSS_LAYER) {
      ceze[mm] = 1.0;
      cezh[mm] = imp0 / 9.0;
    } else {
      ceze[mm] = (1.0 - LOSS) / (1.0 + LOSS);
      cezh[mm] = imp0 / 9.0 / (1.0 + LOSS);
    }

  /* set magnetic-field update coefficients */
  for (mm = 0; mm < SIZE - 1; mm++)
    if (mm < LOSS_LAYER) {
```

```
45        chyh[mm] = 1.0;
46        chye[mm] = 1.0 / imp0;
47      } else {
48        chyh[mm] = (1.0 - LOSS) / (1.0 + LOSS);
49        chye[mm] = 1.0 / imp0 / (1.0 + LOSS);
50      }
51
52    /* do time stepping */
53    for (qTime = 0; qTime < maxTime; qTime++) {
54
55      /* update magnetic field */
56      for (mm = 0; mm < SIZE - 1; mm++)
57        hy[mm] = chyh[mm] * hy[mm] +
58                    chye[mm] * (ez[mm + 1] - ez[mm]);
59
60      /* correction for Hy adjacent to TFSF boundary */
61      hy[49] -= exp(-(qTime - 30.) * (qTime - 30.) / 100.) / imp0;
62
63      /* simple ABC for ez[0] */
64      ez[0] = ez[1];
65
66      /* update electric field */
67      for (mm = 1; mm < SIZE - 1; mm++)
68        ez[mm] = ceze[mm] * ez[mm] +
69                    cezh[mm] * (hy[mm] - hy[mm - 1]);
70
71      /* correction for Ez adjacent to TFSF boundary */
72      ez[50] += exp(-(qTime + 0.5 - (-0.5) - 30.)*
73                     (qTime + 0.5 - (-0.5) - 30.) / 100.);
74
75      /* write snapshot if time a multiple of 10 */
76      if (qTime % 10 == 0) {
77        sprintf(filename, "%s.%d", basename, frame++);
78        snapshot=fopen(filename, "w");
79        for (mm = 0; mm < SIZE; mm++)
80          fprintf(snapshot, "%g\n", ez[mm]);
81        fclose(snapshot);
82      }
83    } /* end of time-stepping */
84
85    return 0;
86  }
```

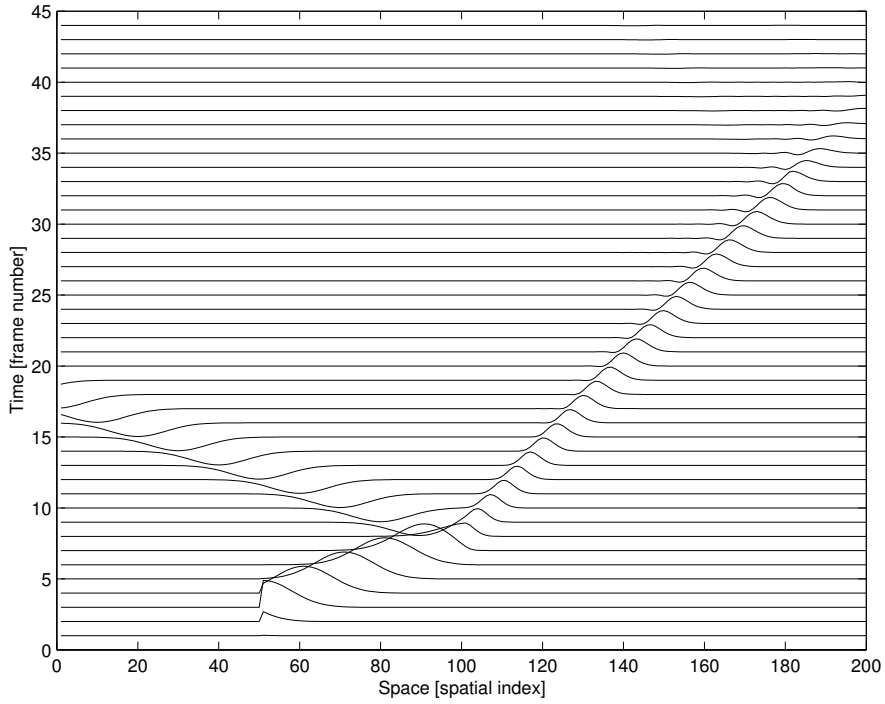


Figure 3.18: Waterfall plot of the electric fields produced by Program 3.8 which has a dielectric region starting at node 100 with a relative permittivity of 9. This lossless region is followed by a lossy layer with matched impedance. The lossy region starts at node 180.

Chapter 4

Improving the FDTD Code

4.1 Introduction

The C code presented in the previous chapter adequately served its purpose—it implemented the desired FDTD functionality in a relatively straightforward way. However, as the algorithms get more involved, for instance, when implementing simulations of two- or three-dimensional problems, the readability, portability, and maintainability will be improved if our implementation is slightly more sophisticated. In this chapter we will implement the algorithms of the previous chapter in a way which may seem, at first, to be overly complicated. However, as the complexity of the algorithms increases in coming chapters, we will see that the effort required to understand this more sophisticated approach will have been worth it.

4.2 Arrays and Dynamic Memory Allocation

In C an array is stored in a block of contiguous memory. Memory itself is fundamentally a one-dimensional quantity since memory is ultimately accessed with a single memory address. Let us consider a one-dimensional array of doubles `ez` where the size is specified at run-time rather than at compile-time. The compiler needs to know about the existence of this array, but at compile-time it does not need to know the amount or the location of memory where the array will be stored. We can specify the size of the array when the program is run and we can allow the computer to decide the location of the necessary amount of memory. The compiler is told about the potential existence of the array with a pointer. Once the pointer is associated with the appropriate block of memory, it is virtually indistinguishable from a one-dimensional array.

The code shown in Fragment 4.1 demonstrates how the array `ez` can be created at run-time. In line 1 `ez` is declared as a pointer—it can store an address but when the program initially starts it does not point to anything meaningful. Line 2 declares two integer variables: `num_elements` which will contain the number of elements in the array, and `mm` which will be used as a loop counter. Lines 4 and 5 determine the number of elements the user desires.

Lecture notes by John Schneider. `fdtd-improved-code.tex`

Fragment 4.1 Fragment of code demonstrating how the size of the array `ez` can be set at runtime. The header file `stdlib.h` would typically have to be included to provide the prototype for the `calloc()` function.

```
1   double *ez;
2   int num_elements, mm;
3
4   printf("Enter the size of the array: ");
5   scanf("%d", &num_elements);
6
7   ez = calloc(num_elements, sizeof(double));
8
9   for (mm=0; mm < num_elements; mm++)
10    ez[mm] = 3.0 * mm;
```

Line 7 is the key to getting `ez` to behave as an array. In this line the pointer `ez` is set equal to the memory address that is returned by the function `calloc()`.* `calloc()` takes two arguments. The first specifies the number of elements in the array while the second specifies the amount of memory needed for a single element. Here the `sizeof()` operator is used to obtain the size of a double variable. (Although not shown in this fragment, when using the `calloc()` function one typically has to include the header file `stdlib.h` to provide the function prototype.) If `calloc()` is unable to provide the requested memory it will return `NULL`. †

After the call of `calloc()` in line 7, `ez` points to the start of a contiguous block of memory where the array elements can be stored. To demonstrate this, lines 9 and 10 write the value of three times the array index to each element of the array (so that `ez[0]` would be 0.0, `ez[1]` would be 3.0, `ez[2]` would be 6.0, and so on).

Some compilers will actually complain about the code as it is written in Fragment 4.1. The "problem" is that technically `calloc()` returns a void pointer—it is simply the address of the start of a block of memory, but we have not said what is stored at that memory. We want a pointer to doubles since we will be storing double precision variables in this memory. The compiler really already knows this since we are storing the address in `ez` which is a pointer to doubles. Nevertheless, some compilers will give a warning because of line 7. Therefore, to ensure that compilers do not complain, it would be best to replace line 7 with

```
ez = (double *)calloc(num_elements, sizeof(double));
```

In this way the void pointer returned by `calloc()` is converted (or cast) to a pointer to doubles.

*`calloc()` is closely related to the function `malloc()` which also allocates a block of memory and returns the address of the start of that block. However `calloc()` returns memory which has been cleared, i.e., set to zero, while `malloc()` returns memory which may contain anything. Since we want the field arrays initially to be zero, it is better to use `calloc()` than `malloc()`.

†Robust code would check the return value and take appropriate measures if `NULL` were returned. For now we will assume that `calloc()` succeeded.

4.3 Macros

C provides a preprocessor which "processes" your code prior to the compiler itself. Preprocessor directives start with the pound sign (#) and instruct the preprocessor to do things such as include a header file (with an `#include` statement) or substitute one string for another. Program 3.1 had three preprocessor directives. Two were used to include the files `stdio.h` and `math.h` and one used a `#define` statement to tell the preprocessor to substitute `200` for all occurrences of the string `SIZE`.

Compilers allow you to see what the source code is after passing through the preprocessor. Using the GNU C compiler, one adds the `-E` flag to the compiler command to obtain the output from the preprocessor. So, for example, one can see the source code as it appears after passing through the preprocessor with the command

```
gcc -E 1DbareBones.c
```

In this case you will observe that there are many, many more lines of output than there are in your original program. This is because of the inclusion of the header files. Your original program will appear at the end of the output except now `SIZE` does not appear anywhere. Instead, any place it had appeared you will now see `200`.

The `#define` statement can also be used to create a macro. Unlike the simple string substitution used before, a macro can take one or more arguments. These arguments dictate how strings given as arguments should re-appear in the output. For example, consider the following macro

```
#define SQR(X)  ((X) * (X))
```

This tells the preprocessor that every time `SQR` appears in the source code, the preprocessor should take whatever appeared as an argument and replace that with the argument multiplied by itself. Here the argument `X` is just serving as a place holder. Consider the following code

```
a = 6.0 * SQR(3.0 + 4.0);
```

After passing through the preprocessor, this code would appear as

```
a = 6.0 * ((3.0 + 4.0) * (3.0 + 4.0));
```

The result would be that `a` would be set equal to $6 \times 7^2 = 294$.

It may seem that there are an excess number of parentheses in this macro, but one must be careful with macros to ensure the desired results are obtained. Consider this macro

```
#define BAD_SQR(X)  X * X
```

If a program then contained this statement

```
a = 6.0 * BAD_SQR(3.0 + 4.0);
```

the preprocessor translates this to

```
a = 6.0 * 3.0 + 4.0 * 3.0 + 4.0;
```

Since multiplication has a higher precedence than addition, in this case `a` would be set equal to $18 + 12 + 4 = 34$. There is no harm in using extra parentheses, so do not hesitate to use them to ensure you are getting precisely what you want.

It is also worth noting that the macro definition itself will typically not be terminated by a semicolon. If one includes a semicolon, it could easily produce unintended results. As an example, consider

```
#define BAD_SQR1(X)  ((X) * (X));
```

If a program then contained this statement

```
a = BAD_SQR1(4.0) + 10.0;
```

the preprocessor would translate this to

```
a = ((4.0) * (4.0)); + 10;
```

Note that the "`+ 10`" is consider a separate statement since there is a semicolon between it and the squared term.[‡] Thus, `a` will be set to 16.0.

Macros may have any number of arguments. As an example, consider

```
#define FOO(X, Y) (Y) * cos((X) * (Y))
```

Given this macro, the following code

```
a = FOO(cos(2.15), 7.0 * sqrt(4.0));
```

would be translated by the preprocessor to

```
a = (7.0 * sqrt(4.0)) * cos((cos(2.15)) * (7.0 * sqrt(4.0)))
```

Macros can span multiple lines provided the newline character at the end of each line is "quoted" with a backslash. Said another way, a backslash must be the last character on a line if the statement is to continue on the next line.

There is another "trick" that one can do with macros. One can say they want the string version of an argument in the preprocessor output, essentially the argument will appear enclosed in quotes in the output. To understand why this is useful, it first helps to recall that in C, if one puts two string next to each other, the compiler treats the two separate strings as a single string. Thus, the following commands are all equivalent

```
printf("Hello world.\n");
printf("Hello "  "world.\n");
printf("Hello "
       "world.\n");
```

Each of these produce the output

```
Hello world.
```

[‡]When using the GNU C compiler, this "bad" code will compile without error. If one adds the `-Wall` flag when compiling, the GNU compiler will provide a warning that gives the line number and a statement such as "`warning: statement with no effect`." Nevertheless, the code will compile.

Note that there is no comma between the separated strings in the `printf()` statements and the amount of whitespace between the strings is irrelevant.

If we want the preprocessor to produce the string version of an argument in the output, we affix `#` to the argument name in the place where it should appear in the output. For example, we could use a macro to print what is being calculated and show the result of the calculation as well:

```
#define SHOW_CALC(X) \
        printf(#X " = %g\n", X)
```

The first line of this macro is terminated with a backslash telling the preprocessor that the definition is continued on the next line. Now, if the code contained

```
SHOW_CALC(6.0 + 24.0 / 8.0);
```

the preprocessor would convert this to

```
printf("6.0 + 24.0 / 8.0" " = %g\n", 6.0 + 24.0 / 8.0);
```

When the program is run, the following would be written to the screen

```
6.0 + 24.0 / 8.0 = 9
```

We are now at a point where we can construct a fairly sophisticated macro to do memory allocation. The macro will check if the allocation of memory was successful. If not, the macro will report the problem and terminate the program. The following fragment shows the desired code.

Fragment 4.2 Macro for allocating memory for a one-dimensional array. The trailing backslashes must appear directly before the end of the line. (By "quoting" the newline character at the end of the line we are telling the preprocessor the definition is continued on the next line.)

```
1 #define ALLOC_1D(PNTR, NUM, TYPE)                                    \
2     PNTR = (TYPE *)calloc(NUM, sizeof(TYPE));                        \
3     if (!PNTR) {                                                     \
4       perror("ALLOC_1D");                                            \
5       fprintf(stderr,                                                \
6           "Allocation failed for " #PNTR ".  Terminating...\n");\
7       exit(-1);                                                      \
8     }
```

The macro `ALLOC_1D()` takes three arguments: a pointer (called `PNTR`), an integer specifying the number of elements (`NUM`), and a data type (`TYPE`). The macro uses `calloc()` to obtain the desired amount of memory. It then checks if the allocation was successful. Recall that `calloc()` will return `NULL` if it failed to allocate the requested memory. In C, the exclamation mark is also the "not operator." So, if the value of `PNTR` is `NULL` (or, thought of another way, "false"), then `!PNTR` would be true. If the memory allocation failed, two error messages are printed (one message is generated using the system function `perror()`, the other uses `fprintf()` and

writes the output to `stderr`, which is typically the screen). The program is then terminated with the `exit()` command.

With this macro in our code we could create an array `ez` with 200 elements with statements such as

```
double *ez;

ALLOC_1D(ez, 200, double);
```

Technically the semicolon in the second statement is not necessary (since this macro translates to a block of code that ends with a close-brace), but there is no harm in having it.

4.4 Structures

In C one can group data together, essentially create a compound data type, using what is known as a structure. A program can have different types of structures, i.e., ones that bundle together different kinds of data. For example, a program might use structures that bundle together a person's age, weight, and name as well as structures that bundle together a person's name, social security number, and income. Just as we can have multiple variables of a given type and each variable is a unique instance of that data type, we can have multiple variables that corresponds to a given structure, each of which is a unique instance of that structure.

Initially, when declaring a structure, we first tell the compiler the composition of the structure. As an example, the following command defines a `person` structure that contains three elements: a character pointer called `name` that will be used to store the name of a person, an integer called `age` that will correspond to the person's age, and an integer called `weight` that will correspond to the person's weight.

```
struct person {
  char *name;
  int age;
  int weight;
};
```

This statement is merely a definition—no structure has been created yet. To create a `person` structure called `bob` and another one called `sue`, a command such as the following could be used:

```
struct person bob, sue;
```

Actually, one could combine these two statements and create `bob` and `sue` with the following:

```
struct person {
  char *name;
  int age;
  int weight;
}  bob, sue;
```

However, in the code to come, we will use a variant of the two-statement version of creating structures. It should also be pointed out that elements do not need to be declared with individual statements of their own. For example, we could write

```
  int age, weight;
```

instead of the two separate lines shown above.

To access the elements of a structure we write the name of the structure, a dot, and then the name of the element. For example, `bob.age` would be the `age` element of the structure `bob`, and `sue.age` would be the `age` element of the structure `sue` (despite the fact that these are both `age` elements, they are completely independent variables).

Now, let's write a program that creates two `person` structures. The program has a function called `showPerson1()` to show the elements of a `person` and another function called `changePerson1()` that reduces the person's age by two and decreases the weight by five. Each of these functions takes a single argument, a `person` structure. The program is shown in Program 4.3.

Program 4.3 `struct-demo1.c`: Program to demonstrate the basic use of structures.

```
1  /* Program to demonstrate the use of structures.  Here structures are
2  passed as arguments to functions.*/
3
4  #include <stdio.h>
5
6  struct person {
7    char *name;
8    int age;
9    int weight;
10 };
11
12 void changePerson1(struct person p);
13 void showPerson1(struct person p);
14
15 int main() {
16   struct person bob, sue;
17
18   sue.name = "Sue";
19   sue.age = 21;
20   sue.weight = 120;
21
22   bob.name = "Bob";
23   bob.age = 62;
24   bob.weight = 180;
25
26   showPerson1(sue);
27
28   printf("*** Before changePerson1() ***\n");
29   showPerson1(bob);
30   changePerson1(bob);
31   printf("*** After changePerson1() ***\n");
```

```
32   showPerson1(bob);
33
34   return 0;
35 }
36
37 /* Function to display the elements in a person. */
38 void showPerson1(struct person p) {
39   printf("name: %s\n", p.name);
40   printf("age: %d\n", p.age);
41   printf("weight: %d\n", p.weight);
42   return;
43 }
44
45 /* Function to modify the elements in a person. */
46 void changePerson1(struct person p) {
47   p.age = p.age - 2;
48   p.weight = p.weight - 5;
49   printf("*** In changePerson1() ***\n");
50   showPerson1(p);
51   return;
52 }
```

In line 16 of the `main()` function we declare the two `person` structures `bob` and `sue`. This allocates the space for these structures, but as yet their elements do not have meaningful values. In 18 we set the `name` of element of `sue`. The following two lines set the `age` and `weight`. The elements for `bob` are set starting at line 22. In line 26 the `showPerson1()` function is called with an argument of `sue`. This function, which starts on line 38, shows the `name`, `age`, and `weight` of a `person`.

After showing the elements of `sue`, in line 29 of `main()`, `showPerson1()` is used to show the elements of `bob`. In line 30 the function `changePerson1()` is called with an argument of `bob`. This function, which is given starting on line 46, subtracts two from the `age` and five from the `weight`. After making these modifications, in line 50, `showPerson1()` is called to display the modified person.

Finally, returning to line 32 of the `main()` function, the `showPerson1()` function is called once again with an argument of `bob`. Note that this is after `bob` has supposedly been modified by the `changePerson1()` function. The output produced by Program 4.3 is

```
name: Sue
age: 21
weight: 120
*** Before changePerson1() ***
name: Bob
age: 62
weight: 180
*** In changePerson1() ***
name: Bob
```

```
age: 60
weight: 175
*** After changePerson1() ***
name: Bob
age: 62
weight: 180
```

Here we see that the initial values of `sue` and `bob` are as we would anticipate. Also, when `showPerson1()` is called from within `changePerson1()`, we see the modified values for `bob`, i.e., his age has been reduced by two and his weight has been reduced by five. *However,* the last three lines of output show that, insofar as the `bob` structure in the `main()` function is concerned, nothing has changed! `bob` has not been modified by `changePerson1()`.

Although this behavior may not have been what we would have anticipated, it is correct. When a structure is given as an argument, the function that is called is given a complete copy of the original structure. Thus, when that function modifies the elements of the structure, it is modifying a copy of the original—it is not affecting the original itself.

If we want a function that we call to be able to modify a structure, we must pass a pointer to that structure. We will modify the code to permit this but let use first introduce the `typedef` statement that allows to write slightly cleaner code. Having to write `struct person` everywhere we want to specify a `person` structure is slightly awkward. C allows us to use the `typedef` statement to define an equivalent. The following statement tells the compiler that `Person` is the equivalent of `struct person`

```
typedef struct person Person;
```

Note that there is no requirement that we use the same word for the structure and the `typedef`-equivalent. Additionally, even when using the same word, there is no need to use different capitalization (since the structure and its equivalent are maintained in different name spaces).

Now, let us assume we want to create two *pointers* to structures, one named `susan` and one `robert`. These can be created with

```
struct person *susan, *robert;
```

Assuming the `typedef` statement given above has already appeared in the program, we could instead write:

```
Person *susan, *robert;
```

`susan` and `robert` are pointers to structures but, initially, they do not actually point to any structures. We cannot set their elements because there is is no memory allocated for the storage of these elements. Thus, we must allocate memory for these structures and ensure the pointers point to the memory.

To accomplish this, we can include in our program a statement such as

```
ALLOC_1D(susan, 1, Person);
```

Recalling the `ALLOC_1D()` macro presented in Sec. 4.3, this statement will allocate the memory for one `Person` and associate that memory with the pointer `susan`. We can now set the element associated with `susan`. However, accessing the elements of a *pointer* to a `Person` is different than directly accessing the elements of a `Person`. To set the `age` element of `susan` we would have to write either

```
(*susan).age = 21;
```

or

```
susan->age = 21;
```

Program 4.4 is similar to Program 4.3 in many respects except here, rather than using structures directly, the program primarily deals with pointers to structures. As will be shown, this allows other functions to change the elements within any given structure—the function merely has to be passed a pointer to the structure rather than (a copy of) the structure.

Program 4.4 `struct-demo2.c`: Program to demonstrate the basic use of pointers to structures.

```
1  /* Program to demonstrate the use of pointers to structures. */
2
3  #include <stdlib.h>
4  #include <stdio.h>
5
6  #define ALLOC_1D(PNTR, NUM, TYPE)                                    \
7      PNTR = (TYPE *)calloc(NUM, sizeof(TYPE));                        \
8      if (!PNTR) {                                                     \
9        perror("ALLOC_1D");                                            \
10       fprintf(stderr,                                                \
11           "Allocation failed for " #PNTR ".  Terminating...\n");\
12       exit(-1);                                                      \
13     }
14
15 struct person {
16   char *name;
17   int age;
18   int weight;
19 };
20
21 typedef struct person Person;
22
23 void changePerson2(Person *p);
24 void showPerson2(Person *p);
25
26 int main() {
27   Person *robert, *susan;
28
29   ALLOC_1D(susan, 1, Person);
30   ALLOC_1D(robert, 1, Person);
31
32   susan->name = "Susan";
33   susan->age = 21;
```

```
34    susan->weight = 120;
35
36    robert->name = "Robert";
37    robert->age = 62;
38    robert->weight = 180;
39
40    showPerson2(susan);
41
42    printf("*** Before changePerson2() ***\n");
43    showPerson2(robert);
44    changePerson2(robert);
45    printf("*** After changePerson2() ***\n");
46    showPerson2(robert);
47
48    return 0;
49  }
50
51  /* Function to display the elements in a person. */
52  void showPerson2(Person *p) {
53    printf("name: %s\n", p->name);
54    printf("age: %d\n", p->age);
55    printf("weight: %d\n", p->weight);
56    return;
57  }
58
59  /* Function to modify the elements in a person. */
60  void changePerson2(Person *p) {
61    p->age = p->age - 2;
62    p->weight = p->weight - 5;
63    printf("*** In changePerson2() ***\n");
64    showPerson2(p);
65    return;
66  }
```

The `typedef` statement in line 21 allows us to write simply `Person` instead of `struct person`. This is followed by the prototypes for functions `showPerson2()` and `changePerson2()`. These functions are similar to the corresponding functions in the previous program except now the arguments are pointers to structures instead of structures. Thus, the syntactic changes are necessary in the functions themselves (e.g., we have to write `p->age` instead of `p.age`). Starting on line 29 the `ALLOC_1D()` macro is used to allocate the memory for the `susan` and `robert` pointers that were declared in line 27. The values of the elements are then set, the contents of the structures displayed, `robert` is modified, and the contents or `robert` are shown again.

The output produced by Program 4.4 is

```
name: Susan
age: 21
```

```
weight: 120
*** Before changePerson2() ***
name: Robert
age: 62
weight: 180
*** In changePerson2() ***
name: Robert
age: 60
weight: 175
*** After changePerson2() ***
name: Robert
age: 60
weight: 175
```

Note that in this case, the changes made by `changePerson2()` are persistent—when the `main()` function shows the elements of `robert` we see the modified values (unlike with the previous program).

4.5 Improvement Number One

Now let us use some of the features discussed in the previous sections in a revised version of the "bare-bones" program that appeared in Sec. 3.5. Here we will bundle much of the data associated with an FDTD simulation into a structure that we will call a `Grid` structure. (We will often refer to this structure as simply the `Grid` or a `Grid`.) Here will define the `Grid` as follows:

```
struct Grid {
  double *ez;         // electric field array
  double *hy;         // magnetic field array
  int sizeX;          // size of computational domain
  int time, maxTime; // current and max time step
  double cdtds;       // Courant number
};
```

In the "improved" program that we will soon see, there will not appear to be much reason for creating this structure. Why bother? The motivation will be more clear when we start to modularize the code so that different functions handle different aspects of the FDTD simulation. By bundling all the relevant information about the simulation into a structure, we can simply pass as an argument to each function a pointer to this `Grid`.

First, let us create a header file `fdtd1.h` in which we will put the definition of a `Grid` structure. In this file we will also include the (1D) macro for allocating memory. Finally, in a quest to keep the code as clean as possible, we will also define a few additional preprocessor directives: a macro for accessing the `ez` array elements, a macro for accessing the `hy` array elements, and some simple `#define` statements to facilitate references to `sizeX`, `time`, `maxTime`, and `cdtds`. The complete header file is shown in Program 4.5.

Program 4.5 `fdtd1.h`: Header file for the first "improved" version of a simple 1D FDTD program.

```
1  #ifndef _FDTD1_H
2  #define _FDTD1_H
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  struct Grid {
8    double *ez;
9    double *hy;
10   int sizeX;
11   int time, maxTime;
12   double cdtds;
13 };
14
15 typedef struct Grid Grid;
16
17 /* memory allocation macro */
18 #define ALLOC_1D(PNTR, NUM, TYPE)                                    \
19     PNTR = (TYPE *)calloc(NUM, sizeof(TYPE));                        \
20     if (!PNTR) {                                                     \
21       perror("ALLOC_1D");                                            \
22       fprintf(stderr,                                                \
23           "Allocation failed for " #PNTR ".  Terminating...\n");     \
24       exit(-1);                                                      \
25     }
26
27 /* macros for accessing arrays and such */
28 /* NOTE!!!!  Here we assume the Grid structure is g. */
29 #define Hy(MM)     g->hy[MM]
30 #define Ez(MM)     g->ez[MM]
31 #define SizeX      g->sizeX
32 #define Time       g->time
33 #define MaxTime    g->maxTime
34 #define Cdtds      g->cdtds
35
36 #endif  /* matches #ifndef _FDTD1_H */
```

Line 1 checks if this header file, i.e., `fdtd1.h`, has previously been included. Including multiple copies of a header file can cause errors or warnings (such as when a term that was previously defined in a `#define` statement is again mentioned in a different `#define` statement). In the simple code with which we are working with now, multiple inclusions are not a significant concern, but we want to develop the techniques to ensure multiple inclusions do not cause problems in the future. Line 1 checks for a previous inclusion by testing if the identifier (technically a compiler

directive) _FDTD1_H is not defined. If it is not defined, the next statement on line 2 defines it. Thus, _FDTD1_H serves as something of a flag. It will be defined if this file has been previously included and will not be defined otherwise. Therefore, owing to the #ifndef statement at the start of the file, if this header file has previously been included the preprocessor will essentially ignore the rest of the file. The #ifndef statement on line 1 is paired with the #endif statement on line 36 at the end of the file.

Assuming this file has not previously been included, next, the header files stdio.h and stdlib.h are included. Since the macro ALLOC_1D() uses both calloc() and some printing functions, both these headers have to be included anywhere ALLOC_1D() is used.

The commands for defining the Grid structure start on line 7. Following that, the ALLOC_1D() macro begins on line 18. The macros given on lines 29 and 30 allow us to access the field arrays in a way that is easier to read. (Importantly, note that these statements assume that the Grid in the program has been given the name g! We will, in fact, make a habit of using the variable name g for a Grid in the code to follow.) So, for example, the macro on line 30 allows us to write Ez(50) instead of writing the more cumbersome g->ez[50]. Note that the index for the array element is now enclosed in parentheses and not square brackets. The definitions in lines 31 through 34 allow us to write SizeX instead of g->sizeX, Time instead of g->time, MaxTime instead of g->maxTime, and Cdtds instead of g->cdtds.

An improved version of Program 3.1 is shown in Program 4.6.

Program 4.6 improved1.c: Source code for an improved version of the bare-bones 1D FDTD program.

```
1  /* Improved bare-bones 1D FDTD simulation. */
2
3  #include "fdtd1.h"
4  #include <math.h>
5
6  int main()
7  {
8    Grid *g;
9    double imp0 = 377.0;
10   int mm;
11
12   ALLOC_1D(g, 1, Grid);
13
14   SizeX = 200;     // size of grid
15   MaxTime = 250;   // duration of simulation
16   Cdtds = 1.0;     // Courant number (unused)
17
18   ALLOC_1D(g->ez, SizeX, double);
19   ALLOC_1D(g->hy, SizeX, double);
20
21   /* do time stepping */
22   for (Time = 0; Time < MaxTime; Time++) {
```

```
23
24      /* update magnetic field */
25      for (mm = 0; mm < SizeX - 1; mm++)
26        Hy(mm) = Hy(mm) + (Ez(mm + 1) - Ez(mm)) / imp0;
27
28      /* update electric field */
29      for (mm = 1; mm < SizeX; mm++)
30        Ez(mm) = Ez(mm) + (Hy(mm) - Hy(mm - 1)) * imp0;
31
32      /* hardwire a source node */
33      Ez(0) = exp(-(Time - 30.0) * (Time - 30.0) / 100.0);
34
35      printf("%g\n", Ez(50));
36    } /* end of time-stepping */
37
38    return 0;
39  }
```

In line 8 the pointer to the `Grid` structure `g` is declared. Because we have just declared a pointer to a `Grid`, we next need to obtain the memory to store the elements of the structure itself. This allocation is done in line 12.

Because the `Grid` contains pointers `ez` and `hy`, we do not need to declare those field arrays separately. The size of the computational domain is set in line 14 while the duration of the simulation is set in 15. After the preprocessor has processed the code, these lines will actually be

```
g->sizeX = 200;
g->maxTime = 250;
```

At this point in the program the pointers `ez` and `hy` do not have any memory associated with them—we cannot store anything yet in the field arrays. Therefore the next step is the allocation of memory for the field arrays which is accomplished in lines 18 and 19.

The rest of the code is largely the same as given in Program 3.1. The only difference being a slight change in notation/syntax. Program 3.1 and Program 4.6 produce identical results.

This new version of the bare-bones simulation is split into two files: the header file `fdtd1.h` and the source file `improved1.c`. This is depicted in Fig. 4.1. The `main()` function in file `improved1.c` handles all the calculations associated with the FDTD simulation. Since there is only a single source file, there is no obvious advantage to creating a header file. However, as we further modularize the code, the header file will provide a convenient way to ensure that different source files share common definitions of various things. For example, many source files may need to know about the details of a `Grid` structure. These details can be written once in the header file and then the header file can be included into different source files as appropriate. Examples of this will be provided in the following sections.

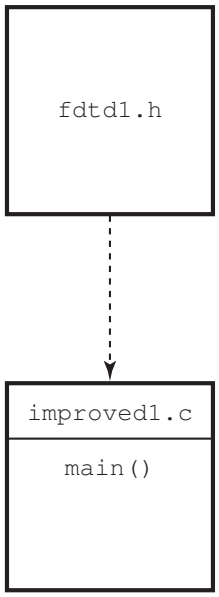


Figure 4.1: The files associated with the first improved version of the FDTD code. The header file `fdtd1.h` is included in the source file `improved1.c` as indicated by the dashed line. The source file `improved1.c` contains the `main()` function which performs all the executable statements associated with the simulation.

4.6 Modular Design and Initialization Functions

Thus far all the programs have been written using a single source file that contains a single function (the `main()` function), or in the case of the previous section, one source file and one header file. As the complexity of the FDTD simulation increases this approach becomes increasingly unwieldy and error prone. A better approach is to modularize the program so that different functions perform specific tasks—not everything is done within `main()`. For example, we may want to use one function to update the magnetic field, another to update the electric field, another to introduce energy into the grid, and another to handle the termination of the grid.

Additionally, with C it is possible to put different functions in different source files and compile the source files separately. By putting different functions in different source files, it is possible to have different functions that are within one source file share variables which are "hidden" from all functions that do not appear in that file. You can think of these variables as being private, known only to the functions contained in the file. As will be shown, such sharing of variables can be useful when one wants to initialize the behavior of a function that will be called multiple times (but the initialization only needs to be done once).

To illustrate how functions within a file can share variables that are otherwise hidden from other parts of a program, assume there is a function that we want to call several times. Further assume this function performs a calculation based on some parameters but these parameters only need to be set once—they will not vary from the value to which they are initially set. For this type of scenario, it is often convenient to split the function into two functions: one function handles the initialization of the parameters (the "initialization function") and the other function handles

the calculation based on those parameters (the "calculation function"). Now, the question is: How can the initialization function make the parameters visible to the calculation function and how can the values of these parameters persist between one invocation of the function and the next? The answer lies in global variables.

Generally the use of global variables is discouraged as they can make programs hard to understand and difficult to debug. However, global variables can be quite useful when used properly. To help minimize the problems associated with global variables, we will further modularizing the program so that the initialization function and calculation function mentioned above are stored in a separate file from the rest of the program. In this way the global variables that these functions share are not visible to any other function.

As a somewhat contrived example of this approach to setting parameters, assume we want to write a program that will calculate the values of a harmonic function where the user can specify the amplitude and the phase, i.e., we want to calculate $f(x) = a\cos(x + \phi)$ where a is the amplitude and ϕ is the phase. Note that we often just write $f(x)$ for a function like this even though the function depends on a and ϕ as well as x. We usually do *not* write $f(x, a, \phi)$ because we think of a and ϕ as fixed values (even though they have to be specified at some point) while x is the value we are interested in varying. Assume we want to write our program so that there is a function `harmonic1()` that is the equivalent of $f(x) = a\cos(x+\phi)$. `harmonic1()` should take a single argument that corresponds to the value of x. We will use a separate function, `harmonicInit1()` that will set the amplitude and phase.

A file that contains a suitable the `main()` function and the associated statements to realize the parameter setting discussed above is shown in Program 4.7. The function prototypes for `harmonicInit1()` and `harmonic1()` are given in lines 12 and 13, respectively. Note, however, that these functions do not appear in this file. Between lines 19 and 23 the user is prompted for the amplitude and phase (and the phase is converted from degrees to radians). These values are passed as arguments to the `harmonicInit1()` function. As we will see a little later, this function sets persistent global parameters to these values so that they are visible to the function `harmonic1()` whenever it is called. The for-loop that starts on line 28 generates the desired output values. Here we set the variable `x` to values between 0 and 2π. In line 30 the value of `x` is printed together with the value of `harmonic1(x)`. Note that the `harmonic1()` function is not passed the amplitude or phase as an argument.

Program 4.7 `param-demo1.c`: File containing the `main()` function and appropriate header material that is used to demonstrate the setting of persistent parameters in an auxilliary function. Here `harmonicInit1()` and `harmonic1()` are serving this auxilliary role. The code associated with those functions is in a separate file (see Program 4.8).

```
1 /* param-demo1.c:  Program that demonstrates the setting of
2  * "persistent" parameters via the arguments of an initialization
3  * function.  Here the parameters control the amplitude and phase of a
4  * harmonic function f(x) = a cos(x + phi).  This program generates
5  * num_points of the harmonic function with the "x" value of the
6  * varying between zero and 2*pi.
7  */
```

```
8
9  #include <stdio.h>
10 #include <math.h>  // To obtain M_PI, i.e., 3.14159...
11
12 void harmonicInit1(double amp, double phase);
13 double harmonic1(double x);
14
15 int main() {
16   double amp, phase, x;
17   int mm, num_points = 100;
18
19   printf("Enter the amplitude: ");
20   scanf(" %lf", &amp);
21   printf("Enter the phase [in degrees]: ");
22   scanf(" %lf", &phase);
23   phase *= M_PI / 180.0;
24
25   /* Set the amplitude and phase. */
26   harmonicInit1(amp, phase);
27
28   for (mm = 0; mm < num_points; mm++) {
29     x = 2.0 * M_PI * mm / (float)(num_points - 1);
30     printf("%f %f\n", x, harmonic1(x));
31   }
32
33   return 0;
34 }
```

The file containing the functions `harmonicInit1()` and `harmonic1()` is shown in Program 4.8. In line 12 two static double variables are declared: `amp` is the amplitude and `phase` is the phase. These variables are visible to all the functions in this file but are not visible to any other functions (despite the common name, these variables are distinct from those with the same name in the `main()` function). Furthermore, the value of these variables will "persist." They will remain unchanged (unless we explicitly change them) and available for our use through the duration of the running of the program. (Note that, it may perhaps be somewhat confusing, but the "`static`" qualifier in line 12 does not mean constant. The value of these variables can be changed. Rather, it means these global variables are local to this file.)

The `harmonicInit1()` function starts on line 17. It takes two arguments. Here those arguments are labeled `the_amp` and `the_phase`. We must distinguish these variable names from the corresponding global variables. We accomplish this by putting the prefix `the_` on the corresponding global variable name. The global variables are set to the desired values in lines 18 and 19. The `harmonic1()` function that begins on line 25 then uses these global values in the calculation of $a\cos(x+\phi)$.

Program 4.8 `harmonic-demo1.c`: File containing the functions `harmonicInit1()` and

harmonic1().

```
/*
 * harmonic-demo1.c: Functions to calculate a harmonic function of a
 * given amplitude and phase.  The desired amplitude and phase are
 * passed as arguments to harmonicInit1().  harmonicInit1() sets the
 * corresponding static global variables so that these values will be
 * available for the harmonic1() function to use whenever it is
 * called.
 */

#include <math.h> // for cos() function

/* Global static variables that are visible only to functions inside
   this file. */
static double amp, phase;

// initialization function
void harmonicInit1(double the_amp, double the_phase) {
  amp = the_amp;
  phase = the_phase;

  return;
}

// calculation function
double harmonic1(double x) {
  return amp * cos(x + phase);
}
```

Now, let us change these programs in order to further separate the `main()` function from the harmonic function. There is no reason that the `main()` function should have to prompt the user for the amplitude or phase. These values are simply passed along to the harmonic initialization function and never actually used in `main()`. Thus, a better approach would be to let the harmonic initialization function prompt the user for whatever input it needs. The `main()` function would merely call the initialization function and leave all the details up to it. So in the future, if one wanted to change the harmonic function so the user could specify a frequency as well as the amplitude and phase, that code could be added to the harmonic functions but the `main()` function would not have to be changed in any way.

The new version of the `main()` function is shown in Program 4.9. Note that there is now no mention of amplitude or phase in `main()`. That information has all been relegated to the harmonic functions themselves.

Program 4.9 `param-demo2.c`: Modified file containing the `main()` function which demonstrates the use of an initialization function to set parameters. In this version of the code the ini-

tilization function `harmonicInit2()` takes no arguments. The code for `harmonicInit2()` and `harmonic2()` is given in Program 4.10.

```
1  /* param-demo2.c: Program that demonstrates the setting of
2   * "persistent" parameters via an initialization function.  Here the
3   * initialization function handles all the details of obtaining the
4   * parameters associated with the harmonic function.
5   */
6
7  #include <stdio.h>
8  #include <math.h>  // To obtain M_PI, i.e., 3.14159...
9
10 void harmonicInit2();
11 double harmonic2(double x);
12
13 int main() {
14   double x;
15   int mm, num_points = 100;
16
17   /* Initialize the harmonic function. */
18   harmonicInit2();
19
20   for (mm = 0; mm < num_points; mm++) {
21     x = 2.0 * M_PI * mm / (float)(num_points - 1);
22     printf("%f %f\n", x, harmonic2(x));
23   }
24
25   return 0;
26 }
```

The file containing `harmonicInit2()` and `harmonic2()` is shown in Program 4.10. As before, the amplitude and phase are global static variables that are declared in line 13. Note that `harmonicInit2()` takes no arguments. Instead, this function prompts the user for the amplitude and phase and sets the global variables appropriately. Having done this, these values are visible to the function `harmonic2()` (which is unchanged from the function `harmonic1()` given previously).

Program 4.10 `harmonic-demo2.c`: File containing the functions `harmonicInit2()` and `harmonic2()`.

```
1  /*
2   * harmonic-demo2.c: Functions to calculate a harmonic function of a
3   * given amplitude and phase.  harmonicInit2() prompts the user for
4   * the amplitude and phase and sets the global variables so these
```

```
 * values will be available for the harmonic2() function to use
 * whenever it is called.
 */

#include <math.h> // for cos() function

/* Global static variables that are visible only to functions inside
   this file. */
static double amp, phase;

// initialization function
void harmonicInit2() {

  printf("Enter the amplitude: ");
  scanf(" %lf", &amp);
  printf("Enter the phase [in degrees]: ");
  scanf(" %lf", &phase);
  phase *= M_PI / 180.0;

  return;
}

// calculation function
double harmonic2(double x) {
  return amp * cos(x + phase);
}
```

In Sec. 4.8 we will discuss the compilation of multi-file programs such as these.

4.7 Improvement Number Two

Let us consider a further refinement to the simple bare-bones FDTD simulation. In this version of the code the updating of the electric and magnetic fields will be handled by separate functions. The grid arrays will be initialized with a separate function and the source function will be calculated using a separate function. The arrangement of the functions among the various files is depicted in Fig. 4.2.

Program 4.11 shows the contents of the file `improved2.c`. As indicated in Fig. 4.2, `main()` calls `gridInit2()`. This function initializes the `Grid` structure `g`. The function `gridInit2()` is contained in the separate file `gridinit2.c`. The magnetic fields are updated using the function `updateH2()` while the electric fields are updated using `updateE2()`. Both these functions are contained in the file `update2.c`. The source function is calculated using the function `ezInc()` that is contained in the file `ezinc2.c`.

Program 4.11 `improved2.c`: Further code improvement of the bare-bones 1D FDTD simulation. Here the initialization of the grid as well as the updating of the fields are handled by separate

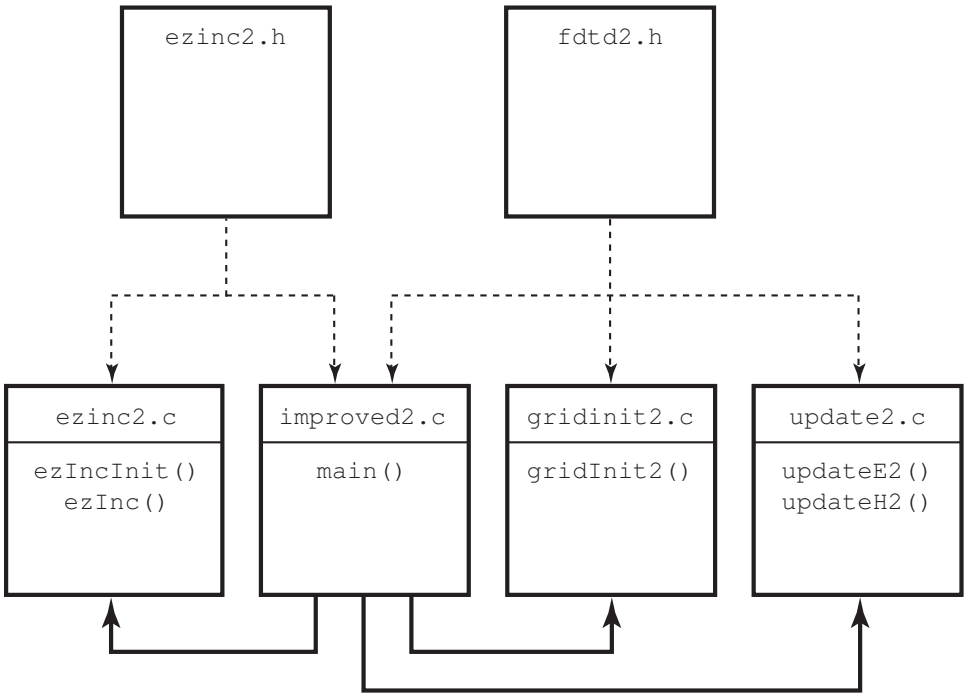


Figure 4.2: The files associated with the second improved version of the FDTD code. The header files `fdtd2.h` and `ezinc2.h` are included in the source files to which they are joined by a dashed line. The file `improved2.c` contains the `main()` function but the initialization of the grid, the calculation of the source function, and the updating of the fields are no longer done in `main()`. Instead, other functions are called to accomplish these tasks. The heavy black lines indicates which functions call which other functions. In this case, `main()` originates calls to all the other functions.

functions. The argument for these functions is merely the `Grid` pointer `g`. Additionally, the source function is initialized and calculated by separate functions.

```
1  /* Version 2 of the improved bare-bones 1D FDTD simulation. */
2
3  #include "fdtd2.h"
4  #include "ezinc2.h"
5
6  int main()
7  {
8    Grid *g;
9
10   ALLOC_1D(g, 1, Grid);       // allocate memory for Grid
11   gridInit2(g);               // initialize the grid
12
13   ezIncInit(g);               // initialize source function
14
15   /* do time stepping */
16   for (Time = 0; Time < MaxTime; Time++) {
17     updateH2(g);              // update magnetic field
18     updateE2(g);              // update electric field
19     Ez(0) = ezInc(Time, 0.0); // apply source function
20     printf("%g\n", Ez(50));   // print output
21   } // end of time-stepping
22
23   return 0;
24 }
```

Line 8 declares `g` to be a pointer to a `Grid`. Since a `Grid` structure has as elements the field arrays, the time step, the duration of the simulations, and the maximum number of time steps, none of these variables need to be declared explicitly. Note, however, that line 8 merely creates a pointer but as yet this pointer does not point to anything meaningful. Line 10 uses `ALLOC_1D()` to allocated memory for the `Grid` and ensures `g` points to that memory. Assuming there were no errors in the allocation, this line is effectively the equivalent of

```
g = calloc(1, sizeof(Grid));
```

As shown in lines 11, 17, and 18, `gridInit2()`, `updateH2()`, and `updateE2()` each have a single argument: `g` (a pointer to the `Grid`). The parameters of the source function are initialized by calling `ezIncInit()` in line 13.

The header file `fdtd2.h` is shown in Program 4.12. This file is largely the same as `fdtd1.h`. The only significant difference is the function prototypes that are provided in lines 36–38 for three of the functions called by `main()` (note that the prototypes for the functions related to the source are provided in `ezinc2.h`).

Program 4.12 `fdtd2.h`: Header file to accompany the second version of the improved code showin in Program 4.11. The differences between this file and `fdtd1.h` are shown in bold.

```
1  #ifndef _FDTD2_H
2  #define _FDTD2_H
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  struct Grid {
8    double *ez;
9    double *hy;
10   int sizeX;
11   int time, maxTime;
12   double cdtds;
13 };
14
15 typedef struct Grid Grid;
16
17 /* memory allocation macro */
18 #define ALLOC_1D(PNTR, NUM, TYPE)                                    \
19     PNTR = (TYPE *)calloc(NUM, sizeof(TYPE));                        \
20     if (!PNTR) {                                                     \
21       perror("ALLOC_1D");                                            \
22       fprintf(stderr,                                                \
23           "Allocation failed for " #PNTR ".  Terminating...\n");\
24       exit(-1);                                                      \
25     }
26
27 /* macros for accessing arrays and such */
28 #define Hy(MM)     g->hy[MM]
29 #define Ez(MM)     g->ez[MM]
30 #define SizeX      g->sizeX
31 #define Time       g->time
32 #define MaxTime    g->maxTime
33 #define Cdtds      g->cdtds
34
35 /* function prototypes */
36 void gridInit2(Grid *g);
37 void updateH2(Grid *g);
38 void updateE2(Grid *g);
39
40 #endif  /* matches #ifndef _FDTD2_H */
```

Program 4.13 shows the contents of the file `update2.c`. The static global variable `imp0` represents the characteristic impedance of free space and is set to 377.0 in line 6. This variable is never

changed throughout the program. The magnetic field is updated with the function `updateH2()` which is given between lines 9 and 16. Note that the update equation uses `Hy()` and `Ez()` to refer to the elements of the field arrays. The macros in `fdtd2.h` translate these to the necessary syntax (which is essentially `g->hy[]` and `g->ez[]`). The electric field is updated using `updateE2()` which is given between lines 19 and 26.

Program 4.13 `update2.c`: Source code for the functions `updateH2()` and `updateE2()`.

```
1  /* Functions to update the electric and magnetic fields. */
2
3  #include "fdtd2.h"
4
5  /* characteristic impedance of free space */
6  static double imp0 = 377.0;
7
8  /* update magnetic field */
9  void updateH2(Grid *g) {
10   int mm;
11
12   for (mm = 0; mm < SizeX - 1; mm++)
13     Hy(mm) = Hy(mm) + (Ez(mm + 1) - Ez(mm)) / imp0;
14
15   return;
16 }
17
18 /* update electric field */
19 void updateE2(Grid *g) {
20   int mm;
21
22   for (mm = 1; mm < SizeX - 1; mm++)
23     Ez(mm) = Ez(mm) + (Hy(mm) - Hy(mm - 1)) * imp0;
24
25   return;
26 }
```

Program 4.14 shows the source code for the function `gridInit2()`. This function is used to set the value of various elements of the `Grid`. (For this rather simple simulation, this program is itself quite simple.) Line 6 sets the size of the Grid, `SizeX`, to 200. Actually, after the preprocessor has processed the code, this line will be

```
g->sizeX = 200;
```

Line 7 sets the duration of the simulation to 250 time-steps. Line 8 sets the Courant number to unity. Lines 10 and 11 use the `ALLOC_1D()` macro to allocate the necessary memory for the electric and magnetic field arrays.

Program 4.14 `gridinit2.c`: Source code for the function `gridInit2()`.

```
1  /* Function to initialize the Grid structure. */
2
3  #include "fdtd2.h"
4
5  void gridInit2(Grid *g) {
6    SizeX = 200;                    // set the size of the grid
7    MaxTime = 250;                  // set duration of simulation
8    Cdtds = 1.0;                    // set Courant number
9
10   ALLOC_1D(g->ez, SizeX, double); // allocate memory for Ez
11   ALLOC_1D(g->hy, SizeX, double); // allocate memory for Hy
12
13   return;
14 }
```

Finally, Program 4.15 shows the contents of the file `ezinc2.c` which contains the code to implement the functions `ezIncInit()` and `ezInc()`. The function `ezinc()` is Gaussian pulse whose width and delay are parameters that are set by `ezIncInit()`. The implementation of this source function is slightly different than in the original bare-bones code in that here the user is prompted for the width and delay. Additionally, the source function `ezInc()` takes two arguments, the time and the location, so that this function can be used in a TFSF formulation. When `ezInc()` is called from `main()`, the location is simply hardwired to zero (ref. line 19 of Program 4.11).

Program 4.15 `ezinc2.c`: File for the functions `ezIncInit()` and `ezInc()`. `ezInc()` is a traveling-wave implementation of a Gaussian pulse. There are three private static global variables in this code. These variables, representing the width and delay of the pulse as well as the Courant number, are determined and set by the initialization function `ezIncInit()`. The Courant number is taken from the `Grid` pointer that is passed as an argument while the user is prompted to provide the width and delay.

```
1  /* Functions to calculate the source function (i.e., the incident
2   * field). */
3
4  #include "ezinc2.h"
5
6  /* global variables -- but private to this file */
7  static double delay, width = 0, cdtds;
8
9  /* prompt user for source-function width and delay. */
10 void ezIncInit(Grid *g){
11
```

```
12   cdtds = Cdtds;
13   printf("Enter delay: ");
14   scanf(" %lf", &delay);
15   printf("Enter width: ");
16   scanf(" %lf", &width);
17
18   return;
19 }
20
21 /* calculate source function at given time and location */
22 double ezInc(double time, double location) {
23   if (width <= 0) {
24     fprintf(stderr,
25        "ezInc: must call ezIncInit before ezInc.\n"
26        "       Width must be positive.\n");
27     exit(-1);
28   }
29   return exp(-pow((time - delay - location / cdtds) / width, 2));
30 }
```

In Program 4.15 the variable `width` is used to determine if initialization has been done. `width` is initialized to zero when it is declared in line 7. If it is not positive when `ezInc()` is called, an error message is printed and the program terminates. In practice one should check that all the parameters have been set to reasonable values, but here we only check the width.

The private variable `cdtds` declared in line 7 is distinct from `Cdtds` which the preprocessor expands to `g->cdtds`. That is to say the Courant number `cdtds` that is an element within the `Grid` is different from the private variable `cdtds` in this file. But, of course, we want these values to be the same and line 12 assures this.

The header file to accompany `ezinc2.c` is shown in Program `ezinc2.h`. As well as providing the necessary function prototypes, this file ensures the inclusion of `fdtd2.h` (which provides the description a `Grid` structure).

Program 4.16 `ezinc2.h`: Header file that accompanies `ezinc2.c` and is also included in the file that specifies `main()` (i.e., the file `improved2.c`.

```
1  /* Header file to accompany ezinc2.c. */
2
3  #ifndef _EZINC2_H
4  #define _EZINC2_H
5
6  #include <math.h>
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include "fdtd2.h"
10
```

```
11 void ezIncInit(Grid *g);
12 double ezInc(double time, double location);
13
14 #endif  /* matches #ifndef _EZINC2_H */
```

4.8 Compiling Modular Code

When a program is divided between multiple files it is typically not necessary to recompile every file if there was a change in only some of them. Each source file can be compiled individually to an "object file" or object code. An object file, by itself, is not executable. (To create executable code, all the object code must be linked together.) To create an object file with the GNU C compiler, one uses the `-c` flag. Thus, for example, to obtain the object file for `ezinc2.c`, one would issue a command such as

```
gcc -Wall -O -c ezinc2.c
```

The flag `-Wall` means to show all warnings, while `-O` tells the compiler to optimize the code (greater optimization can be obtained by instead using `-O2` or `-O3`—the greater optimization usually comes at the cost of slower compilation and larger object and executable files). When this command is issued the compiler will create the object file `ezinc2.o`.

The object files for all the components of the program can be created separately by reissuing commands similar to the one shown above, e.g.,

```
gcc -Wall -O -c ezinc2.c
gcc -Wall -O -c improved2.c
gcc -Wall -O -c update2.c
gcc -Wall -O -c gridinit2.c
```

These commands would create the object files `ezinc2.o`, `improved2.o`, `update2.o`, and `gridinit2.o`. Alternatively, one can create all the object files at once using a command such as

```
gcc -Wall -O -c ezinc2.c improved2.c update2.c gridinit2.c
```

No matter how the object files are created, they need to be linked together to obtain an executable. The command to accomplish this is

```
gcc ezinc2.o improved2.o update2.o gridinit2.o -lm -o improved2
```

The flag `-lm` tells the compiler to link to the math library (which is necessary because of the math functions used in `ezinc2.c`). The `-o` flag allows one to specify the name of the output/executable file, in this case `improved2`.

For small programs there is not much advantage to incremental compilation. However, as the software increases in size and complexity, there may be a significant savings in time realized by recompiling only the code that has changed. For example, assume a change was made in the file `gridinit2.c` but in no other. Also assume that the object code for each file has previously been created. To obtain an executable that reflects this change one merely needs to recompile this one file and then link the resulting object file to the others, i.e.,

```
gcc -Wall -O -c gridinit2.c
gcc ezinc2.o improved2.o update2.o gridinit2.o -lm -o improved2
```

The details of incremental compilations can actually be handled by utilities that detect the files that need to be recompiled and react accordingly. Those who are interested in an example of such a utility may be interested in the `make` utility which is available on most Unix-based machines. Another helpful utility, which goes hand-in-hand with `make` is `makedepend` which sorts out the dependence of all the source files on the header files. `make` is a rather old utility—going back to the early days of Unix—and there are alternatives available such as SCons available from `www.scons.org`.

4.9 Improvement Number Three

Now let us modularize the program that contained the matched lossy layer (Program 3.8). As shown in Fig. 4.3, we will use separate functions for initializing the grid, taking snapshots, applying the TFSF boundary, updating the grid, and applying the ABC. Additionally, the calculation of the incident source function will be handled by a separate function. This source function will be called by the function that implements the TFSF boundary, but no other. Each box in Fig. 4.3 represents a separate file that contains one or more functions. The dashed lines indicate the inclusion of header files and the heavy lines indicate function calls from one file to another.

The source code `improved3.c` which contains the `main()` function is shown in Program 4.17. Note that nearly all activities associated with the FDTD simulation are now done by separate functions. `main()` merely serves to ensure that proper initialization is done and then implements the time-stepping loop in which the various functions are called.

Program 4.17 `improved3.c`: Source code containing the `main()` function. Nearly all the FDTD related activities have been relegated to separate functions which are called from `main()`.

```
1  /* FDTD simulation where main() is primarily used to call other
2   * functions that perform the necessary operations. */
3
4  #include "fdtd3.h"
5
6  int main()
7  {
8    Grid *g;
9
10   ALLOC_1D(g, 1, Grid);  // allocate memory for Grid
11
12   gridInit3(g);         // initialize the grid
13   abcInit(g);           // initialize ABC
14   tfsfInit(g);          // initialize TFSF boundary
15   snapshotInit(g);      // initialize snapshots
```

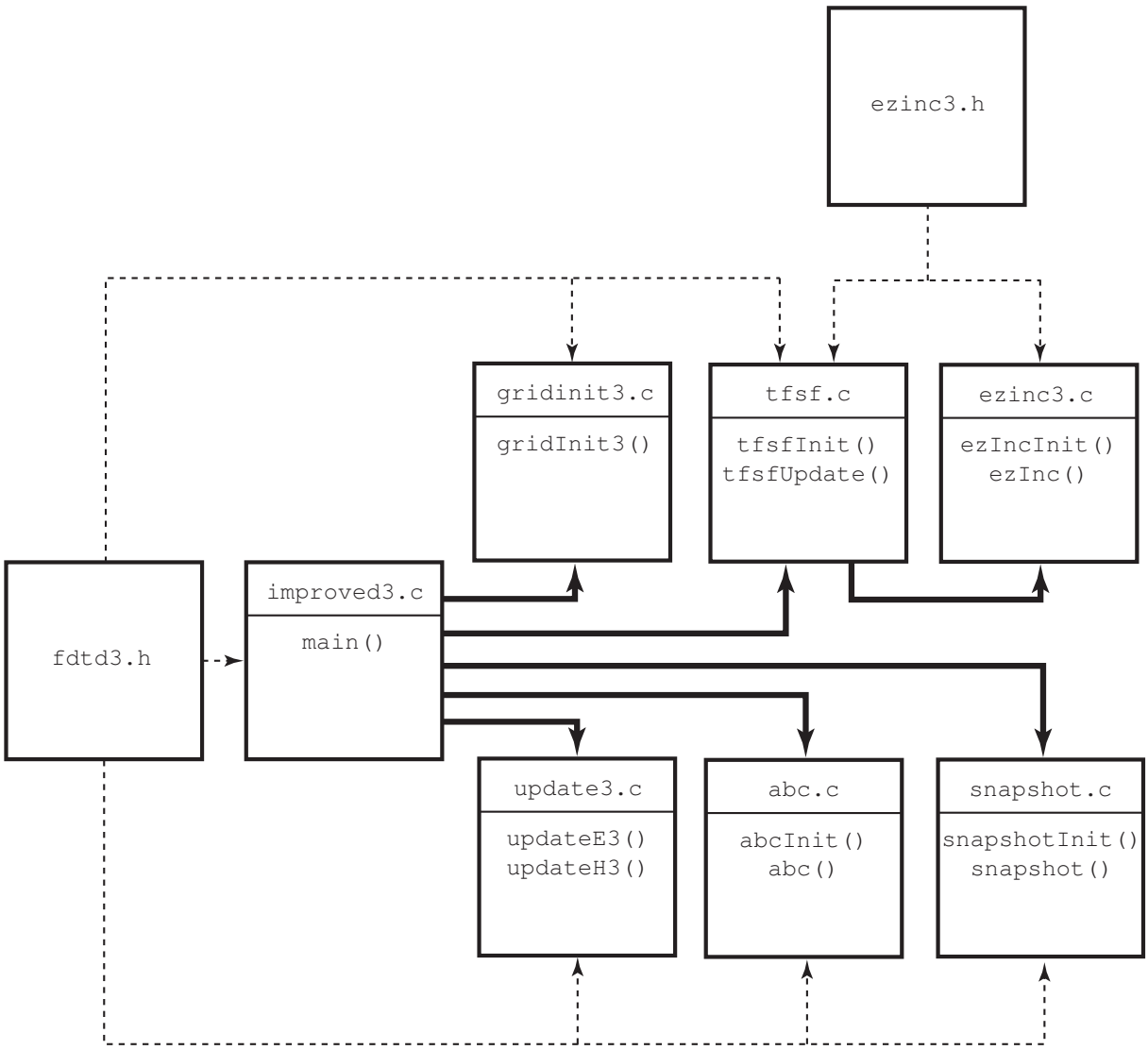


Figure 4.3: The files associated with the third improvement of the FDTD code. The header file `fdtd3.h` is explicitly included in the source files to which it is joined by a dashed line. (Wherever `ezinc3.h` appears it also ensures `fdtd3.h` is included.) The file `improved3.c` contains the `main()` function but the initialization of the grid, the application of the absorbing boundary condition, the calculation of the source function, the updating of the fields, and the taking of the snapshots is no longer done in `main()`. Instead, other functions are called to accomplish these tasks.

```
16
17    /* do time stepping */
18    for (Time = 0; Time < MaxTime; Time++) {
19      updateH3(g);    // update magnetic field
20      tfsfUpdate(g); // correct field on TFSF boundary
21      abc(g);         // apply ABC
22      updateE3(g);    // update electric field
23      snapshot(g);    // take a snapshot (if appropriate)
24    } // end of time-stepping
25
26    return 0;
27  }
```

We have any number of options in terms of how functions should be initialized or what arguments they should be passed. Thus, one should not consider this code to be optimum in any way. Rather this code is being used to illustrate implementation options.

As in the previous program, `main()` starts by defining a `Grid` pointer and allocating space for the actual structure. Then, in lines 12–15, four initialization functions are called. `gridInit3()` initializes the `Grid` (this will be discussed in more detail shortly). `abcInit()` handles any initialization associated with the ABC (as we will see, in this particular case there is nothing for this initialization function to do). `tfsfInit()` initializes the TFSF boundary while `snapshotInit()` does the necessary snapshot initialization. Following these initialization steps is the time-stepping loop where the various functions are called that do the actual calculations.

The header `fdtd3.h` is shown in Program 4.18. Looking just at `improved3.c` you would be unaware that the `Grid` structure has changed. However, four pointers have been added that will contain the update-equation coefficients. As can be seen in lines 8 and 9 of Program 4.18, these pointers are named `ceze`, `cezh`, `chyh`, and `chye`. Besides this and besides providing the function prototypes for the new functions, this header file is largely the same as `fdtd2.h`.

Program 4.18 `fdtd3.h`: Header file to accompany `improved3.c`. Differences from `fdtd2.h` are shown in bold.

```
1   #ifndef _FDTD3_H
2   #define _FDTD3_H
3
4   #include <stdio.h>
5   #include <stdlib.h>
6
7   struct Grid {
8     double *ez, *ceze, *cezh;
9     double *hy, *chyh, *chye;
10    int sizeX;
11    int time, maxTime;
12    double cdtds;
```

```
13 };
14
15 typedef struct Grid Grid;
16
17 /* macros for accessing arrays and such */
18 #define Hy(MM)    g->hy[MM]
19 #define Chyh(MM)  g->chyh[MM]
20 #define Chye(MM)  g->chye[MM]
21
22 #define Ez(MM)    g->ez[MM]
23 #define Ceze(MM)  g->ceze[MM]
24 #define Cezh(MM)  g->cezh[MM]
25
26 #define SizeX     g->sizeX
27 #define Time      g->time
28 #define MaxTime   g->maxTime
29 #define Cdtds     g->cdtds
30
31 /* memory allocation macro */
32 #define ALLOC_1D(PNTR, NUM, TYPE)                                    \
33     PNTR = (TYPE *)calloc(NUM, sizeof(TYPE));                        \
34     if (!PNTR) {                                                     \
35       perror("ALLOC_1D");                                            \
36       fprintf(stderr,                                                \
37           "Allocation failed for " #PNTR ".  Terminating...\n");\
38       exit(-1);                                                      \
39     }
40
41 /* Function prototypes */
42 void abcInit(Grid *g);
43 void abc(Grid *g);
44
45 void gridInit3(Grid *g);
46
47 void snapshotInit(Grid *g);
48 void snapshot(Grid *g);
49
50 void tfsfInit(Grid *g);
51 void tfsfUpdate(Grid *g);
52
53 void updateE3(Grid *g);
54 void updateH3(Grid *g);
55
56 #endif  /* matches #ifndef _FDTD3_H */
```

The function `gridInit3()` is contained in the file `gridinit3.c` shown in Program 4.19. Keep in mind that it does not matter what file names are—file names do not have to match the

contents of the file in any way, but, of course, it is best to use names that are descriptive of the contents.

The preprocessor directives in lines 5–7 are simply to provide convenient names to the amount of loss, the starting location of the lossy layer, and the relatively permittivity of the half space. These parameters are the same as they were in Program 3.8.

Program 4.19 `gridinit3.c`: The `gridInit3()` function to initialize the `Grid`.

```
1  /* Function initialize Grid structure. */
2
3  #include "fdtd3.h"
4
5  #define LOSS 0.02
6  #define LOSS_LAYER 180  // node at which lossy layer starts
7  #define EPSR 9.0
8
9  void gridInit3(Grid *g) {
10   double imp0 = 377.0;
11   int mm;
12
13   SizeX = 200;   // size of domain
14   MaxTime = 450; // duration of simulation
15   Cdtds = 1.0;   // Courant number
16
17   ALLOC_1D(g->ez,   SizeX, double);
18   ALLOC_1D(g->ceze, SizeX, double);
19   ALLOC_1D(g->cezh, SizeX, double);
20   ALLOC_1D(g->hy,   SizeX - 1, double);
21   ALLOC_1D(g->chyh, SizeX - 1, double);
22   ALLOC_1D(g->chye, SizeX - 1, double);
23
24   /* set electric-field update coefficients */
25   for (mm = 0; mm < SizeX; mm++)
26     if (mm < 100) {
27       Ceze(mm) = 1.0;
28       Cezh(mm) = imp0;
29     } else if (mm < LOSS_LAYER) {
30       Ceze(mm) = 1.0;
31       Cezh(mm) = imp0 / EPSR;
32     } else {
33       Ceze(mm) = (1.0 - LOSS) / (1.0 + LOSS);
34       Cezh(mm) = imp0 / EPSR / (1.0 + LOSS);
35     }
36
37   /* set magnetic-field update coefficients */
38   for (mm = 0; mm < SizeX - 1; mm++)
```

```
39      if (mm < LOSS_LAYER) {
40        Chyh(mm) = 1.0;
41        Chye(mm) = 1.0 / imp0;
42      } else {
43        Chyh(mm) = (1.0 - LOSS) / (1.0 + LOSS);
44        Chye(mm) = 1.0 / imp0 / (1.0 + LOSS);
45      }
46
47    return;
48  }
```

Lines 13–15 set the size of the `Grid`, the duration of the simulation, and the Courant number. Lines 17–22 allocate the necessary memory for the various arrays. The coefficient arrays are then set as they were in Program 3.8.

The functions to update the electric and magnetic fields, i.e., `updateE3()` and `updateH3()` are contained in the file `update3.c`. The contents of this file are shown in Program 4.20. The functions are largely unchanged from those that appeared in Program 4.13. The only significant differences are the appearance of the coefficient arrays in the update equations that start on lines 10 and 21.

Program 4.20 `update3.c`: Functions to update the electric and magnetic fields.

```
1  /* Functions to update the electric and magnetic fields. */
2
3  #include "fdtd3.h"
4
5  /* update magnetic field */
6  void updateH3(Grid *g) {
7    int mm;
8
9    for (mm = 0; mm < SizeX - 1; mm++)
10     Hy(mm) = Chyh(mm) * Hy(mm) +
11              Chye(mm) * (Ez(mm + 1) - Ez(mm));
12
13   return;
14 }
15
16 /* update electric field */
17 void updateE3(Grid *g) {
18   int mm;
19
20   for (mm = 1; mm < SizeX - 1; mm++)
21     Ez(mm) = Ceze(mm) * Ez(mm) +
22              Cezh(mm) * (Hy(mm) - Hy(mm - 1));
23
```

```
24   return;
25 }
```

The function to apply the absorbing boundary conditions is rather trivial and is shown in Program 4.21. Also shown in Program 4.21 is the initiliztaion function `abcInit()`. In this particular case the ABC is so simple that there is no initializataion that needs to be done and hence this function simply returns. In Chap. 6 we will begin to consider more sophisticated ABC's that do indeed require some initialization. Thus, this call the initialization function is done in anticipation of that. As was the case in Program 3.8, the ABC is only applied to the left side of the grid. The right side of the grid is terminated with a lossy layer.

Program 4.21 `abc.c`: Absorbing boundary condition used by `improved3.c`. For this particular simple ABC there is nothing for the initialization function to do and hence it simply returns.

```
1  /* Functions to terminate left side of grid. */
2
3  #include "fdtd3.h"
4
5  // Initialize the ABC -- in this case, there is nothing to do.
6  void abcInit(Grid *g) {
7
8    return;
9  }
10
11 // Apply the ABC -- in this case, only to the left side of grid.
12 void abc(Grid *g) {
13
14   /* simple ABC for left side of grid */
15   Ez(0) = Ez(1);
16
17   return;
18 }
```

The code associated with the TFSF boundary is shown in Program 4.22. Line 7 declares a static global variable `tfsfBoundary` which specifies the location of the TFSF boundary. This variable is initialized to zero with the understanding that when the code is initialized it will be set to some meaningful (positive) value.

Program 4.22 `tfsf.c`: Code to implement the TFSF boundary.

```
1  /* Function to implement a 1D FDTD boundary. */
2
```

```
3  #include <math.h>
4  #include "fdtd3.h"
5  #include "ezinc3.h"
6
7  static int tfsfBoundary = 0;
8
9  void tfsfInit(Grid *g) {
10
11   printf("Enter location of TFSF boundary: ");
12   scanf(" %d", &tfsfBoundary);
13
14   ezIncInit(g); // initialize source function
15
16   return;
17 }
18
19 void tfsfUpdate(Grid *g) {
20   /* check if tfsfInit() has been called */
21   if (tfsfBoundary <= 0) {
22     fprintf(stderr,
23       "tfsfUpdate: tfsfInit must be called before tfsfUpdate.\n"
24       "            Boundary location must be set to positive value.\n");
25     exit(-1);
26   }
27
28   /* correct Hy adjacent to TFSF boundary */
29   Hy(tfsfBoundary) -= ezInc(Time, 0.0) * Chye(tfsfBoundary);
30
31   /* correct Ez adjacent to TFSF boundary */
32   Ez(tfsfBoundary + 1) += ezInc(Time + 0.5, -0.5);
33
34   return;
35 }
```

The initialization function `tfsfInit()` begins on line 9. The user is prompted to enter the location of the TFSF boundary. Then the initialization function for the source `ezIncInit()` is called to set the parameters that control the shape of the pulse.

The code to calculate the source function, i.e., the functions `ezIncInit()` and `ezInc()` is identical to that shown in Program 4.15. However, there would be one slight change to the file: instead of including `ezinc2.h`, line 4 of Program 4.15 would be changed to include `ezinc3.h`. We assume this modified program is in the file `ezinc3.c` which is not shown. The header file `ezinc3.h` would be nearly identical to the code shown in Program 4.16 except the "2" in lines 3, 4, and 9, would be changed to "3" (thus ensuring the proper inclusion of the header file `fdtd3.h` instead of `fdtd2.h`). Again, since this is such a minor change, the contents of `ezinc3.h` are not shown.

The function `tfsfUpdate()` which begins on line 19 is called once every time-step. This

function applies the necessary correction to the nodes adjacent to the TFSF boundary. Because of the implicit timing within this file, this function needs to be called after the magnetic-field update, but before the electric-field update. The function first checks that the boundary location is positive. If it is not, an error message is printed and the program terminates, otherwise the fields adjacent to the boundary are corrected.

Finally, the code associated with taking snapshots of the field is shown in Program 4.23. `snapshotInit()` allows the user to specify the time at which snapshots should start, the temporal stride between snapshots, the node at which a snapshot should start, the node at which it should end, the spatial stride between nodes, and the base name of the output files. Assuming the user entered a base name of `sim`, then, as before, the output files would be named `sim.0`, `sim.1`, `sim.2`, and so on. If the user said the `startTime` is 105 and the `temporalStride` is 10, then snapshots would be taken at time-steps 105, 115, 125, and so on. Similarly, if the user specified that the `startNode` and `endNode` are 0 and 180, respectively, and the `spatialStride` is 1, then the value of every node between 0 and 180, inclusive, would be recorded to the snapshot file. If the `spatialStride` were 2, every other node would be recorded. If it were 3, every third node would be recorded. (Because of this, the `endNode` only corresponds to the actual last node in the snapshot file if its offset from the `startNode` is an even multiple of the spatial stride.)

Program 4.23 `snapshot.c`: Code for taking snapshots of the electric field.

```
1  /* Function to take a snapshot of a 1D grid. */
2
3  #include "fdtd3.h"
4
5  static int temporalStride = 0, spatialStride, startTime,
6    startNode, endNode, frame = 0;
7  static char basename[80];
8
9  void snapshotInit(Grid *g) {
10
11   printf("For the snapshots:\n");
12   printf("  Duration of simulation is %d steps.\n", MaxTime);
13   printf("  Enter start time and temporal stride: ");
14   scanf(" %d %d", &startTime, &temporalStride);
15   printf("  Grid has %d total nodes (ranging from 0 to %d).\n",
16          SizeX, SizeX-1);
17   printf("  Enter first node, last node, and spatial stride: ");
18   scanf(" %d %d %d", &startNode, &endNode, &spatialStride);
19   printf("  Enter the base name: ");
20   scanf(" %s", basename);
21
22   return;
23 }
24
25 void snapshot(Grid *g) {
```

```
26   int mm;
27   char filename[100];
28   FILE *snapshot;
29
30   /* ensure temporal stride set to a reasonable value */
31   if (temporalStride <= 0) {
32     fprintf(stderr,
33       "snapshot: snapshotInit must be called before snapshot.\n"
34       "          Temporal stride must be set to positive value.\n");
35     exit(-1);
36   }
37
38   /* get snapshot if temporal conditions met */
39   if (Time >= startTime &&
40       (Time - startTime) % temporalStride == 0) {
41     sprintf(filename, "%s.%d", basename, frame++);
42     snapshot = fopen(filename, "w");
43     for (mm = startNode; mm <= endNode; mm += spatialStride)
44       fprintf(snapshot, "%g\n", Ez(mm));
45     fclose(snapshot);
46   }
47
48   return;
49 }
```

As shown in line 9, `snapshotInit()` takes a single argument, a pointer to a `Grid` structure. This function prompts the user to set the appropriate snapshot control values. The snapshot files themselves are created by the function `snapshot()` which starts on line 25. This function starts by checking that the temporal stride has been set to a reasonable value. If not, an error message is printed and the program terminates. The program then checks if time is greater than or equal to `startTime` and the difference between the current time and the `startTime` is a multiple of `temporalStride`. If not, the function returns. If those conditions are met, then, as we have seen before, a snapshot file is written and the frame counter is advanced. However, now there is some control over which nodes are actually written. Note that the function `snapshot()` is called every time-step. We could easily have checked in the `main()` function if the time-step was such that a snapshot should be generated and only then called `snapshot()`. Reasons for adopting either approach could be made but be aware that the overhead for calling a function is typically small. Thus, the savings realized by calling `snapshot()` less often (but still ultimately generating the same number of snapshots) is likely to be trivial. The approach used here keeps all the snapshot-related data in the snapshot code itself.

After compiling all this code (and linking it), the executable will produce the same results as were obtained from Program 3.8 (assuming, of course, the user enters the same parameters as were used in Program 3.8). With the new version of the code there are several improvements we could potentially use to our advantage. Assume, for instance, we wanted to do simulations of two different scenarios. We could create two `Grid` structures, one for scenario. Each grid would have

its own `gridInit()` function, but other than that all the code could be used by either grid. The update functions could be applied, without any modification, to any grid. The snapshot function could be applied to any grid, and so on.

Chapter 5

Scaling FDTD Simulations to Any Frequency

5.1 Introduction

The FDTD method requires the discretization of time and space. Samples in time are Δ_t apart whereas, in simulations with one spatial dimension, samples in space are Δ_x apart. It thus appears that one must specify Δ_t and Δ_x in order to perform a simulation. However, as shown in Sec. 3.3, it is possible to write the coefficients $\Delta_t/\epsilon\Delta_x$ and $\Delta_t/\mu\Delta_x$ in terms of the material parameters and the Courant number (ref. (3.19) and (3.20)). Since the Courant number contains the ratio of the temporal step to the spatial step, it allows one to avoid explicitly stating a definite temporal or spatial step—all that matters is their ratio. This chapter continues to examine ways in which FDTD simulations can be treated as generic simulations that can be scaled to any size/frequency. As will be shown, the important factors which dictate the behavior of the fields in a simulation are the Courant number and the points per wavelength for any given frequency. We conclude the chapter by considering how one can obtain the transmission coefficient for a planar interface from a FDTD simulation.

5.2 Sources

5.2.1 Gaussian Pulse

In the previous chapter the source function, whether hardwired, additive, or incorporated in a TFSF formulation, was always a Gaussian. In the continuous world this function can be expressed as

$$f_g(t) = e^{-\left(\frac{t-d_g}{w_g}\right)^2} \tag{5.1}$$

where d_g is the temporal delay and w_g is a pulse-width parameter. The Gaussian has its peak value at $t = d_g$ (when the exponent is zero) and has a value of e^{-1} when $t = d_g \pm w_g$. Since (5.1) is only a function of time, i.e., a function of $q\Delta_t$ in the discretized world, it again appears as if the temporal step Δ_t must be given explicitly. However, if one specifies the delay and pulse width in

Lecture notes by John Schneider. `fdtd-dimensionless.tex`

terms of temporal steps, the term Δ_t appears in both the numerator and the denominator of the exponent. For example, in the last chapter d_g was $30\Delta_t$ and w_g was $10\Delta_t$ so the source function could be written

$$f_g(q\Delta_t) = f_g[q] = e^{-\left(\frac{q-30}{10}\right)^2}. \tag{5.2}$$

Note that Δ_t does not appear on the right-hand side.

The discretized version of a function $f(q\Delta_t)$ will be written $f[q]$, i.e., the temporal step will be dropped from the argument since it does not appear explicitly in the expression for the function itself. The key to being able to discard Δ_t from the source function was the fact that the source parameters were expressed in terms of the number of temporal steps.

5.2.2 Harmonic Sources

For a harmonic source, such as

$$f_h(t) = \cos(\omega t), \tag{5.3}$$

there is no explicit numerator and denominator in the argument. Replacing t with $q\Delta_t$ it again appears as if the temporal step must be given explicitly. However, keep in mind that with electromagnetic fields there is an explicit relationship between frequency and wavelength. For a plane wave propagating in free space, the wavelength λ and frequency f are related by

$$f\lambda = c \qquad \Rightarrow \qquad f = \frac{c}{\lambda}. \tag{5.4}$$

Thus the argument ωt (i.e., $2\pi f t$) can also be written as

$$\omega t = \frac{2\pi c}{\lambda} t. \tag{5.5}$$

For a given frequency, the wavelength is a fixed length. Being a length, it can be expressed in terms of the spatial step, i.e.,

$$\lambda = N_\lambda \Delta_x \tag{5.6}$$

where N_λ is the number of points per wavelength. This does not need to be an integer.

By relating frequency to wavelength and the wavelength to N_λ, the discretized version of the harmonic function can be written

$$f_h(q\Delta_t) = \cos\left(\frac{2\pi c}{N_\lambda \Delta_x} q\Delta_t\right) = \cos\left(\frac{2\pi}{N_\lambda}\frac{c\Delta_t}{\Delta_x} q\right), \tag{5.7}$$

or, simply,

$$f_h[q] = \cos\left(\frac{2\pi S_c}{N_\lambda} q\right). \tag{5.8}$$

The expression on the right now contains the Courant number and the parameter N_λ. In this form one does not need to state an explicit value for the temporal step. Rather, one specifies the Courant number S_c and the number of spatial steps per wavelength N_λ. Note that N_λ we will always be defined in terms of the number of spatial steps per the wavelength in free space. Furthermore, the wavelength is the one in the continuous world—we will see later that the wavelength in the FDTD grid is not always the same.

The period for a harmonic function is the inverse of the frequency

$$T = \frac{1}{f} = \frac{\lambda}{c} = \frac{N_\lambda \Delta_x}{c}. \tag{5.9}$$

The number of time steps in a period is thus given by

$$\frac{T}{\Delta_t} = \frac{N_\lambda \Delta_x}{c\Delta_t} = \frac{N_\lambda}{S_c}. \tag{5.10}$$

A harmonic wave traveling in the positive x direction is given by

$$f_h(x,t) = \cos(\omega t - kx) = \cos\left(\omega\left(t - \frac{k}{\omega}x\right)\right). \tag{5.11}$$

Since $k = \omega\sqrt{\mu_0\mu_r\epsilon_0\epsilon_r} = \omega\sqrt{\mu_r\epsilon_r}/c$, the argument can be written

$$\omega\left(t - \frac{k}{\omega}x\right) = \omega\left(t - \frac{\sqrt{\mu_r\epsilon_r}}{c}x\right). \tag{5.12}$$

Expressing all quantities in terms of the discrete values which pertain in the FDTD grid yields

$$\omega\left(t - \frac{\sqrt{\mu_r\epsilon_r}}{c}x\right) = \frac{2\pi c}{N_\lambda \Delta_x}\left(q\Delta_t - \frac{\sqrt{\mu_r\epsilon_r}}{c}m\Delta_x\right) = \frac{2\pi}{N_\lambda}(S_c q - \sqrt{\mu_r\epsilon_r}m). \tag{5.13}$$

Therefore the discretized form of (5.11) is given by

$$f_h[m,q] = \cos\left(\frac{2\pi}{N_\lambda}\left(S_c q - \sqrt{\mu_r\epsilon_r}m\right)\right). \tag{5.14}$$

This equation could now be used as the source in a total-field/scattered-field implementation. (However, note that when the temporal and spatial indices are zero this source has a value of unity. If that is the initial "turn-on" value of the source, that may cause artifacts which are undesirable. This will be considered further when dispersion is discussed. It is generally better to ramp the source up gradually. A simple improvement is offered by using a sine function instead of a cosine since sine is initially zero.)

5.2.3 The Ricker Wavelet

One of the features of the FDTD technique is that it allows the modeling of a broad range of frequencies using a single simulation. Therefore it is generally advantageous to use pulsed sources—which can introduce a wide spectrum of frequencies—rather than a harmonic source. The Gaussian pulse is potentially an acceptable source except that it contains a dc component. In fact, for a Gaussian pulse dc is the frequency with the greatest energy. Generally one would not use the FDTD technique to model dc fields. Sources with dc components also have the possibility of introducing artifacts which are not physical (e.g., charges which sit in the grid). Therefore we consider a different pulsed source which has no dc component and which can have its most energetic frequency set to whatever frequency (or discretization) is desired.

The Ricker wavelet is equivalent to the second derivative of a Gaussian; it is simple to implement; it has no dc component; and, its spectral content is fixed by a single parameter. The Ricker wavelet is often written

$$f_r(t) = \Big(1 - 2\big\{\pi f_P\,[t - d_r]\big\}^2\Big)\exp\Big(-\big\{\pi f_P\,[t - d_r]\big\}^2\Big) \tag{5.15}$$

where f_P is the "peak frequency" and d_r is the temporal delay. As will be more clear when the spectral representation of the function is shown below, the peak frequency is the frequency with the greatest spectral content.

The delay d_r can be set to any desired amount, but it is convenient to express it as a multiple of $1/f_P$, i.e.,

$$d_r = M_d \frac{1}{f_P} \tag{5.16}$$

where M_d is the delay multiple (which need not be an integer). An FDTD simulation is typically assumed to start at $t = 0$, but $f_r(t)$ is not zero for $t < 0$—rather $f_r(t)$ asymptotically approaches zero for large and small values of the argument, but never actually reaches zero (other than at two discrete zero-crossings). However, with a delay of $d_r = 1/f_P$ (i.e., $M_d = 1$), $|f_r(t < 0)|$ is bound by 0.001, which is small compared to the peak value of unity. Thus, the transient caused by "switching on" $f_r(t)$ at $t = 0$ is relatively small with this amount of delay. Said another way, since the magnitude of $f_r(t)$ is small for $t < 0$, these values can be approximated by assuming they are zero. For situations that may demand a smoother transition (i.e., a smaller initial turn-on value), the bound on $|f_r(t < 0)|$ can be made arbitrarily small by increasing d_r. For example, with a delay multiple M_d of 2, $|f_r(t < 0)|$ is bound by 10^{-15}.

The Fourier transform of (5.15) is

$$F_r(\omega) = -\frac{2}{f_P\sqrt{\pi}}\left(\frac{\omega}{2\pi f_P}\right)^2 \exp\left(-jd_r\omega - \left[\frac{\omega}{2\pi f_P}\right]^2\right). \tag{5.17}$$

Note that the delay d_r only appears as the imaginary part of the exponent. Thus it affects only the phase of $F_r(\omega)$, not the magnitude.

The functions $f_r(t)$ and $|F_r(\omega)|$ are shown in Fig. 5.1. For the sake of illustration, f_P is arbitrarily chosen to be 1 Hz and the delay is 1 s. Different values of f_P change the horizontal scale but they do not change the general shape of the curve. To obtain unit amplitude at the peak frequency, $F_r(\omega)$ has been scaled by $f_P e\sqrt{\pi}/2$.

The peak frequency f_P has a corresponding wavelength λ_P. This wavelength can be expressed in terms of the spatial step such that $\lambda_P = N_P\Delta_x$, where N_P does not need to be an integer. Thus

$$f_P = \frac{c}{\lambda_P} = \frac{c}{N_P\Delta_x} \tag{5.18}$$

The Courant number S_c is $c\Delta_t/\Delta_x$ so the spatial step can be expressed as $\Delta_x = c\Delta_t/S_c$. Using this in (5.18) yields

$$f_P = \frac{S_c}{N_P\Delta_t}. \tag{5.19}$$

The delay can thus be expressed as

$$d_r = M_d\frac{1}{f_P} = M_d\frac{N_P\Delta_t}{S_c}. \tag{5.20}$$

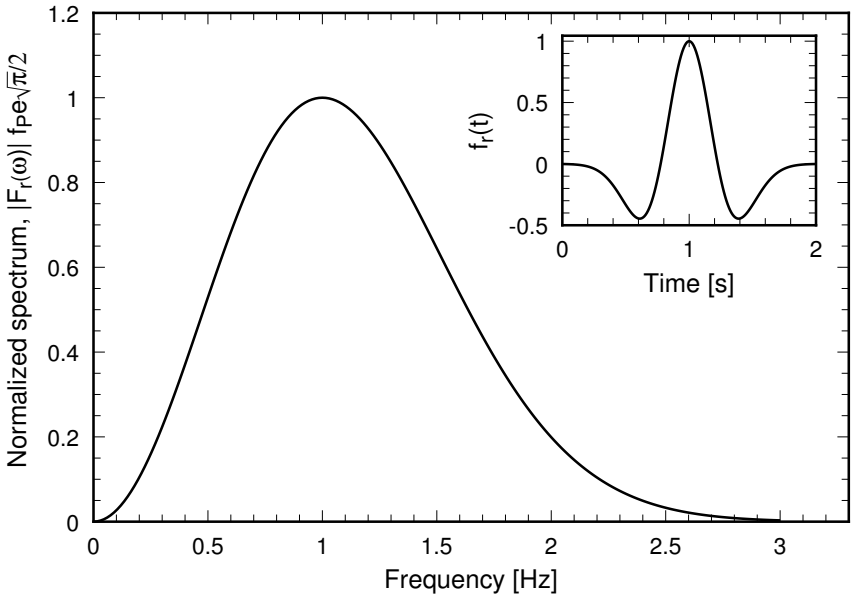


Figure 5.1: Normalized spectrum of the Ricker wavelet with $f_P = 1$ Hz. The corresponding temporal form $f_r(t)$ is shown in the inset box where a delay of 1 s has been assumed. For other values of f_P, the horizontal axis in the time domain is scaled by $1/f_P$. For example, if f_P were 1 MHz, the peak would occur at 1 μs rather than at 1 s. In the spectral domain, the horizontal axis is directly scaled by f_P so that if f_P were 1 MHz, the peak would occur at 1 MHz.

Letting time t be $q\Delta_t$ and expressing f_P and d_r as in (5.19) and (5.20), the discrete form of (5.15) can be written as

$$f_r[q] = \left(1 - 2\pi^2 \left[\frac{S_c q}{N_P} - M_d\right]^2\right) \exp\left(-\pi^2 \left[\frac{S_c q}{N_P} - M_d\right]^2\right). \tag{5.21}$$

Note that the parameters that specify $f_r[q]$ are the Courant number S_c, the points per wavelength at the peak frequency N_P, and the delay multiple M_d—there is no Δ_t in (5.21). This function appears to be independent of the temporal and spatial steps, but it does depend on their ratio via the Courant number S_c.

Equation (5.15) gives the Ricker wavelet as a function of only time. However, as was discussed in Sec. 3.10, when implementing a total-field/scattered-field boundary, it is necessary to parameterize an incident field in both time and space. As shown in Sec. 2.16, one can always obtain a traveling plane-wave solution to the wave equation simply by tweaking the argument of any function that is twice differentiable. Given the Ricker wavelet $f_r(t)$, $f_r(t \pm x/c)$ is a solution to the wave equation where c is the speed of propagation. The plus sign corresponds to a wave traveling in the negative x direction and the negative sign corresponds to a wave traveling in the positive x direction. (Although only 1D propagation will be considered here, this type of tweaking can also be done in 2D and 3D.) Therefore, a traveling Ricker wavelet can be constructed by replacing the argument t in (5.15) with $t \pm x/c$. The value of the function now depends on both time and

location, i.e., it is a function of two variables:

$$\begin{aligned} f_r(t \pm x/c) &= f_r(x,t), \\ &= \left(1 - 2\pi^2 f_P^2 \left[t \pm \frac{x}{c} - d_r\right]^2\right) \exp\left(-\pi^2 f_P^2 \left[t \pm \frac{x}{c} - d_r\right]^2\right). \end{aligned} \tag{5.22}$$

As before, (5.20) and (5.19) can be used to rewrite d_r and f_P in terms of the Courant number, the points per wavelength at the peak frequency, the temporal step, and the delay multiple. Replacing t with $q\Delta_t$, x with $m\Delta_x$, and employing the identity $x/c = m\Delta_x/c = m\Delta_t/S_c$, yields

$$f_r[m,q] = \left(1 - 2\pi^2 \left[\frac{S_c q \pm m}{N_P} - M_d\right]^2\right) \exp\left(-\pi^2 \left[\frac{S_c q \pm m}{N_P} - M_d\right]^2\right). \tag{5.23}$$

This gives the value of the Ricker wavelet at temporal index q and spatial index m. Note that when m is zero (5.23) reduces to (5.21).

5.3 Mapping Frequencies to Discrete Fourier Transforms

Assume the field was recorded during an FDTD simulation and then the recorded field was transformed to the frequency domain via a discrete Fourier transform. The discrete transform will yield a set of complex numbers that represents the amplitude of discrete spectral components. The question naturally arises: what is the correspondence between the indices of the transformed set and the actual frequency?

In any simulation, the highest frequency $f_{\max}$ that can exist is the inverse of the shortest period that can exist. In a discrete simulation one must have at least two samples per period. Since the time samples are Δ_t apart, the shortest possible period is $2\Delta_t$. Therefore

$$f_{\max} = \frac{1}{2\Delta_t}. \tag{5.24}$$

The change in frequency from one discrete frequency to the next is the spectral resolution Δ_f. The spectral resolution is dictated by the total number of samples which we will call N_T (an integer value). In general N_T would correspond to the number of time steps in an FDTD simulation. The spectral resolution is given by

$$\Delta_f = \frac{f_{\max}}{N_T/2}. \tag{5.25}$$

The factor of two is a consequence of the fact that there are both positive and negative frequencies. Thus one can think of the entire spectrum as ranging from $-f_{\max}$ to $f_{\max}$, i.e., an interval of $2f_{\max}$ which is then divided by N_T. Alternatively, to find Δ_f we can divide the maximum positive frequency by $N_T/2$ as done in (5.25). Plugging (5.24) into (5.25) yields

$$\Delta_f = \frac{1}{N_T \Delta_t}. \tag{5.26}$$

In Sec. 5.2.2 it was shown that a given frequency f could be written $c/\lambda = c/N_\lambda \Delta_x$ where N_λ is the number of points per wavelength (for the free-space wavelength). After transforming to the

spectral domain, this frequency would have a corresponding index given by

$$N_{\text{freq}} = \frac{f}{\Delta_f} = \frac{N_T c \Delta_t}{N_\lambda \Delta_x} = \frac{N_T}{N_\lambda} \frac{c\Delta_t}{\Delta_x} = \frac{N_T}{N_\lambda} S_c. \tag{5.27}$$

Thus, the spectral index is dictated by the duration of the simulation N_T, the Courant number S_c, and the points per wavelength N_λ.

Note that in practice different software packages may index things differently. The most typical practice is to have the first element in the spectral array correspond to dc, the next $N_T/2$ elements correspond to the positive frequencies, and then the next $(N_T/2) - 1$ elements correspond to the negative frequencies (which will always be the complex conjugates of the positive frequencies in any real FDTD simulation). The negative frequencies typically are stored from highest frequency to lowest (i.e., fastest varying to slowest) so that the last value in the array corresponds to the negative frequency closest to dc ($-\Delta_f$). Additionally, note that in (5.27) dc corresponds to an N_{freq} of zero (since N_λ is infinity for dc). However when using Matlab the dc term in the array obtained using the `fft()` command has an index of one. The first element in the array has an index of one—there is no "zero" element in Matlab arrays. So, one must understand and keep in mind the implementation details for a given software package.

From (5.19), the most energetic frequency in a Ricker wavelet can be written

$$f_P = \frac{S_c}{N_P \Delta_t}. \tag{5.28}$$

The spectral index corresponding to this is given by

$$N_{\text{freq}} = \frac{f_P}{\Delta_f} = \frac{N_T}{N_P} S_c. \tag{5.29}$$

This is identical to (5.27) except the generic value N_λ has been replaced by N_P which is the points per wavelength at the peak frequency.

A discrete Fourier transform yields an array of numbers which inherently has integer indices. The array elements correspond to discrete frequencies at multiples of Δ_f. However, N_{freq} given in (5.27) need not be thought of as an integer value. As an example, assume there were 65536 times steps in a simulation in which the Courant number was $1/\sqrt{2}$. Further assume that we are interested in the frequency which corresponds to 30 points per wavelength ($N_\lambda = 30$). In this case N_{freq} equals $65536/(30\sqrt{2}) = 1544.6983\cdots$. There is no reason why one cannot think in terms of this particular frequency. However, this frequency is not directly available from a discrete Fourier transform of the temporal data since the transform will only have integer values of N_{freq}. For this simulation an N_{freq} of 1544 corresponds $30.0136\cdots$ points per wavelength while an N_{freq} of 1545 corresponds $29.9941\cdots$ points per wavelength. One can interpolate between these values if the need is to measure the spectral output right at 30 points per wavelength.

5.4 Running Discrete Fourier Transform (DFT)

If one calculates the complete Fourier transform of the temporal data of length N_T points, one obtains information about the signal at N_T distinct frequencies (if one counts both positive and

negative frequencies). However, much of that spectral information is of little practical use. Because of the dispersion error inherent in the FDTD method and other numerical artifacts, one does not trust results which correspond to frequencies with too coarse a discretization. When taking the complete Fourier transform one often relies upon a separate software package such as Mathematica, Matlab, or perhaps the FFTw routines (a suite of very good C routines to perform discrete Fourier transforms). However, if one is only interested in obtaining spectral information at a few frequencies, it is quite simple to implement a discrete Fourier transform (DFT) which can be incorporated directly into the FDTD code. The DFT can be performed as the simulation progresses and hence the temporal data does not have to be recorded. This section outlines the steps involved in computing such a "running DFT."

Let us consider a discrete signal $g[q]$ which is, at least in theory, periodic with a period of N_T so that $g[q] = g[q+N_T]$ (in practice only one period of the signal is of interest and this corresponds to the data obtained from an FDTD simulation). The discrete Fourier series representation of this signal is

$$g[q] = \sum_{k=\langle N_T \rangle} a_k e^{jk(2\pi/N_T)q} \tag{5.30}$$

where k is an integer index and

$$a_k = \frac{1}{N_T} \sum_{q=\langle N_T \rangle} g[q] e^{-jk(2\pi/N_T)q} \tag{5.31}$$

The summations are taken over N_T consecutive points—we do not care which ones, but in practice for our FDTD simulations a_k would be calculated with q ranging from 0 to $N_T - 1$ (the term below the summation symbols in (5.30) and (5.31) indicates the summations are done over N_T consecutive integers, but the actual start and stop points do not matter). The term a_k gives the spectral content at the frequency $f = k\Delta_f = k/(N_T\Delta_t)$. Note that a_k is obtained simply by taking the sum of the sequence $g[q]$ weighted by an exponential.

The exponential in (5.31) can be written in terms of real and imaginary parts, i.e.,

$$\Re[a_k] = \frac{1}{N_T} \sum_{q=0}^{N_T-1} g[q] \cos\left(\frac{2\pi k}{N_T} q\right), \tag{5.32}$$

$$\Im[a_k] = -\frac{1}{N_T - 1} \sum_{q=0}^{N_T} g[q] \sin\left(\frac{2\pi k}{N_T} q\right), \tag{5.33}$$

where $\Re[]$ indicates the real part and $\Im[]$ indicates the imaginary part. In practice, $g[q]$ would be a particular field component and these calculations would be performed concurrently with the time-stepping. The value of $\Re[a_k]$ would be initialized to zero. Then, at each time step $\Re[a_k]$ would be set equal to its previous value plus the current value of $g[q]\cos(2\pi kq/N_T)$, i.e., we would perform a running sum. Once the time-stepping has completed, the stored value of $\Re[a_k]$ would be divided by N_T. A similar procedure is followed for $\Im[a_k]$ where the only difference is that the value used in the running sum is $-g[q]\sin(2\pi kq/N_T)$. Note that the factor $2\pi k/N_T$ is a constant for a particular frequency, i.e., for a particular k.

Similar to the example at the end of the previous section, let us consider how we would extract information at a few specific discretizations. Assume that $N_T = 8192$ and $S_c = 1/\sqrt{2}$. Further

assume we want to find the spectral content of a particular field for discretizations of $N_\lambda = 20$, 30, 40, and 50 points per wavelength. Plugging these values into (5.27) yields corresponding spectral indices (i.e., N_{freq} or index k in (5.31)) of

$$\begin{aligned} N_\lambda = 20 &\Rightarrow k = 289.631, \\ N_\lambda = 30 &\Rightarrow k = 193.087, \\ N_\lambda = 40 &\Rightarrow k = 144.815, \\ N_\lambda = 50 &\Rightarrow k = 115.852. \end{aligned}$$

The spectral index must be an integer. Therefore, rounding k to the nearest integer and solving for the corresponding N_λ we find

$$\begin{aligned} k = 290 &\Rightarrow N_\lambda = 19.9745, \\ k = 193 &\Rightarrow N_\lambda = 30.0136, \\ k = 145 &\Rightarrow N_\lambda = 39.9491, \\ k = 116 &\Rightarrow N_\lambda = 49.9364. \end{aligned}$$

Note that these values of N_λ differ slightly from the "desired" values. Assume that we are modeling an object with some characteristic dimension of twenty cells, i.e., $a = 20\Delta_x$. Given the initial list of integer N_λ values, we could say that we were interested in finding the field at frequencies with corresponding wavelengths of a, $\frac{3}{2}a$, $2a$, and $\frac{5}{2}a$. However, using the non-integer N_λ's that are listed above, the values we obtain correspond to $0.99877a$, $1.500678a$, $1.997455a$, and $2.496818a$. As long as one is aware of what is actually being calculated, this should not be a problem. Of course, as mentioned previously, if we want to get closer to the desired values, we can interpolate between the two spectral indices which bracket the desired value.

5.5 Real Signals and DFT's

In nearly all FDTD simulations we are concerned with real signals, i.e., $g[q]$ in (5.30) and (5.31) are real. For a given spectral index k (which can be thought of as a frequency), the spectral coefficient a_k is as given in (5.31). Now, consider an index of $N_T - k$. In this case the value of a_{N_T-k} is given by

$$\begin{aligned} a_{N_T-k} = &= \frac{1}{N_T} \sum_{q=\langle N_T \rangle} g[q] e^{-j(N_T-k)(2\pi/N_T)q}, \\ &= \frac{1}{N_T} \sum_{q=\langle N_T \rangle} g[q] e^{-j2\pi} e^{jk(2\pi/N_T)q}, \\ &= \frac{1}{N_T} \sum_{q=\langle N_T \rangle} g[q] e^{jk(2\pi/N_T)q}, \\ a_{N_T-k} &= a_k^*, \end{aligned} \tag{5.34}$$

where a_k^* is the complex conjugate of a_k. Thus, for real signals, the calculation of one value of a_k actually provides the value of two coefficients: the one for index k and the one for index $N_T - k$.

Another important thing to note is that, similar to the continuous Fourier transform where there are positive and negative frequencies, with the discrete Fourier transform there are also pairs of frequencies that should typically be considered together. Let us assume that index k goes from 0 to $N_T - 1$. The $k = 0$ frequency is dc. Assuming N_T is even, $k = N_T/2$ is the Nyquist frequency where the spectral basis function $\exp(j\pi q)$ alternates between positive and negative one at each successive time-step. Both a_0 and $a_{N_T/2}$ will be real. All other spectral components can be complex, but for real signals it will always be true that $a_{N_T-k} = a_k^*$.

If we are interested in the amplitude of a spectral component at a frequency of k, we really need to consider the contribution from both k and $N_T - k$ since the oscillation at both these indices is at the same frequency. Let us assume we are interested in reconstructing just the k'th frequency of a signal. The spectral components would be given in the time domain by

$$\begin{aligned} f[q] &= a_{k'} e^{jk'(2\pi/N_T)q} + a_{N_T-k'} e^{j(N_T-k')(2\pi/N_T)q} \\ &= a_{k'} e^{jk'(2\pi/N_T)q} + a_{k'}^* e^{j2\pi} e^{-jk'(2\pi/N_T)q} \\ &= 2\Re[a_{k'}] \cos\left(\frac{2k'\pi}{N_T} q\right) - 2\Im[a_{k'}] \sin\left(\frac{2k'\pi}{N_T} q\right) \\ &= 2|a_{k'}| \cos\left(\frac{2k'\pi}{N_T} q + \tan^{-1}\left(\frac{\Im[a_{k'}]}{\Re[a_{k'}]}\right)\right) \end{aligned} \tag{5.35}$$

Thus, importantly, the amplitude of this harmonic function is given by $2|a_{k'}|$ (and not merely $|a_{k'}|$).

Let us explore this further by considering the x component of an electric field at some arbitrary point that is given by

$$\begin{aligned} \mathbf{E}[q] &= E_{x0} \cos[\omega_{k'} q + \theta_e] \hat{\mathbf{a}}_x, \\ &= \Re[E_{x0} e^{j\theta_e} e^{j\omega_{k'} q}] \hat{\mathbf{a}}_x, \\ &= \frac{1}{2}[\hat{E}_{x0} e^{j\omega_{k'} q} + \hat{E}_{x0}^* e^{-j\omega_{k'} q}] \hat{\mathbf{a}}_x, \end{aligned} \tag{5.36}$$

where

$$\omega_{k'} = k' \frac{2\pi}{N_T}, \tag{5.37}$$

$$\hat{E}_{x0} = E_{x0} e^{j\theta_e}, \tag{5.38}$$

and k' is some arbitrary integer constant. Plugging (5.36) into (5.31) and dropping the unit vector yields

$$\begin{aligned} a_k &= \frac{1}{2N_T} \sum_{q=\langle N_T \rangle} [\hat{E}_{x0} e^{j\omega_{k'} q} + \hat{E}_{x0}^* e^{-j\omega_{k'} q}] e^{-j\omega_k q}, \\ &= \frac{1}{2N_T} \sum_{q=\langle N_T \rangle} [\hat{E}_{x0} e^{j(\omega_{k'} - \omega_k) q} + \hat{E}_{x0}^* e^{-j(\omega_{k'} + \omega_k) q}], \\ &= \frac{1}{2N_T} \sum_{q=\langle N_T \rangle} [\hat{E}_{x0} e^{j(k'-k)(2\pi/N_T) q} + \hat{E}_{x0}^* e^{-j(k'+k)(2\pi/N_T) q}]. \end{aligned} \tag{5.39}$$

Without loss of generality, let us now assume q varies between 0 and $N_T - 1$ while k and k' are restricted to be between 0 and $N_T - 1$. Noting the following relationship for geometric series

$$\sum_{q=0}^{N-1} x^q = 1 + x + x^2 + \cdots + x^{N-1} = \frac{1 - x^N}{1 - x} \tag{5.40}$$

we can express the sum of the first exponential expression in (5.39) as

$$\sum_{q=0}^{N_T-1} e^{j(k'-k)(2\pi/N_T)q} = \frac{1 - e^{j(k'-k)2\pi}}{1 - e^{j(k'-k)(2\pi/N_T)}}. \tag{5.41}$$

Since k and k' are integers, the second term in the numerator on the right side, $\exp(j(k' - k)2\pi)$, equals 1 for all values of k and k' and thus this numerator is always zero. Provided $k \neq k'$, the denominator is non-zero and we conclude that for $k \neq k'$ the sum is zero. For $k = k'$, the denominator is also zero and hence the expression on the right does not provide a convenient way to determine the value of the sum. However, when $k = k'$ the terms being summed are simply 1 for all values of q. Since there are N_T terms, the sum is N_T. Given this, we can write the "orthogonality relationship" for the exponentials

$$\sum_{q=0}^{N_T-1} e^{j(k'-k)(2\pi/N_T)q} = \begin{cases} 0 & k' \neq k \\ N_T & k' = k \end{cases} \tag{5.42}$$

Using this orthogonality relationship in (5.39), we find that for the harmonic signal given in (5.36), one of the non-zero spectral coefficients is

$$a_{k'} = \frac{1}{2}\hat{E}_{x0}. \tag{5.43}$$

Conversely, expressing the (complex) amplitude in terms of the spectral coefficient from the DFT, we have

$$\hat{E}_{x0} = 2a_{k'}. \tag{5.44}$$

For the sake of completeness, now consider the sum of the other complex exponential term in (5.39). Here the orthogonality relationship is

$$\sum_{q=0}^{N_T-1} e^{-j(k'+k)(2\pi/N_T)q} = \begin{cases} 0 & k \neq N_T - k' \\ N_T & k = N_T - k' \end{cases} \tag{5.45}$$

Using this in (5.39) tells us

$$a_{N_T-k'} = \frac{1}{2}\hat{E}^*_{x0}. \tag{5.46}$$

Or, expressing the (complex) amplitude in terms of this spectral coefficient from the DFT, we have

$$\hat{E}_{x0} = 2a^*_{N_T-k'}. \tag{5.47}$$

Now, let us consider the calculation of the time-averaged Poynting vector $\mathbf{P}_{avg}$ which is given by

$$\mathbf{P}_{\text{avg}} = \frac{1}{2}\Re\left[\hat{\mathbf{E}} \times \hat{\mathbf{H}}^*\right] \tag{5.48}$$

where the carat indicates the phasor representation of the field, i.e.,

$$\begin{aligned}\hat{\mathbf{E}} &= \hat{E}_{x0}\hat{\mathbf{a}}_x + \hat{E}_{y0}\hat{\mathbf{a}}_y + \hat{E}_{z0}\hat{\mathbf{a}}_z, \\ &= 2a_{e,x,k}\hat{\mathbf{a}}_x + 2a_{e,y,k}\hat{\mathbf{a}}_y + 2a_{e,z,k}\hat{\mathbf{a}}_z \end{aligned} \tag{5.49}$$

where $a_{e,w,k}$ is the kth spectral coefficient for the w component of the electric field with $w \in \{x, y, z\}$. Defining things similarly for the magnetic field yields

$$\begin{aligned}\hat{\mathbf{H}} &= \hat{H}_{x0}\hat{\mathbf{a}}_x + \hat{H}_{y0}\hat{\mathbf{a}}_y + \hat{H}_{z0}\hat{\mathbf{a}}_z, \\ &= 2a_{h,x,k}\hat{\mathbf{a}}_x + 2a_{h,y,k}\hat{\mathbf{a}}_y + 2a_{h,z,k}\hat{\mathbf{a}}_z. \end{aligned} \tag{5.50}$$

The time-averaged Poynting vector can now be written in terms of the spectral coefficients as

$$\begin{aligned}\mathbf{P}_{avg} = 2\,\Re\Big[&\left[a_{e,y,k}a^*_{h,z,k} - a_{e,z,k}a^*_{h,y,k}\right]\hat{\mathbf{a}}_x + \left[a_{e,z,k}a^*_{h,x,k} - a_{e,x,k}a^*_{h,z,k}\right]\hat{\mathbf{a}}_y + \\ &\left[a_{e,x,k}a^*_{h,y,k} - a_{e,y,k}a^*_{h,x,k}\right]\hat{\mathbf{a}}_z\Big]. \end{aligned} \tag{5.51}$$

Thus, importantly, when directly using these DFT coefficients in the calculation of the Poynting vector, instead of the usual "one-half the real part of E cross H," the correct expression is "two times the real part of E cross H."

5.6 Amplitude and Phase from Two Time-Domain Samples

In some applications the FDTD method is used in a quasi-harmonic way, meaning the source is turned on, or ramped up, and then run continuously in a harmonic way until the simulation is terminated. Although this prevents one from obtaining broad spectrum output, for applications that have highly inhomogeneous media and where only a single frequency is of interest, the FDTD method used in a quasi-harmonic way may still be the best numerical tool to analyze the problem. The FDTD method is often used in this when in the study of the interaction of electromagnetic fields with the human body. (Tissues in the human body are typically quite dispersive. Implementing dispersive models that accurately describe this dispersion over a broad spectrum can be quite challenging. When using a quasi-harmonic approach one can simply use constant coefficients for the material constants, i.e., use the values that pertain at the frequency of interest which also corresponds to the frequency of the excitation.)

When doing a quasi-harmonic simulation, the simulation terminates when the fields have reached steady state. Steady state can be defined as the time beyond which temporal changes in the amplitude and phase of the fields are negligibly small. Thus, one needs to know the amplitude and phase of the fields at various points. Fortunately the amplitude and phase can be calculated from only two sample points of the time-domain field.

Let us assume the field at some point (x, y, z) is varying harmonically as

$$f(x, y, z, t) = A(x, y, z)\cos(\omega t + \phi(x, y, z)). \tag{5.52}$$

where A is the amplitude and ϕ is the phase, both of which are initially unknown. We will drop the explicit argument of space in the following with the understanding that we are talking about the field at a fixed point.

Consider two different samples of the field in time

$$f(t_1) = f_1 = A\cos(\omega t_1 + \phi), \tag{5.53}$$
$$f(t_2) = f_2 = A\cos(\omega t_2 + \phi). \tag{5.54}$$

We will assume that the samples are taken roughly a quarter of a cycle apart to ensure the samples are never simultaneously zero. A quarter of a cycle is given by $T/4$ where T is the period. We can express this in terms of time steps as

$$\frac{T}{4} = \frac{1}{4f} = \frac{\lambda}{4c} = \frac{N_\lambda \Delta_x}{4c}\frac{\Delta_t}{\Delta_t} = \frac{N_\lambda}{4S_c}\Delta_t. \tag{5.55}$$

Since we only allow an integer number of time-steps between t_1 and t_2, the temporal separation between t_1 and t_2 would be int$(N_\lambda/4S_c)$ time-steps.

Given these two samples, f_1 and f_2, we wish to determine A and ϕ. Using the trig identity for the cosine of the sum of two values, we can express these fields as

$$f_1 = A(\cos(\omega t_1)\cos(\phi) - \sin(\omega t_1)\sin(\phi)), \tag{5.56}$$
$$f_2 = A(\cos(\omega t_2)\cos(\phi) - \sin(\omega t_2)\sin(\phi)). \tag{5.57}$$

We are assuming that f_1 and f_2 are known and we know the times at which the samples were taken. Using (5.56) to solve for A we obtain

$$A = \frac{f_1}{\cos(\omega t_1)\cos(\phi) - \sin(\omega t_1)\sin(\phi)}. \tag{5.58}$$

Plugging (5.58) into (5.57) and rearranging terms ultimately leads to

$$\frac{\sin(\phi)}{\cos(\phi)} = \tan(\phi) = \frac{f_2\cos(\omega t_1) - f_1\cos(\omega t_2)}{f_2\sin(\omega t_1) - f_1\sin(\omega t_2)}. \tag{5.59}$$

The time at which we take the first sample, t_1, is almost always arbitrary. We are free to call that "time zero," i.e., $t_1 = 0$. By doing this we have $\cos(\omega t_1) = 1$ and $\sin(\omega t_1) = 0$. Using these values in (5.59) and (5.58) yields

$$\phi = \tan^{-1}\left(\frac{\cos(\omega t_2) - \frac{f_2}{f_1}}{\sin(\omega t_2)}\right), \tag{5.60}$$

and

$$A = \frac{f_1}{\cos(\phi)}. \tag{5.61}$$

Note that these equations assume that $f_1 \neq 0$ and, similarly, that $\phi \neq \pm\pi/2$ radians (which is the same as requiring that $\cos(\phi) \neq 0$).

If f_1 is identically zero, we can say that ϕ is $\pm\pi/2$ (where the sign can be determined by the other sample point: $\phi = -\pi/2$ if $f_2 > 0$ and $\phi = +\pi/2$ if $f_2 < 0$). If f_1 is not identically zero, we can use (5.60) to calculate the phase since the inverse tangent function has no problem with large arguments. However, if the phase is close to $\pm\pi/2$, the amplitude as calculated by (5.61) would

involve the division of two small numbers which is generally not a good way to determine a value. Rather than doing this, we can use (5.57) to express the amplitude as

$$A = \frac{f_2}{\cos(\omega t_2)\cos(\phi) - \sin(\omega t_2)\sin(\phi)}. \tag{5.62}$$

This equation should perform well when the phase is close to $\pm\pi/2$.

At this point we have the following expressions for the phase and magnitude (the reason for adding a prime to the phase and magnitude will become evident shortly):

$$\phi' = \begin{cases} -\pi/2 & \text{if } f_1 = 0 \text{ and } f_2 > 0 \\ \pi/2 & \text{if } f_1 = 0 \text{ and } f_2 < 0 \\ \tan^{-1}\left(\frac{\cos(\omega t_2) - \frac{f_2}{f_1}}{\sin(\omega t_2)}\right) & \text{otherwise} \end{cases} \tag{5.63}$$

and

$$A' = \begin{cases} \frac{f_1}{\cos(\phi')} & \text{if } |f_1| \geq |f_2| \\ \frac{f_2}{\cos(\omega t_2)\cos(\phi') - \sin(\omega t_2)\sin(\phi')} & \text{otherwise} \end{cases} \tag{5.64}$$

There is one final step in calculating the magnitude and phase. The inverse-tangent function returns values between $-\pi/2$ and $\pi/2$. As given by (5.64), the amplitude A' can be either positive or negative. Generally we think of phase as varying between $-\pi$ and π and assume the amplitude is non-negative. To obtained such values we can obtain the final values for the amplitude and phase as follows

$$(A, \phi) = \begin{cases} (A', \phi') & \text{if } A' > 0 \\ (-A', \phi' - \pi) & \text{if } A' < 0 \text{ and } \phi' \geq 0 \\ (-A', \phi' + \pi) & \text{if } A' < 0 \text{ and } \phi' < 0 \end{cases} \tag{5.65}$$

Finally, we have assumed that $t_1 = 0$. Correspondingly, that means that t_2 can be expressed as $t_2 = N_2\Delta_t$ where N_2 is the (integer) number of time steps between the samples f_1 and f_2. Thus the argument of the trig functions above can be written as

$$\omega t_2 = 2\pi S_c \frac{N_2}{N_\lambda} \tag{5.66}$$

where N_λ is the number of points per wavelength of the excitation (and frequency of interest).

5.7 Conductivity

When a material is lossless, the phase constant β for a harmonic plane wave is given by $\omega\sqrt{\mu\epsilon}$ and the spatial dependence is given by $\exp(\pm j\beta x)$. When the material has a non-zero electrical conductivity ($\sigma \neq 0$), the material is lossy and the wave experiences exponential decay as it propagates. The spatial dependence is given by $\exp(\pm\gamma x)$ where

$$\gamma = j\omega\sqrt{\mu\epsilon\left(1 - j\frac{\sigma}{\omega\epsilon}\right)} = \alpha + j\beta \tag{5.67}$$

where α (the real part of γ) is the attenuation constant and β (the imaginary part of γ) is the phase constant. The attenuation and phase constants can be expressed directly in terms of the material parameters and the frequency:

$$\alpha = \frac{\omega\sqrt{\mu\epsilon}}{\sqrt{2}}\left(\left[1+\left(\frac{\sigma}{\omega\epsilon}\right)^2\right]^{1/2}-1\right)^{1/2}, \tag{5.68}$$

$$\beta = \frac{\omega\sqrt{\mu\epsilon}}{\sqrt{2}}\left(\left[1+\left(\frac{\sigma}{\omega\epsilon}\right)^2\right]^{1/2}+1\right)^{1/2}. \tag{5.69}$$

When the conductivity is zero, the attenuation constant is zero and the phase constant reduces to that of the lossless case, i.e., $\gamma = j\beta = j\omega\sqrt{\mu\epsilon}$.

Assume a wave is propagating in the positive x direction in a material with non-zero electrical conductivity. The wave amplitude will decay as $\exp(-\alpha x)$. The skin depth δ_{skin} is the distance over which the wave decays an amount $1/e$. Starting with a reference point of $x = 0$, the fields would have decayed an amount $1/e$ when x is $1/\alpha$ (so that the exponent is simply -1). Thus, δ_{skin} is given by

$$\delta_{\text{skin}} = \frac{1}{\alpha}. \tag{5.70}$$

Since the skin depth is merely a distance, it can be expressed in terms of the spatial step, i.e.,

$$\delta_{\text{skin}} = \frac{1}{\alpha} = N_L\Delta_x \tag{5.71}$$

where N_L is the number of spatial steps in the skin depth (think of the subscript L as standing for loss). N_L does not need to be an integer.

It is possible to use (5.68) to solve for the conductivity in terms of the attenuation constant. The resulting expression is

$$\sigma = \omega\epsilon\left(\left[1+\frac{2\alpha^2}{\omega^2\mu\epsilon}\right]^2-1\right)^{1/2}. \tag{5.72}$$

As shown in Sec. 3.12, when the electrical conductivity is non-zero the electric-field update equation contains the term $\sigma\Delta_t/2\epsilon$. Multiplying both side of (5.72) by $\Delta_t/2\epsilon$ yields

$$\frac{\sigma\Delta_t}{2\epsilon} = \frac{\omega\Delta_t}{2}\left(\left[1+\frac{2\alpha^2}{\omega^2\mu\epsilon}\right]^2-1\right)^{1/2}. \tag{5.73}$$

Assume that one wants to obtain a certain skin depth (or decay rate) at a particular frequency which is discretized with N_λ points per wavelength, i.e., $\omega = 2\pi f = 2\pi c/N_\lambda\Delta_x$. Thus the term $\omega\Delta_t/2$ can be rewritten

$$\frac{\omega\Delta_t}{2} = \frac{\pi}{N_\lambda}\frac{c\Delta_t}{\Delta_x} = \frac{\pi}{N_\lambda}S_c. \tag{5.74}$$

Similarly, using the same expression for ω and using $\alpha = 1/N_L\Delta_x$, one can write

$$\frac{2\alpha^2}{\omega^2\mu\epsilon} = \frac{2\left(\frac{1}{N_L\Delta_x}\right)^2}{\left(\frac{2\pi c}{N_\lambda\Delta_x}\right)^2\mu_0\mu_r\epsilon_0\epsilon_r} = \frac{N_\lambda^2}{2\pi^2N_L^2\epsilon_r\mu_r}. \tag{5.75}$$

Using (5.74) and (5.75) in (5.73) yields

$$\frac{\sigma\Delta_t}{2\epsilon} = \frac{\pi}{N_\lambda} S_c \left(\left[1 + \frac{N_\lambda^2}{2\pi^2 N_L^2 \epsilon_r \mu_r} \right]^2 - 1 \right)^{1/2}. \tag{5.76}$$

Note that neither the temporal nor the spatial steps appear in the right-hand side.

As an example how (5.76) can be used, assume that one wants a skin depth of $20\Delta_x$ for a wavelength of $40\Delta_x$. Thus $N_L = 20$ and $N_\lambda = 40$ and the skin depth is one half of the free-space wavelength. Further assume the Courant number S_c is unity, $\epsilon_r = 4$, and $\mu_r = 1$. Plugging these values into (5.76) yields $\sigma\Delta_t/2\epsilon = 0.0253146$.

Let us write a program where a TFSF boundary introduces a sine wave with a frequency that is discretized at 40 points per wavelength. We will only implement electrical loss (the magnetic conductivity is zero). Snapshots will be taken every time step after the temporal index is within 40 steps from the final time step. The lossy layer starts at node 100. To implement this program, we can re-use nearly all the code that was described in Sec. 4.9. To implement this, we merely have to change the `gridInit3()` function and the functions associated with the source function. The new `gridInit3()` function is shown in Program 5.1. The harmonic source function is given by the code presented in Program 5.2.

Program 5.1 `gridinitlossy.c` A `Grid` initialization function for modeling a lossy half space. Here the conductivity results in a skin depth of 20 cells for an excitation that is discretized using 40 cells per wavelength.

```
1  #include "fdtd3.h"
2
3  #define LOSS 0.0253146
4  #define LOSS_LAYER 100
5  #define EPSR 4.0
6
7  void gridInit3(Grid *g) {
8    double imp0 = 377.0;
9    int mm;
10
11   SizeX = 200;   // size of domain
12   MaxTime = 450; // duration of simulation
13   Cdtds = 1.0;   // Courant number
14
15   /* Allocate memory for arrays. */
16   ALLOC_1D(g->ez,   SizeX, double);
17   ALLOC_1D(g->ceze, SizeX, double);
18   ALLOC_1D(g->cezh, SizeX, double);
19   ALLOC_1D(g->hy,   SizeX - 1, double);
20   ALLOC_1D(g->chyh, SizeX - 1, double);
21   ALLOC_1D(g->chye, SizeX - 1, double);
22
```

```
23    /* set electric-field update coefficients */
24    for (mm=0; mm < SizeX; mm++)
25      if (mm < 100) {
26        Ceze(mm) = 1.0;
27        Cezh(mm) = imp0;
28      } else {
29        Ceze(mm) = (1.0 - LOSS) / (1.0 + LOSS);
30        Cezh(mm) = imp0 / EPSR / (1.0 + LOSS);
31      }
32
33    /* set magnetic-field update coefficients */
34    for (mm=0; mm < SizeX - 1; mm++) {
35      Chyh(mm) = 1.0;
36      Chye(mm) = 1.0 / imp0;
37    }
38
39    return;
40  }
```

Program 5.2 `ezincharm.c` Functions to implement a harmonic source. When initialized, the user is prompted to enter the number of points per wavelength (in the results to follow it is assumed the user enters 40).

```
1   #include "ezinc3.h"
2
3   /* global variables -- but private to this file */
4   static double ppw = 0, cdtds;
5
6   /* prompt user for source-function points per wavelength */
7   void ezIncInit(Grid *g){
8
9     cdtds = Cdtds;
10    printf("Enter points per wavelength: ");
11    scanf(" %lf", &ppw);
12
13    return;
14  }
15
16  /* calculate source function at given time and location */
17  double ezInc(double time, double location) {
18    if (ppw <= 0) {
19      fprintf(stderr,
20         "ezInc: must call ezIncInit before ezInc.\n"
21         "       Points per wavelength must be positive.\n");
```

```
22      exit(-1);
23    }
24
25    return sin(2.0 * M_PI / ppw * (cdtds * time - location));
26  }
```

Assuming the user specified that the TFSF boundary should be at node 50 and there should be 40 points per wavelength for the harmonic source, Fig. 5.2(a) shows the resulting maximum of the magnitude of the electric field as a function of position. For each position, all the snapshots were inspected and the maximum recorded. Figure 5.2(b) shows a superposition of 41 snapshots taken one time-step apart. One can see the exponential decay starting at node 100. Between node 50 and node 100 there is a standing-wave pattern caused by the interference of the incident and reflected waves. From the start of the grid to node 50 the magnitude is flat. This is caused by the fact that there is only scattered field here—there is nothing to interfere with the reflected wave and we see the constant amplitude associated with a pure traveling wave. The ratio of the amplitude at nodes 120 and 100 was found to be 0.3644 whereas the ideal value of $1/e$ is 0.3679 (thus there is approximately a one percent error in this simulation).

If one were interested in non-zero magnetic conductivity σ_m, the loss term which appears in the magnetic-field update equations is $\sigma_m \Delta_t / 2\mu$. This term can be handled in exactly the same way as the term resulting from electric conductivity.

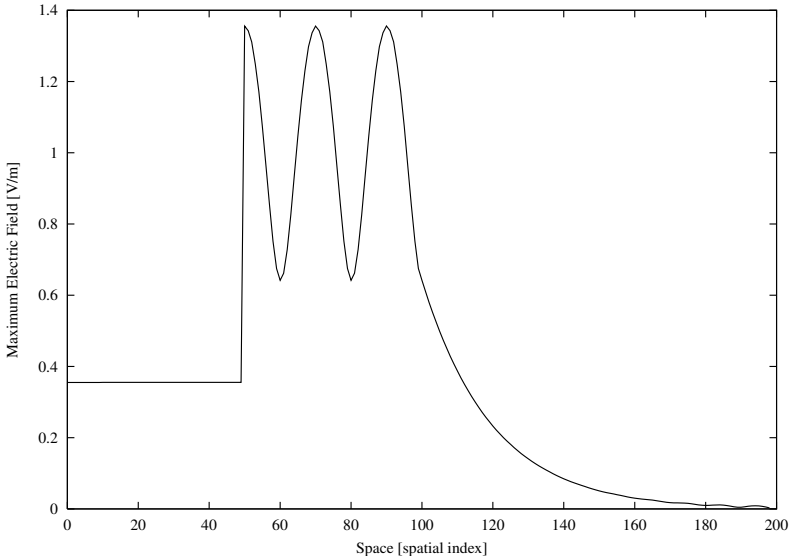
5.8 Example: Obtaining the Transmission Coefficient for a Planar Interface

In this last section, we demonstrate how the transmission coefficient for a planar dielectric boundary can be obtained from the FDTD method. This serves to highlight many of the points that were considered in the previous section. We will compare the results to the exact solution. The disparity between the two provides motivation to determine the dispersion relation in the FDTD (which is covered in Chap. 7).

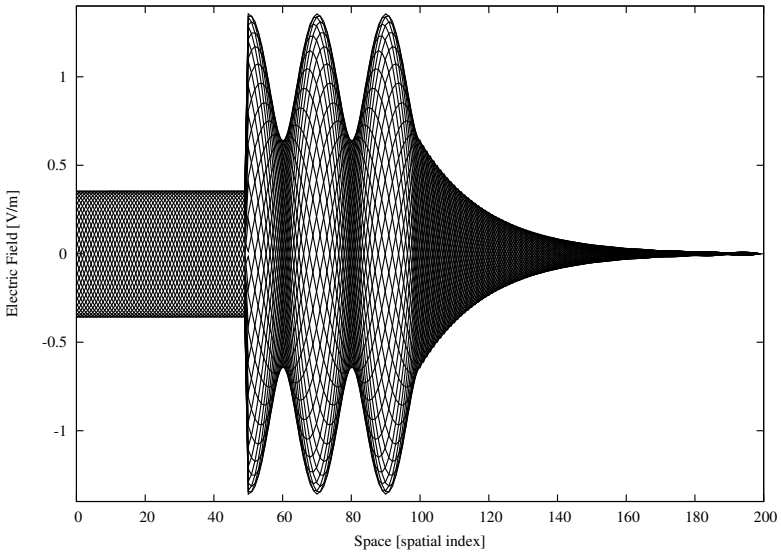
The majority of instructional material concerning electromagnetics is expressed in terms of harmonic, or frequency-domain, signals. A temporal dependence of $\exp(j\omega t)$ is understood and therefore one only has to consider the spatial variation. In the frequency domain, the fields and quantities such as the propagation constant and the characteristic impedance are represented by complex numbers. These complex numbers give the magnitude and phase of the value and will be functions of frequency. In the following discussion a caret (hat) will be used to indicate a complex quantity and one should keep in mind that complex numbers are inherently tied to the frequency domain. Given the frequency-domain representation of the field at a point, the temporal signal is recovered by multiplying by $\exp(j\omega t)$ and taking the real part. Thus a 1D harmonic field propagating in the $+x$ direction could be written in any of these equivalent forms

$$E_z(x,t) = \Re\left[\hat{E}_z^+(x,t)\right] = \Re\left[\hat{E}_z^+(x)e^{j\omega t}\right] = \Re\left[\hat{E}_0^+ e^{-\hat{\gamma}x} e^{j\omega t}\right] = \Re\left[\hat{E}_0^+ e^{-(\alpha+j\beta)x} e^{j\omega t}\right], \quad (5.77)$$

where $\Re[]$ indicates the real part. $\hat{E}_z^+(x)$ is the frequency-domain representation of the field (i.e., a phasor that is a function of position), $\hat{\gamma}$ is the propagation constant which has a real part α and



(a)



(b)

Figure 5.2: (a) Maximum electric field magnitude that exists at each point (obtained by obtaining the maximum value in all the snapshots). The flat line over the first 50 nodes corresponds to the scattered-field region. The reflected field travels without decay and hence produces the flat line. The total-field region between nodes 50 and 100 contains a standing-wave pattern caused by the interference of the incident and scattered fields. There is exponential decay of the fields beyond node 100 which is where the lossy layer starts. (b) Superposition of 41 individual snapshots of the field which illustrates the envelope of the field.

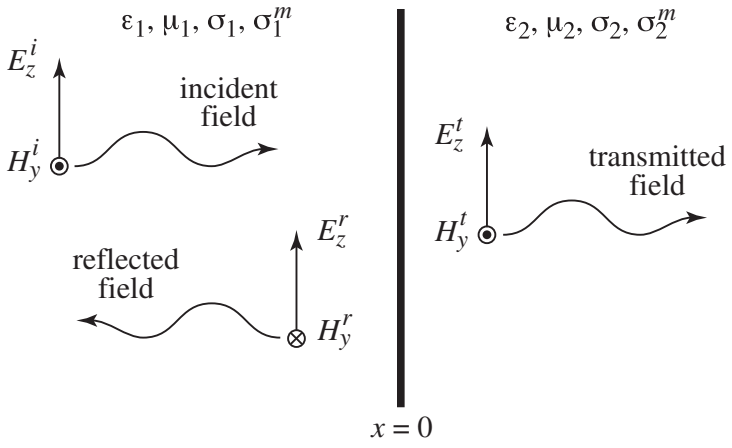


Figure 5.3: Planar interface between two media. The interface is at $x = 0$ and a wave is incident on the interface from the left. When the impedances of the two media are not matched, a reflected wave must exist in order to satisfy the boundary conditions.

an imaginary part β, and $\hat{E}_0^+$ is a complex constant that gives the amplitude of the wave (it is independent of position; the superscript "+" is merely used to emphasize that we are discussing a wave propagating in the $+x$ direction). Note that if more than a single frequency is present, $\hat{E}_0^+$ does not need to be the same for each frequency.

More generally, $\hat{E}_0^+$ will be a function of frequency (we could write this expressly as $\hat{E}_0^+(\omega)$ but the caret implicitly indicates dependence on frequency). To construct a temporal that consists of a multiple frequencies, or even a continuous spectrum of frequencies, one must sum the contributions from each frequency such as is done with a Fourier integral.

In FDTD simulations the time-domain form of the signal is obtained directly. However, one often is interested in the behavior of the fields as a function of frequency. As has been discussed in Sec. 5.3, one merely has to take the Fourier transform of a signal to obtain its spectral content. This transform, by itself, is typically of little use. One has to normalize the signal in some way. Knowing what came out of a system is rather meaningless unless one knows what went into the system. Here we will work through a couple of simple examples to illustrate how a broad range of spectral information can be obtained from FDTD simulations.

5.8.1 Transmission through a Planar Interface (Continuous World)

Consider the planar interface at $x = 0$ between two media as depicted in Fig. 5.3. We restrict consideration to electric polarization in the z direction and assume the incident field originates in the first medium. In the frequency domain, the incident, reflected, and transmitted fields are given by

$$\hat{E}_z^i(x) = \hat{E}_{a1}^+ e^{-\hat{\gamma}_1 x} \quad \text{incident,} \tag{5.78}$$

$$\hat{E}_z^r(x) = \hat{E}_{a1}^- e^{+\hat{\gamma}_1 x} = \hat{\Gamma}\hat{E}_{a1}^+ e^{+\hat{\gamma}_1 x} \quad \text{reflected,} \tag{5.79}$$

$$\hat{E}_z^t(x) = \hat{E}_{a2}^+ e^{-\hat{\gamma}_2 x} = \hat{T}\hat{E}_{a1}^+ e^{-\hat{\gamma}_2 x} \quad \text{transmitted,} \tag{5.80}$$

where $\hat{\Gamma}$ is the reflection coefficient, $\hat{T}$ is the transmission coefficient, and $\hat{\gamma}_n$ is the propagation constant given by $j\omega\sqrt{\mu_n(1-j\sigma_n^m/\omega\mu_n)\epsilon_n(1-j\sigma_n/\omega\epsilon_n)}$ where the n indicates the medium and σ^m is the magnetic conductivity. The amplitude of the incident field $\hat{E}_{a1}^+$ is, in general, complex and a function of frequency. By definition $\hat{\Gamma}$ and $\hat{T}$ are

$$\hat{\Gamma} = \left.\frac{\hat{E}_z^r(x)}{\hat{E}_z^i(x)}\right|_{x=0} = \frac{\hat{E}_{a1}^-}{\hat{E}_{a1}^+}, \tag{5.81}$$

$$\hat{T} = \left.\frac{\hat{E}_z^t(x)}{\hat{E}_z^i(x)}\right|_{x=0} = \frac{\hat{E}_{a2}^+}{\hat{E}_{a1}^+}. \tag{5.82}$$

The fact that these are defined at $x = 0$ is important.

The magnetic field is related to the electric field by

$$\hat{H}_y^i(x) = -\frac{1}{\hat{\eta}_1}\hat{E}_z^i(x), \tag{5.83}$$

$$\hat{H}_y^r(x) = \frac{1}{\hat{\eta}_1}\hat{E}_z^r(x), \tag{5.84}$$

$$\hat{H}_y^t(x) = -\frac{1}{\hat{\eta}_2}\hat{E}_z^t(x), \tag{5.85}$$

where the characteristic impedance $\hat{\eta}_n$ is given by $\sqrt{\mu_n(1-j\sigma_n^m/\omega\mu_n)/\epsilon_n(1-j\sigma_n/\omega\epsilon_n)}$. Since the magnetic and electric fields are purely tangential to the planar interface, the sum of the incident and reflected field at $x = 0$ must equal the transmitted field at the same point. Matching the boundary condition on the electric field yields

$$1 + \hat{\Gamma} = \hat{T}, \tag{5.86}$$

while matching the boundary conditions on the magnetic field produces

$$\frac{1}{\hat{\eta}_1}(1 - \hat{\Gamma}) = \frac{1}{\hat{\eta}_2}\hat{T}. \tag{5.87}$$

Solving these for $\hat{T}$ yields

$$\hat{T} = \frac{2\hat{\eta}_2}{\hat{\eta}_1 + \hat{\eta}_2}. \tag{5.88}$$

Using this in (5.86) the reflection coefficient is found to be

$$\hat{\Gamma} = \frac{\hat{\eta}_2 - \hat{\eta}_1}{\hat{\eta}_2 + \hat{\eta}_1}. \tag{5.89}$$

5.8.2 Measuring the Transmission Coefficient Using FDTD

Now consider two FDTD simulations. The first simulation will be used to record the incident field. In this simulation the computational domain is homogeneous and the material properties correspond to that of the first medium. The field is recorded at some observation point x_1. Since nothing is present to interfere with the incident field, the recorded field will be simply the incident

field at this location, i.e, $E_z^i(x_1,t)$. In the second simulation, the second medium is present. We record the fields at the same observation point but we ensure the point was chosen such that it is located in the second medium (i.e., the interface is to the left of the observation point). Performing an FDTD simulation in this case yields the transmitted field $E_z^t(x_1,t)$. The goal now is to obtain the transmission coefficient using the temporal recordings of the field obtained from these two simulations.

Note that in this section we will not distinguish between the way in which field propagate in the continuous world and the way in which they propagate in the FDTD grid. In Chap. 7 we will discuss in some detail how these differ.

One cannot use $E_z^t(x_1,t)/E_z^i(x_1,t)$ to obtain the transmission coefficient. The transmission coefficient is inherently a frequency-domain concept and currently we have time-domain signals. The division of these temporal signals is essentially meaningless (e.g., the result is undefined when the incident signal is zero).

The incident and transmitted fields must be converted to the frequency domain using a Fourier transform. Thus one obtains

$$\hat{E}_z^i(x_1) = \mathcal{F}\left(E_z^i(x_1,t)\right), \tag{5.90}$$

$$\hat{E}_z^t(x_1) = \mathcal{F}\left(E_z^t(x_1,t)\right), \tag{5.91}$$

where $\mathcal{F}$ indicates the Fourier transform. The division of these two functions *is* meaningful—at least at all frequencies where $\hat{E}_z^i(x_1)$ is non-zero. (At frequencies where $\hat{E}_z^i(x_1)$ is zero, there is no incident spectral energy and hence one cannot obtain the transmitted field at those particular frequencies. In practice it is relatively easy to introduce energy into an FDTD grid that spans a broad range of frequencies.)

Since the observation point was not specified to be on the boundary, the ratio of these field fields is

$$\frac{\hat{E}_z^t(x_1)}{\hat{E}_z^i(x_1)} = \frac{\hat{E}_{a2}^+ e^{-\hat{\gamma}_2 x_1}}{\hat{E}_{a1}^+ e^{-\hat{\gamma}_1 x_1}} = \frac{\hat{E}_{a2}^+}{\hat{E}_{a1}^+} e^{(\hat{\gamma}_1-\hat{\gamma}_2)x_1} = \hat{T} e^{(\hat{\gamma}_1-\hat{\gamma}_2)x_1}. \tag{5.92}$$

Solving this for $\hat{T}$ yields

$$\hat{T}(\omega) = e^{(\hat{\gamma}_2-\hat{\gamma}_1)x_1} \frac{\hat{E}_z^t(x_1)}{\hat{E}_z^i(x_1)}. \tag{5.93}$$

To demonstrate how the transmission coefficient can be reconstructed from FDTD simulations, let us consider an example where the first medium is free space and the second one has a relative permittivity ϵ_r of 9. In this case $\hat{\gamma}_1 = j\omega\sqrt{\mu_0\epsilon_0} = j\beta_0$ and $\hat{\gamma}_2 = j\omega\sqrt{\mu_0 9\epsilon_0} = j3\beta_0$. Therefore (5.93) becomes

$$\hat{T}(\omega) = e^{j(3\beta_0-\beta_0)x_1} \frac{\hat{E}_z^t(x_1)}{\hat{E}_z^i(x_1)} = e^{j2\beta_0 x_1} \frac{\hat{E}_z^t(x_1)}{\hat{E}_z^i(x_1)}. \tag{5.94}$$

The terms in the exponent can be written

$$2\beta_0 x_1 = 2\frac{2\pi}{\lambda}x_1 = \frac{4\pi}{N_\lambda \Delta_x} N_1 \Delta_x = \frac{4\pi}{N_\lambda} N_1 \tag{5.95}$$

where N_1 is the number of spatial steps between the interface and the observation point at x_1 and, as was discussed in Chap. 5, N_λ is the number of spatial steps per a free-space wavelength of λ.

The continuous-world transmission coefficient can be calculated quite easily from (5.88) and this provides a reference solution. Ideally the FDTD simulation would yield this same value for all frequencies. For this particular example the characteristic impedance of the first medium is $\hat{\eta}_1 = \eta_0$ while for the second medium it is $\hat{\eta}_2 = \eta_0/3$. Thus the transmission coefficient is

$$\hat{T}_{\text{exact}} = \frac{2\eta_0/3}{\eta_0 + \eta_0/3} = 1/2. \tag{5.96}$$

Note that this is a real number and independent of frequency (so the tilde on T is somewhat misleading).

For the FDTD simulations, let us record the field 80 spatial steps away from the interface, i.e., $N_1 = 80$, and run the simulation for 8192 time steps, i.e., $N_T = 8192$. The simulation is run at the Courant limit $S_c = 1$. The source is a Ricker wavelet discretized so that the peak spectral content exists at 50 points per wavelength ($N_P = 50$). From Sec. 5.3, recall the relationship between the points per wavelength N_λ and frequency index N_{freq} which is repeated below:

$$N_{\text{freq}} = \frac{N_T}{N_\lambda} S_c. \tag{5.97}$$

With a Courant number of unity and 8192 time steps, the points per wavelength for any given frequency (or spectral index) is given by

$$N_\lambda = \frac{8192}{N_{\text{freq}}}. \tag{5.98}$$

Combining this with (5.93) and (5.95) yields

$$\hat{T}_{\text{FDTD}} = e^{j\left(\frac{4\pi N_1 N_{\text{freq}}}{8192}\right)} \frac{\hat{E}_z^t(x_1)}{\hat{E}_z^i(x_1)}. \tag{5.99}$$

Ideally (5.96) and (5.99) will agree at all frequencies. To see if that is the case, Fig. 5.4 shows three plots related to the incident and transmitted fields. Figure 5.4(a) shows the first 500 time steps of the temporal signals recorded at the observation point both with and without the interface present (i.e., the transmitted and incident fields, respectively). Figure 5.4(b) shows the magnitude of the Fourier transforms of the incident and transmitted fields for the first 500 frequencies. Since a Ricker wavelet was used, the spectra are essentially in accordance with the discussion of Sec. 5.2.3. There is no spectral energy at dc and the spectral content exponentially approaches zero at high frequencies.

Figure 5.4(c) plots the magnitude of the ratio of the transmitted and incident field as a function of frequency. Ideally this would be $1/2$ for all frequencies. Note the rather small vertical scale of the plot. Near dc the normalized transmitted field differs rather significantly from the ideal value, but this is in a region where the results should not be trusted because there is not enough incident energy at these frequencies. At the higher frequencies some oscillations are present. The normalized field generally remains within two percent of the ideal value over this range of frequencies.

Figure 5.5(a) provides the same information as Fig. 5.4(c) except now the result is plotted versus the discretization N_λ. In this figure dc is off the scale to the right (since in theory dc has an infinite number of points per wavelength). As the frequency goes up, the wavelength gets shorter

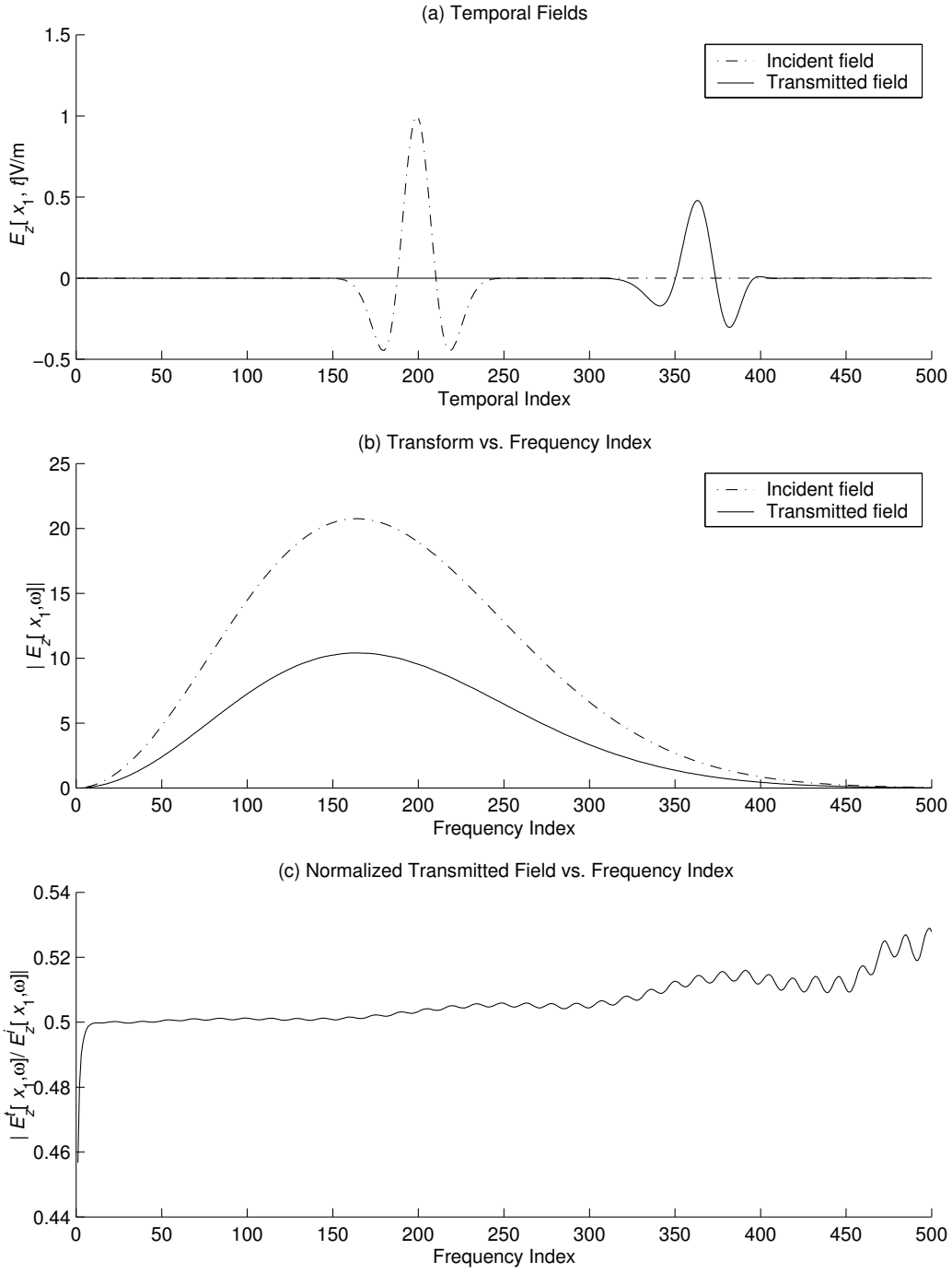


Figure 5.4: (a) Time-domain fields at the observation point both with and the without the interface present. The field without the interface is the incident field and the one with the interface is the transmitted field. (b) Magnitude of the Fourier transforms of the incident and transmitted fields. The transforms are plotted versus the frequency index N_{freq}. It can be seen that the fields do not have much spectral content near dc nor at high frequencies. (c) Magnitude of the transmitted field normalized by the incident field versus the frequency index. Ideally this would be $1/2$ for all frequencies. Errors are clearly evident when the spectral content of the incident field is small.

and hence the number of points per wavelength decreases. Thus high frequencies are to the left and low frequencies are to the right. The highest frequency in this plot corresponds to an N_{freq} of 500. In terms of the discretization, this is $N_\lambda = N_T/N_{\text{freq}} = 16.384$. This may not seem like a particularly coarse discretization, but one needs to keep in mind that this is the discretization in frees pace. Within the dielectric, which here has $\epsilon_r = 9$, the wavelength is three times smaller and hence within the dielectric the fields are only discretized at approximately five points per wavelength (which is considered a very coarse discretization). From this figure it is clear that the FDTD simulations provide results which are close to the ideal over a fairly broad range of frequencies.

Figure 5.5(b) shows the real and imaginary part of the reflection coefficient, i.e., $\hat{T}_{\text{FDTD}}$ defined in (5.99), as a function of the discretization. Ideally the imaginary part would be zero and the real part would be $1/2$. As can be seen, although the magnitude of the transmission coefficient is nearly $1/2$ over the entire spectrum, the phase differs rather significantly as the discretization decreases (i.e., the frequency increases).

The Matlab code used to generate Fig. 5.4 is shown in Program 5.3 while the code which generated Fig. 5.5 is given in Program 5.4. It is assumed the incident field from the FDTD simulation is recorded to a file named `inc-8192` while the transmitted field, i.e., the field when the dielectric is present, is recorded in `die-8192`. The code in Program 5.3 has to be run prior to that of 5.4 in order to load and initialize the data.

Let us now consider the same scenario but let the observation point be four steps away from the boundary instead of 80, i.e., $N_1 = 4$. Following the previous steps, the incident and transmitted fields are recorded, their transforms are taken, then divided, and finally the phase is adjusted to obtain the transmission coefficient. The result for this observation point is shown in Fig. 5.6. The real and imaginary parts stay closer to the ideal values over a larger range of frequencies than when the observation point was 80 cells from the boundary. The fact that the quality of the results are frequency sensitive as well as sensitive to the observation point is a consequence of numeric dispersion in the FDTD grid, i.e., different frequencies propagate at different speeds. (This is the subject of Chap. 7.)

Program 5.3 Matlab session used to generate Fig. 5.4.

```
incTime = dlmread('inc-8192'); % incident field file
dieTime = dlmread('die-8192'); % transmitted field file

inc = fft(incTime);  % take Fourier transforms
die = fft(dieTime);

nSteps = length(incTime);  % number of time steps
freqMin = 1;    % minimun frequency index of interest
freqMax = 500;  % maximum frequency of interest
freqIndex = freqMin:freqMax; % range of frequencies of interest
% correct for offset of 1 in matlab's indexing
freqSlice = freqIndex + 1;
courantNumber = 1;
% points per wavelength for freqencies of interest
```

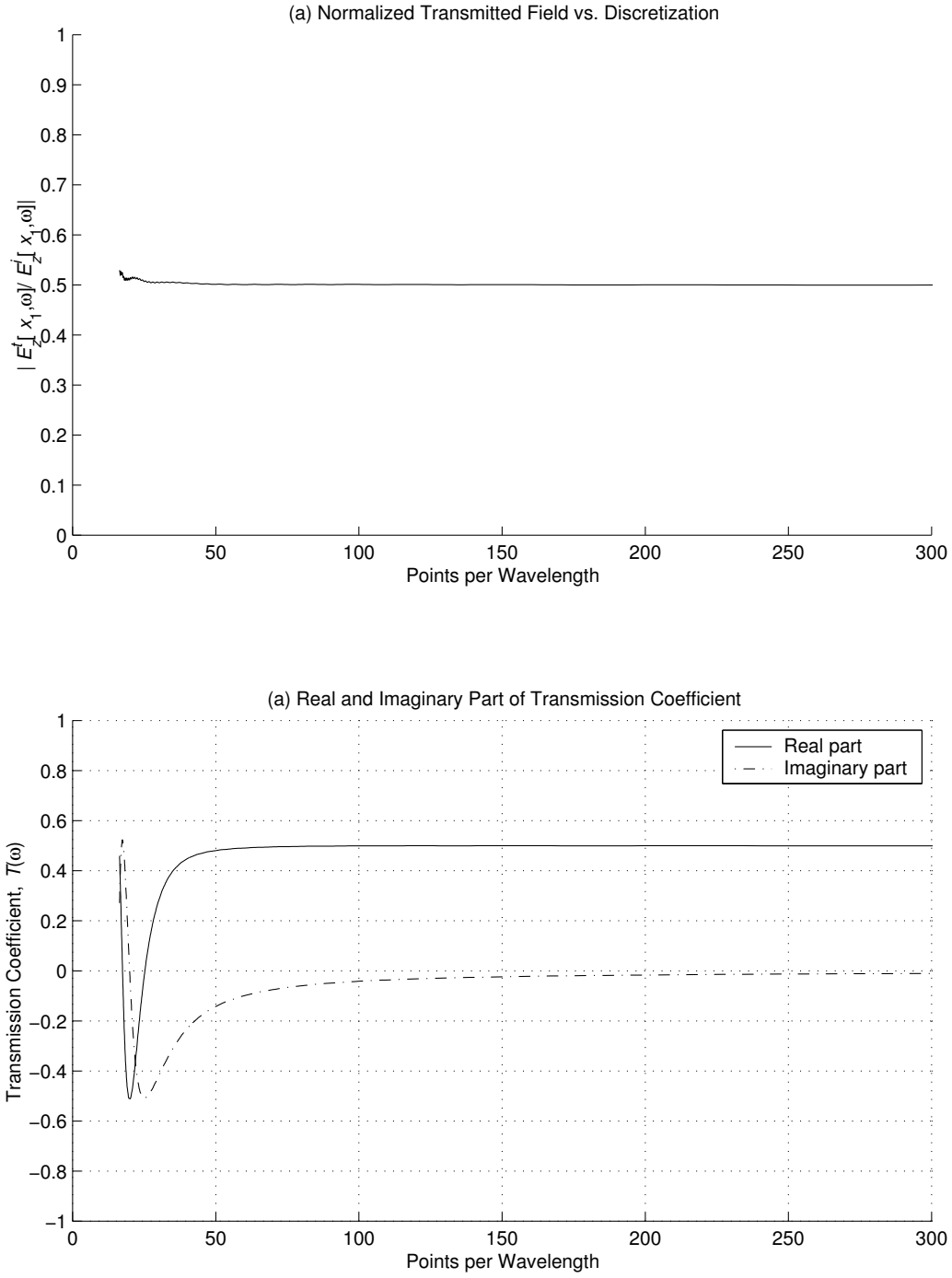


Figure 5.5: (a) The magnitude of the normalized transmitted field as a function of the (free space) discretization N_λ. Ideally this would be $1/2$ for all discretizations. (b) Real and imaginary part of the transmission coefficient transformed back to the interface $x = 0$ versus discretization. Ideally the real part would be $1/2$ and the imaginary part would be zero for all discretization. For these plots the observation point was 80 cells from the interface ($N_1 = 80$).

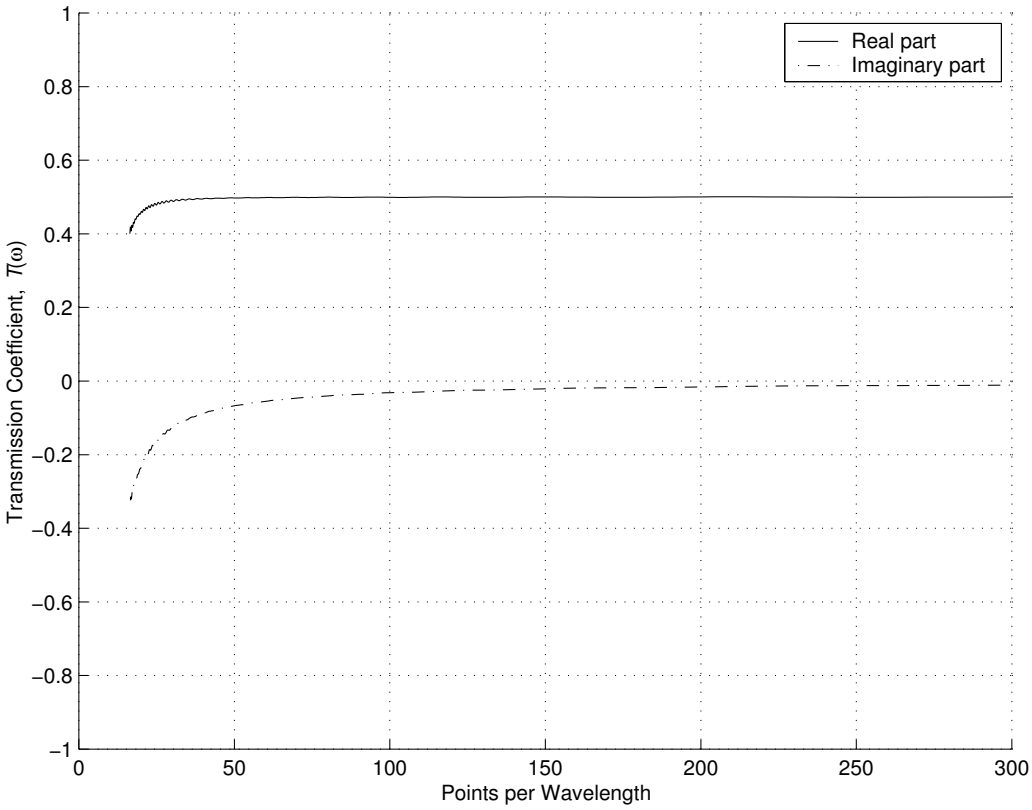


Figure 5.6: Real and imaginary part of the transmission coefficient transformed back to the interface $x = 0$ versus discretization. Ideally the real part would be $1/2$ and the imaginary part would be zero for all discretization. The observation point was four cells from the interface ($N_1 = 4$).

```
nLambda = nSteps ./ freqIndex * courantNumber;
clf
subplot(3, 1, 1)
hold on
plot(incTime(freqSlice), '-.');
plot(dieTime(freqSlice));
legend('Incident field', 'Transmitted field');
xlabel('Temporal Index');
ylabel('{\it E_z}[{\it x}_1,{\it t}] V/m');
title('(a) Temporal Fields');

subplot(3,1,2)
hold on
plot(freqIndex, abs(inc(freqSlice)), '-.');
plot(freqIndex, abs(die(freqSlice)));
legend('Incident field', 'Transmitted field');
xlabel('Frequency Index');
ylabel('|{\it E_z}[{\it x}_1,\omega]|');
title('(b) Transform vs. Frequency Index');
hold off

subplot(3, 1, 3)
hold on
plot(freqIndex, abs(die(freqSlice) ./ inc(freqSlice)));
xlabel('Frequency Index');
ylabel('|{\it E^t_z}[{\it x}_1,\omega]/...
                          {\it E^i_z}[{\it x}_1,\omega]|');
title('(c) Normalized Transmitted Field vs. Frequency Index');
hold off
```

Program 5.4 Matlab session used to generate Fig. 5.5. The commands shown in Program 5.3 would have to be run prior to these commands in order to read the data, generate the Fourier transforms, etc.

```
clf

subplot(2, 1, 1)
hold on
plot(nLambda, abs(die(freqSlice) ./ inc(freqSlice)));
xlabel('Points per Wavelength');
ylabel('|{\it E^t_z}[{\it x}_1,\omega]/...
                          {\it E^i_z}[{\it x}_1,\omega]|');
title('(a) Normalized Transmitted Field vs. Discretization');
axis([0 300 0 1])
```

```
hold off

% Array obtained from exp() must be transposed to make arrays
% conformal.  Simply using ' (a prime) for trasposition will yield
% the conjugate transpose.  Instead, use .' (dot-prime) to get
% transposition without conjugation.
subplot(2, 1, 2)
hold on
plot(nLambda, real(exp(j*pi*freqIndex/25.6).' .* ...
                   die(freqSlice) ./ inc(freqSlice)));
plot(nLambda, imag(exp(j*pi*freqIndex/25.6).' .* ...
                   die(freqSlice) ./ inc(freqSlice)), '-.');
xlabel('Points per Wavelength');
ylabel('Transmission Coefficient, {\it T}(\omega)');
title('(b) Real and Imaginary Part of Transmission Coefficient');
legend('Real part', 'Imaginary part');
axis([0 300 -1 1])
grid on
hold off
```

Although we have only considered the transmission coefficient in this example, the reflection coefficient could be obtained in a similar fashion. We have intentionally considered a very simple problem in order to be able to compare easily the FDTD solution to the exact solution. However one should keep in mind that the FDTD method could be used to analyze the reflection or transmission coefficient for a much more complicated scenario, e.g., one in which the material properties varied continuously, and perhaps quite erratically (with some discontinuities present), over the transition from one half-space to the next. Provided a sufficiently small spatial step-size was used, the FDTD method can solve this problem with essentially no more effort than was used to model the abrupt interface. However, to obtain the exact solution, one may have to work much harder.

Chapter 6

Differential-Equation Based Absorbing Boundary Conditions

6.1 Introduction

A simple absorbing boundary condition (ABC) was used in Chap. 3 to terminate the grid. It relied upon the fact that the fields were propagating in one dimension and the speed of propagation was such that the fields moved one spatial step for every time step, i.e., the Courant number was unity. The node on the boundary was updated using the value of the adjacent interior node from the previous time step. However, when a dielectric was introduced, and the local speed of propagation was no longer equal to c, this ABC ceased to work properly. One would also find in higher dimensions that this simple ABC would not work even in free space. This is because the Courant number cannot be unity in higher dimensions and this ABC does not account for fields which may be obliquely incident on the edge of the grid. The goal now is to find a more general technique to terminate the grid.

Although the ABC we will discuss here is not considered state-of-the-art, it provides a relatively simple way to terminate the grid that is more than adequate in many circumstances. Additionally, some of the mathematical tools we will develop in this chapter can be used in the analysis of a wide range of FDTD-related topics.

6.2 The Advection Equation

The wave equation that governs the propagation of the electric field in one dimension is

$$\frac{\partial^2 E_z}{\partial x^2} - \mu\epsilon \frac{\partial^2 E_z}{\partial t^2} = 0, \tag{6.1}$$

$$\left(\frac{\partial^2}{\partial x^2} - \mu\epsilon \frac{\partial^2}{\partial t^2} \right) E_z = 0. \tag{6.2}$$

The second form represents the equation in terms of an operator operating on E_z where the operator is enclosed in parentheses. This operator can be factored into the product of two operators and is

Lecture notes by John Schneider. `fdtd-abc.tex`

equivalent to

$$\left(\frac{\partial}{\partial x}-\sqrt{\mu\epsilon}\frac{\partial}{\partial t}\right)\left(\frac{\partial}{\partial x}+\sqrt{\mu\epsilon}\frac{\partial}{\partial t}\right)E_z=0. \tag{6.3}$$

Note that it does not matter which operator in (6.3) is written first. They commute and will always ultimately yield (6.1). If either of these operators acting individually on the field yields zero, the wave equation is automatically satisfied. Thus an E_z that satisfies either of the following equations will also be a solution to the wave equation:

$$\frac{\partial E_z}{\partial x}-\sqrt{\mu\epsilon}\frac{\partial E_z}{\partial t} = 0, \tag{6.4}$$

$$\frac{\partial E_z}{\partial x}+\sqrt{\mu\epsilon}\frac{\partial E_z}{\partial t} = 0. \tag{6.5}$$

These equations are sometimes called advection equations. Note that a solution to the wave equation will not simultaneously satisfy both these advection equations (except in trivial cases). It may satisfy one or the other but not both. In fact, fields may be a solution to the wave equation and yet satisfy neither of the advection equations.*

A solution to (6.4) is $E_z(t+\sqrt{\mu\epsilon}x)$, i.e., a wave traveling in the negative x direction. The proof proceeds along the same lines as the proof given in Sec. 2.16. Equate the argument with ξ so that

$$\xi = t+\sqrt{\mu\epsilon}x. \tag{6.6}$$

Derivatives of the argument with respect to time or space are given by

$$\frac{\partial\xi}{\partial t}=1 \quad \text{and} \quad \frac{\partial\xi}{\partial x}=\sqrt{\mu\epsilon}. \tag{6.7}$$

Thus,

$$\frac{\partial E_z}{\partial x} = \frac{\partial E_z}{\partial\xi}\frac{\partial\xi}{\partial x}=\sqrt{\mu\epsilon}\frac{\partial E_z}{\partial\xi}, \tag{6.8}$$

$$\frac{\partial E_z}{\partial t} = \frac{\partial E_z}{\partial\xi}\frac{\partial\xi}{\partial t}=\frac{\partial E_z}{\partial\xi}. \tag{6.9}$$

Plugging the right-hand sides of (6.8) and (6.9) into (6.4) yields zero and the equation is satisfied. It is worth mentioning that although $E_z(t+\sqrt{\mu\epsilon}x)$ is a solution to (6.4), it is not a solution to (6.5).

6.3 Terminating the Grid

Let us now consider how an advection equation can be used to provide an update equation for a node at the end of the computational domain. Let the node $E_z^{q+1}[0]$ be the node on the boundary for which an update equation is sought. Since interior nodes can be updated before the boundary node, assume that all the adjacent nodes in space-time are known, i.e., $E_z^{q+1}[1]$, $E_z^q[0]$, and $E_z^q[1]$ are known. At the left end of the grid, the fields should only be traveling to the left. Thus the fields satisfy the advection equation (6.4). The finite-difference approximation of this equation provides the necessary update equation, but the way to discretize the equation is not entirely obvious. A

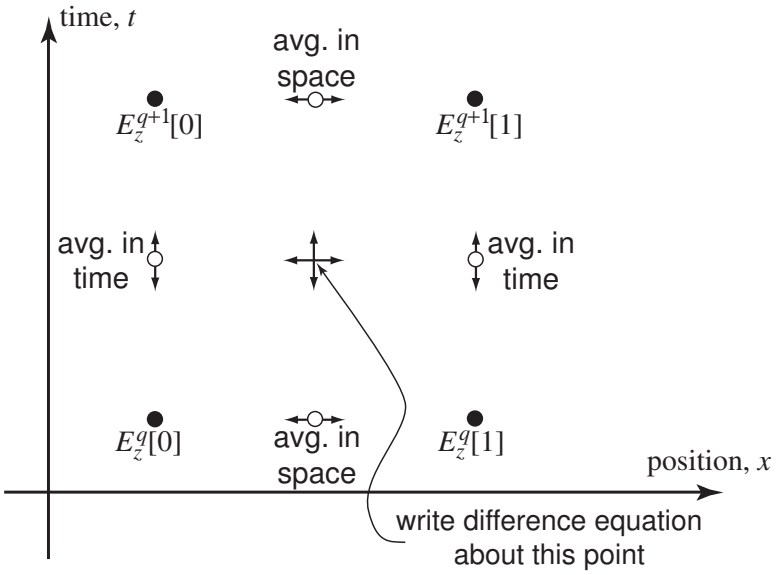


Figure 6.1: Space-time in the neighborhood of the left end of the grid. Only the electric fields are shown. The open circles indicated where an electric field is needed for the advection equation. Since there are none there, averaging is used to approximate the field at those points.

stable ABC will result if the equation is expanded about the space-time point $(\Delta_x/2, (q+1/2)\Delta_t)$. This point is shown in Fig. 6.1.

At first it seems like this in an unacceptable point about which to expand the advection equation since moving forward or backward in time or space a half step does not correspond to the location of an electric field point. To fix this, the electric field will be averaged in either time or space to obtain an estimate of the value at the desired location in space-time. For example, to obtain an approximation of $E_z^{q+1}[1/2]$, the average $(E_z^{q+1}[0] + E_z^{q+1}[1])/2$, would be used. Similarly, an approximation of $E_z^q[1/2]$ would be $(E_z^q[0] + E_z^q[1])/2$. Therefore the temporal derivative would be approximated with the following finite difference:

$$\sqrt{\mu\epsilon}\frac{\partial E_z}{\partial t}\bigg|_{\Delta_x/2,(q+1/2)\Delta_t} \approx \sqrt{\mu\epsilon}\frac{\frac{E_z^{q+1}[0]+E_z^{q+1}[1]}{2} - \frac{E_z^q[0]+E_z^q[1]}{2}}{\Delta_t} \tag{6.10}$$

Averaging in time is used to obtain the fields at the proper locations for the spatial finite difference. The resulting finite difference is

$$\frac{\partial E_z}{\partial x}\bigg|_{\Delta_x/2,(q+1/2)\Delta_t} \approx \frac{\frac{E_z^{q+1}[1]+E_z^q[1]}{2} - \frac{E_z^{q+1}[0]+E_z^q[0]}{2}}{\Delta_x} \tag{6.11}$$

Combining (6.10) and (6.11) yields the finite-difference form of the advection equation

$$\frac{\frac{E_z^{q+1}[1]+E_z^q[1]}{2} - \frac{E_z^{q+1}[0]+E_z^q[0]}{2}}{\Delta_x} - \sqrt{\mu\epsilon}\frac{\frac{E_z^{q+1}[0]+E_z^{q+1}[1]}{2} - \frac{E_z^q[0]+E_z^q[1]}{2}}{\Delta_t} = 0. \tag{6.12}$$

*As an example, consider $E_z(x,t) = \cos(\omega t)\sin(\beta x)$ where $\beta = \omega\sqrt{\mu\epsilon}$.

Letting $\sqrt{\mu\epsilon} = \sqrt{\mu_r\epsilon_r}/c$ and solving for $E_z^{q+1}[0]$ yields

$$E_z^{q+1}[0] = E_z^q[1] + \frac{\frac{S_c}{\sqrt{\mu_r\epsilon_r}} - 1}{\frac{S_c}{\sqrt{\mu_r\epsilon_r}} + 1}\left(E_z^{q+1}[1] - E_z^q[0]\right) \tag{6.13}$$

where S_c is the Courant number $c\Delta_t/\Delta_x$. Equation (6.13) provides a first-order absorbing boundary condition that updates the field on the boundary using the values of past and interior fields. This is known as a first-order ABC because it was constructed from a first-order differential equation. Note that when $S_c/\sqrt{\mu_r\epsilon_r}$ is unity, which would be the case of free space and a unit Courant number, (6.13) reduces to $E_z^{q+1}[0] = E_z^q[1]$ which is the simple grid-termination technique presented in Sec. 3.9.

At the other end of the grid, i.e., at the right end of the grid, an equation that is nearly identical to (6.13) pertains. Equation (6.5) would be expanded in the neighborhood of the last node of the grid. Although (6.4) and (6.5) differ in the sign of one term, when (6.5) is applied it is "looking" in the negative x direction. That effectively cancels the sign change. Hence the update equation for the last node in the grid, which is identified here as $E_z^{q+1}[M]$, would be

$$E_z^{q+1}[M] = E_z^q[M-1] + \frac{\frac{S_c}{\sqrt{\mu_r\epsilon_r}} - 1}{\frac{S_c}{\sqrt{\mu_r\epsilon_r}} + 1}\left(E_z^{q+1}[M-1] - E_z^q[M]\right). \tag{6.14}$$

Recall that for a lossless medium, the coefficient in the electric-field update equation that multiplied the magnetic fields was $\Delta_t/\epsilon\Delta_x$ and this could be expressed as $S_c\eta_0/\epsilon_r$. On the other hand, the coefficient in the magnetic-field update equation that multiplied the electric fields was $\Delta_t/\mu\Delta_x$ and this could be expressed as $S_c/\eta_0\mu_r$. Therefore, taking the product of these two coefficients and taking the square root yields

$$\left(\frac{\Delta_t}{\epsilon\Delta_x}\frac{\Delta_t}{\mu\Delta_x}\right)^{1/2} = \frac{S_c}{\sqrt{\mu_r\epsilon_r}}. \tag{6.15}$$

Note that this is the term that appears in (6.13) and (6.14). Thus by knowing the update coefficients that pertain at the ends of the grid, one can calculate the coefficients that appear in the ABC.

6.4 Implementation of a First-Order ABC

Program 3.6 modeled two half spaces: free space and a dielectric. That program was written as a "monolithic" program with a `main()` function in which all the calculations were performed. For that program we did not have a suitable way to terminate the grid within the dielectric. Let us re-implement that program but use the modular design that was discussed in Chap. 4 and use the ABC presented in the previous section.

Recalling the modular design used to model a lossy layer that was discussed in Sec. 4.9, we saw that the ABC was so simple there was no need to do any initialization of the ABC. Nevertheless, recalling the code shown in Program 4.17, an ABC initialization function was called, but it merely returned without doing anything. Now that we wish to implement a first-order ABC, the ABC initialization function actually needs to perform some calculations: it will calculate any of the constants associated with the ABC.

Naturally, the code associated with the various "blocks" in our modular design will need to change from what was presented in Sec. 4.9. However, the overall framework remains essentially the same! The arrangement of files associated with our model of halfspace that uses a first-order ABC is shown in Fig. 6.2. Note that this figure and Fig. 4.3 are nearly the same. They both have the same layout. All the functions have the same name, but the implementation of some of those functions have changed. Note that the file names have changed for the files containing the `main()` function, the `gridInit()` function, and the `abc()` and `abcInit()` function.

The code associated with the TFSF boundary, the snapshots, the source function, and the update equations are all unchanged from that which was described in Chap. 4. The grid initialization function `gridInit()` constructs a half-space dielectric consistent with Program 3.6. The code is shown in Program 6.1.

Program 6.1 `gridhalfspace.c`: Function to initialize the `Grid` such that there are two half-spaces: free space to the left and a dielectric with $\epsilon_r = 9$ to the right.

```
1  /* Function to initialize the Grid structure. */
2
3  #include "fdtd3.h"
4
5  #define EPSR 9.0
6
7  void gridInit(Grid *g) {
8    double imp0 = 377.0;
9    int mm;
10
11   SizeX = 200;   // size of domain
12   MaxTime = 450; // duration of simulation
13   Cdtds = 1.0;   // Courant number
14
15   ALLOC_1D(g->ez,   SizeX, double);
16   ALLOC_1D(g->ceze, SizeX, double);
17   ALLOC_1D(g->cezh, SizeX, double);
18   ALLOC_1D(g->hy,   SizeX - 1, double);
19   ALLOC_1D(g->chyh, SizeX - 1, double);
20   ALLOC_1D(g->chye, SizeX - 1, double);
21
22   /* set electric-field update coefficients */
23   for (mm = 0; mm < SizeX; mm++)
24     if (mm < 100) {
25       Ceze(mm) = 1.0;
26       Cezh(mm) = imp0;
27     } else {
28       Ceze(mm) = 1.0;
29       Cezh(mm) = imp0 / EPSR;
30     }
31
```

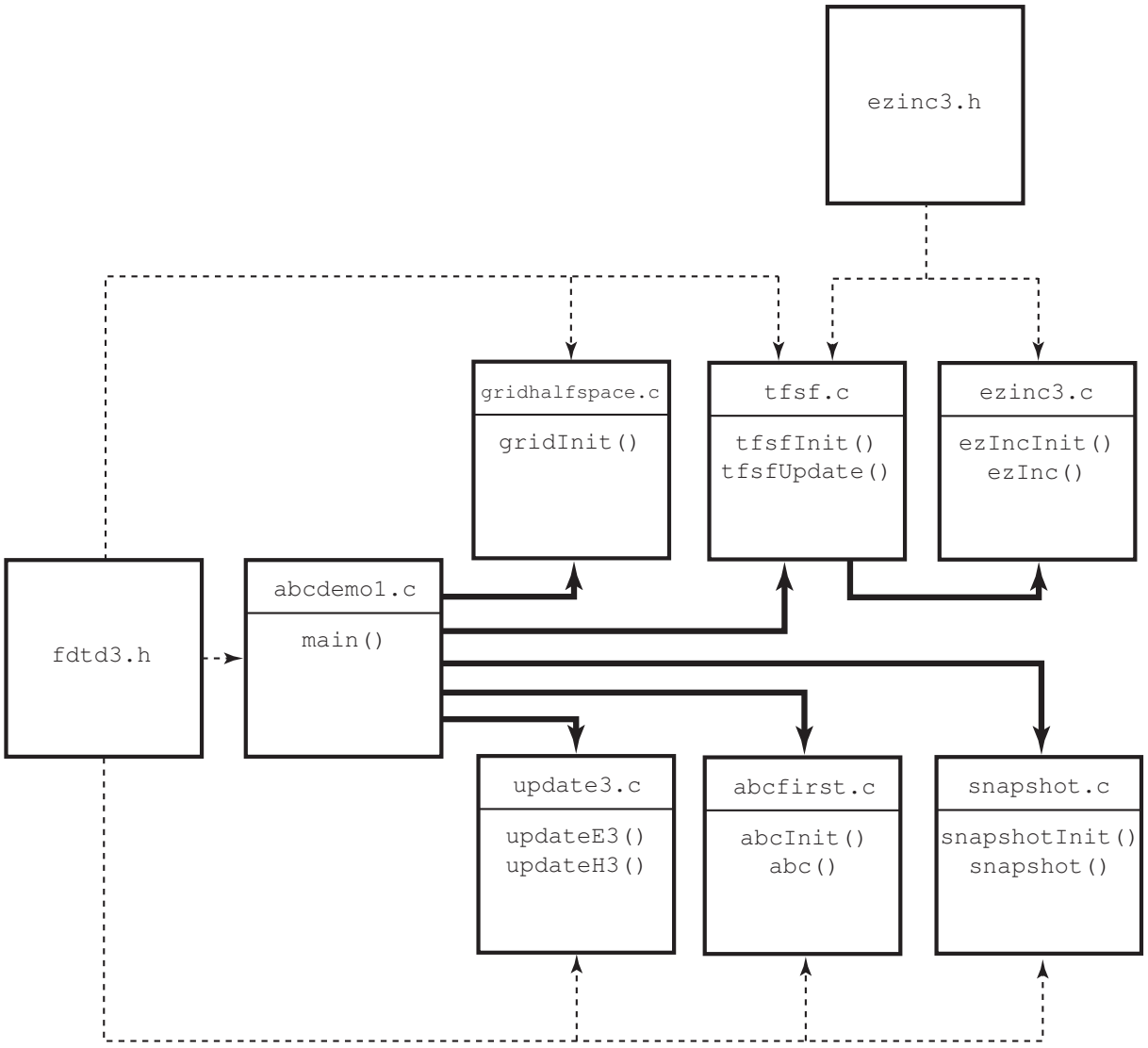


Figure 6.2: Files associated with the modular implementation of a simulation of a dielectric half space. Since a first-order ABC is now being used, some of these files differ from those of Fig. 4.3. Nevertheless, the overall structure of the program is unchanged and all the function names are the same as before.

```
32    /* set magnetic-field update coefficients */
33    for (mm = 0; mm < SizeX - 1; mm++) {
34      Chyh(mm) = 1.0;
35      Chye(mm) = 1.0 / imp0;
36    }
37
38    return;
39  }
```

The contents of the file `abcdemo1.c` are shown in Program 6.2. The difference between `abcdemo1.c` and `improved3.c`, which is given in Program 4.17, is shown in bold. In fact, the only change in the code is the location of the call of the ABC function `abc()`. Function `abc()` is now called in line 23 which is *after* the updating of the electric fields. Since the ABC relies on the "future" value of a neighboring interior electric field, this field must be updated before the node on the grid boundary can be updated. (The ABC function presented in Program 4.17 could, in fact, have been written in such a way that it would be called after the electric-field update. After all, the first-order ABC reduces to the simple ABC when the Courant number is one and the medium is free space. Nevertheless, the code in Program 4.17 was written to be consistent with the way in which the simple ABC was originally presented in Chap. 3.)

Program 6.2 `abcdemo1.c`: One-dimensional FDTD simulation employing a first-order ABC (although the actual ABC code is contained in the file `abcfirst.c` which is shown in Program 6.3). The difference between this program and Program 4.17 is shown in bold.

```
1   /* FDTD simulation where main() is primarily used to call other
2    * functions that perform the necessary operations. */
3
4   #include "fdtd3.h"
5
6   int main()
7   {
8     Grid *g;
9
10    ALLOC_1D(g, 1, Grid);  // allocate memory for Grid
11
12    gridInit(g);          // initialize the grid
13    abcInit(g);           // initialize ABC
14    tfsfInit(g);          // initialize TFSF boundary
15    snapshotInit(g);      // initialize snapshots
16
17    /* do time stepping */
18    for (Time = 0; Time < MaxTime; Time++) {
19
20      updateH3(g);    // update magnetic field
21      tfsfUpdate(g); // correct field on TFSF boundary
```

```
22      updateE3(g);    // update electric field
23      abc(g);          // apply ABC -- after E-field update
24      snapshot(g);    // take a snapshot (if appropriate)
25
26    } /* end of time-stepping */
27
28    return 0;
29  }
```

The code for `abcInit()` and `abc()` is contained in the file `abcfirst.c` which is shown in Program 6.3. The initialization function calculates the coefficients that appeared in (6.13) and (6.14). Recall from (6.15) that these coefficients can be obtained as a function of the electric- and magnetic-field update-equation coefficients. Since the material at the left and right side of the grid may be different, there are separate coefficients for the two sides.

Program 6.3 `abcfirst.c`: Implementation of a first-order absorbing boundary condition.

```
1  /* Function to implement a first-order ABC. */
2
3  #include "fdtd3.h"
4  #include <math.h>
5
6  static int initDone = 0;
7  static double ezOldLeft = 0.0, ezOldRight = 0.0;
8  static double abcCoefLeft, abcCoefRight;
9
10 /* Initialization function for first-order ABC. */
11 void abcInit(Grid *g) {
12   double temp;
13
14   initDone = 1;
15
16   /* calculate coefficient on left end of grid */
17   temp = sqrt(Cezh(0) * Chye(0));
18   abcCoefLeft = (temp - 1.0) / (temp + 1.0);
19
20   /* calculate coefficient on right end of grid */
21   temp = sqrt(Cezh(SizeX - 1) * Chye(SizeX - 2));
22   abcCoefRight = (temp - 1.0) / (temp + 1.0);
23
24   return;
25 }
26
27 /* First-order ABC. */
28 void abc(Grid *g) {
```

```
29    /* check if abcInit() has been called */
30    if (!initDone) {
31      fprintf(stderr,
32              "abc: abcInit must be called before abc.\n");
33      exit(-1);
34    }
35
36    /* ABC for left side of grid */
37    Ez(0) = ezOldLeft + abcCoefLeft * (Ez(1) - Ez(0));
38    ezOldLeft = Ez(1);
39
40    /* ABC for right side of grid */
41    Ez(SizeX - 1) = ezOldRight +
42      abcCoefRight * (Ez(SizeX - 2) - Ez(SizeX - 1));
43    ezOldRight = Ez(SizeX - 2);
44
45    return;
46  }
```

As shown in (6.13) and (6.14), for the first-order ABC both the "past" and the future value of the interior neighbor nearest to the boundary are needed. However, once we update the fields, past values are overwritten. Thus, the function `abc()`, which applies the ABC to the two ends of the grid, locally stores the "past" values of the nodes that are adjacent to the ends of the grid. For the left side of the grid the past neighbor is stored as `ezOldLeft` and on the right it is stored as `ezOldRight`. These are static global variables that are retained from one invocation of `abc()` to the next. Thus, even though the interior fields have been updated, these past values will still be available. (Once the nodes at the ends of the grid have been updated, `ezOldLeft` and `ezOldRight` are set to the current value of the neighboring nodes as shown in lines 38 and 43. When `abc()` is called next, these are indeed the "old" values of these neighbors.)

Figure 6.3 shows the waterfall plot of the snapshots generated by Program 6.2. One can see that this is the same as Fig. 3.13 prior to the transmitted field encountering the right end of the grid. After that time there is a reflected field evident in Fig. 3.13 but none is visible here. The ABC has absorbed the incident field and hence the grid behaves as if it were infinite. In reality this ABC is only approximate and there is some reflected field at the right boundary. We will return to this point in Sec. 6.6

6.5 ABC Expressed Using Operator Notation

Let us define an identity operator I, a forward spatial shift operator s_x^1, and a backward temporal shift operator s_t^{-1}. When they act on a node in the grid their affect is given by

$$IE_z^{q+1}[m] = E_z^{q+1}[m]\,, \tag{6.16}$$

$$s_x^1 E_z^{q+1}[m] = E_z^{q+1}[m+1]\,, \tag{6.17}$$

$$s_t^{-1} E_z^{q+1}[m] = E_z^{q}[m]\,. \tag{6.18}$$

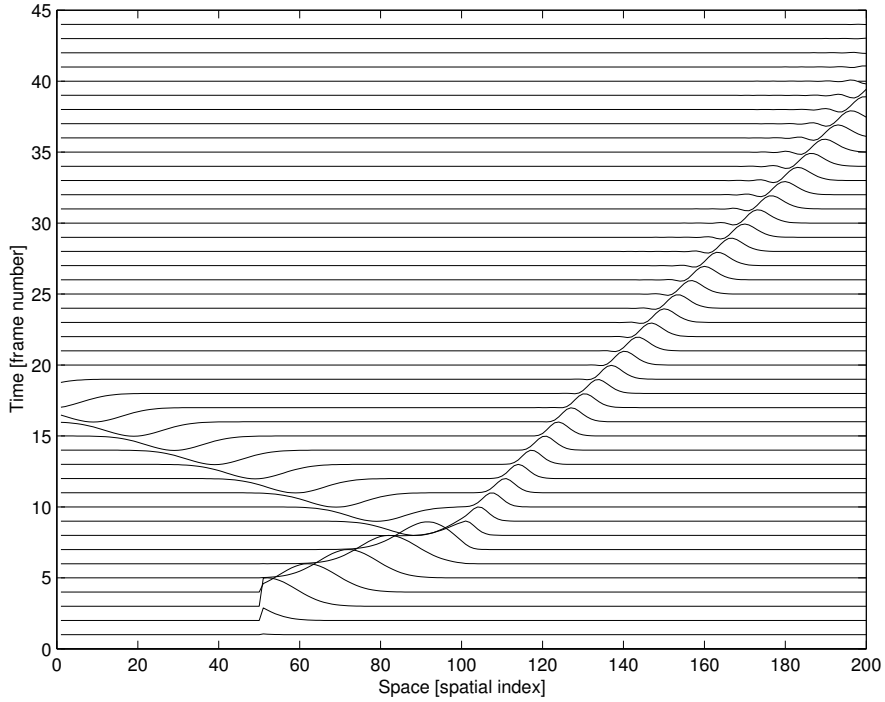


Figure 6.3: Waterfall plot of the snapshots generated by Program 6.2. Comparing this to Fig. 3.13 one sees they are the same until the transmitted field encounters the right end of the grid. At that point 3.13 shows there is a reflected field. However, because of the ABC used here, no reflected field is visible.

Note that these operators all commute, e.g., $s_x^1 s_t^{-1} = s_t^{-1} s_x^1$. Furthermore, the identity operator times another operator is merely that operator, e.g., $I s_x^1 = s_x^1$ likewise $II = I$. Using these operators a spatial average of a node can be represented by

$$\frac{E_z^{q+1}[m] + E_z^{q+1}[m+1]}{2} = \left(\frac{I + s_x^1}{2}\right) E_z^{q+1}[m]\,, \tag{6.19}$$

while a temporal average can be written

$$\frac{E_z^{q+1}[m] + E_z^{q}[m]}{2} = \left(\frac{I + s_t^{-1}}{2}\right) E_z^{q+1}[m]\,. \tag{6.20}$$

When applying the advection equation, the finite differences, as originally formulated, needed electric-field nodes where none were present in space-time. Therefore, averaging had to be used to obtain approximations to the fields at the desired locations. This required averaging in time followed by a spatial finite difference or averaging in space followed by a temporal finite difference. Consider the finite difference approximation to the temporal derivative at the point $(\Delta_x/2, (q + 1/2)\Delta_t)$. Starting from the node $E_z^{q+1}[0]$, this requires adding the node one spatial step to the right and then dividing by two. Then, going back one step in time, these same two nodes are again averaged and then subtracted from the previous average. The result is divided by the temporal step to obtain the temporal finite difference. Expressed in operator notation this is

$$\begin{aligned}
\left.\frac{\partial E_z}{\partial t}\right|_{\Delta_x/2,(q+1/2)\Delta_t} &= \left(\frac{I - s_t^{-1}}{\Delta_t}\right)\left(\frac{I + s_x^1}{2}\right) E_z^{q+1}[0] && (6.21)\\
&= \frac{1}{2\Delta_t}\left(I - s_t^{-1} + s_x^1 - s_t^{-1} s_x^1\right) E_z^{q+1}[0] && (6.22)\\
&= \frac{1}{2\Delta_t}\left(E_z^{q+1}[0] - E_z^{q}[0] + E_z^{q+1}[1] - E_z^{q}[1]\right). && (6.23)
\end{aligned}$$

The second term in parentheses in (6.21) accomplishes the averaging while the first term in parentheses yields the temporal finite difference. The result, shown in (6.23), is the same as (6.10) (other than the factor of $\sqrt{\mu\epsilon}$).

A similar approach can be used for the spatial finite difference about the same point except now the averaging is done in time

$$\begin{aligned}
\left.\frac{\partial E_z}{\partial x}\right|_{\Delta_x/2,(q+1/2)\Delta_t} &= \left(\frac{s_x^1 - I}{\Delta_x}\right)\left(\frac{I + s_t^{-1}}{2}\right) E_z^{q+1}[0] && (6.24)\\
&= \frac{1}{2\Delta_x}\left(-I + s_x^1 - s_t^{-1} + s_t^{-1} s_x^1\right) E_z^{q+1}[0] && (6.25)\\
&= \frac{1}{2\Delta_x}\left(-E_z^{q+1}[0] + E_z^{q+1}[1] - E_z^{q}[0] + E_z^{q}[1]\right). && (6.26)
\end{aligned}$$

This is exactly the same as (6.11).

From (6.21) and (6.24) we see the finite-difference form of the advection equation can be expressed as

$$\left\{\left(\frac{s_x^1 - I}{\Delta_x}\right)\left(\frac{I + s_t^{-1}}{2}\right) - \sqrt{\mu\epsilon}\left(\frac{I - s_t^{-1}}{\Delta_t}\right)\left(\frac{I + s_x^1}{2}\right)\right\} E_z^{q+1}[0] = 0. \tag{6.27}$$

Solving this equation for $E_z^{q+1}[0]$ yields the update equation (6.13). The term in braces is the finite-difference equivalent of the first advection operator that appeared on the left-hand side of (6.3).

6.6 Second-Order ABC

Equation (6.27) provides an update equation which is, in general, approximate. In many circumstance the field reflected by a first-order ABC is unacceptably large. A more accurate update equation can be obtained by applying the advection operator twice. Consider

$$\left(\frac{\partial}{\partial x} - \sqrt{\mu\epsilon}\frac{\partial}{\partial t}\right)\left(\frac{\partial}{\partial x} - \sqrt{\mu\epsilon}\frac{\partial}{\partial t}\right) E_z = 0. \tag{6.28}$$

Without employing too many mathematical details, assume the field E_z is not a proper solution to the advection equation. For example, the speed at which it is propagating is not precisely $1/\sqrt{\mu\epsilon}$. If the field is close to a proper solution, the advection operator operating on the field should yield a number which is close to zero. However, if the advection operator acts on it again, the result should be something smaller still—the equation is closer to the truth.

To demonstrate this, consider a wave $E_z(t+x/c')$ which is traveling in the negative x direction with a speed $c' \neq c$. Following the notation used in Sec. 6.2, the advection operator operating on this fields yields

$$\left(\frac{1}{c'} - \frac{1}{c}\right)\frac{\partial E_z}{\partial \xi}. \tag{6.29}$$

If the advection operator again operates on this, the result is

$$\left(\frac{1}{c'} - \frac{1}{c}\right)^2\frac{\partial^2 E_z}{\partial \xi^2}. \tag{6.30}$$

If c and c' are close, (6.30) will be smaller than (6.29) for a broad class of signals. Hence the repeated application of the advection operator may still only be approximately satisfied, but one anticipates that it will perform better than the first-order operator alone.

The finite-difference form of the second-order advection operator operating on the node $E_z^{q+1}[0]$ is

$$\begin{aligned}\Bigg[\left\{\left(\frac{s_x^1 - I}{\Delta_x}\right)\left(\frac{I + s_t^{-1}}{2}\right) - \sqrt{\mu\epsilon}\left(\frac{I - s_t^{-1}}{\Delta_t}\right)\left(\frac{I + s_x^1}{2}\right)\right\} \\ \left\{\left(\frac{s_x^1 - I}{\Delta_x}\right)\left(\frac{I + s_t^{-1}}{2}\right) - \sqrt{\mu\epsilon}\left(\frac{I - s_t^{-1}}{\Delta_t}\right)\left(\frac{I + s_x^1}{2}\right)\right\}\Bigg] E_z^{q+1}[0] &= 0.\end{aligned} \tag{6.31}$$

One expands this equation and solves for $E_z^{q+1}[0]$ to obtain the second-order ABC. The result is

$$\begin{aligned}E_z^{q+1}[0] = \frac{-1}{1/S_c' + 2 + S_c'}\Big\{&(1/S_c' - 2 + S_c')\left[E_z^{q+1}[2] + E_z^{q-1}[0]\right] \\ &+ 2(S_c' - 1/S_c')\left[E_z^q[0] + E_z^q[2] - E_z^{q+1}[1] - E_z^{q-1}[1]\right] \\ &- 4(1/S_c' + S_c')E_z^q[1]\Big\} - E_z^{q-1}[2]\end{aligned} \tag{6.32}$$

where $S_c' = \Delta_t/(\sqrt{\mu\epsilon}\Delta_x) = S_c/\sqrt{\mu_r\epsilon_r}$. This update equation requires two interior points at time step $q+1$ as well as the boundary node and these same interior points at time steps q and $q-1$. Typically these past values would not be available to use in the update equation and must therefore be stored in some auxiliary manner such as had to be done in Program 6.3 (there just a single point on either end of the grid needed to be stored). Two 3×2 arrays (one used at either end of the computational domain) could be used to store the values at the three spatial locations and two previous time steps required by (6.32). Alternatively, four 1D arrays (two used at either end) of three points each could also be used to stored the old values. (It may be noted that $E_z^q[0]$ would not need to be stored since it would be available when updating the boundary node. However, for the sake of symmetry when writing the loops which store the boundary values, it is simplest to store this value explicitly.)

When S_c' is unity, as would be the case for propagation in free space with a Courant number of unity, (6.32) reduces to

$$E_z^{q+1}[0] = 2E_z^q[1] - E_z^{q-1}[2]\,. \tag{6.33}$$

This may appear odd at first but keep in mind that the field is only traveling to the left and it moves one spatial step per time step, thus $E_z^q[1]$ and $E_z^{q-1}[2]$ are equal. Therefore this effectively reduces to $E_z^{q+1}[0] = E_z^q[1]$ which is again the original grid termination approach used in Sec. 3.9.

As was the case for first-order, for the right side of the grid, the second-order termination is essentially the same as the one on the left side. One merely uses interior nodes to the left of the boundary instead of to the right.

To demonstrate the improvement realized by using a second-order ABC instead of a first-order one, consider the same computational domain as was used in Program 6.2, i.e., a pulse is incident from free space to a dielectric half-space with a relative permittivity of 9 which begins at node 100. Figure 6.4 shows the electric field in the computational domain at time-step 550 (`MaxTime` was increased from the 450 shown in Program 6.1). Ideally the transmitted pulse would be perfectly absorbed and the reflected fields should be zero. However, for both the first- and second-order ABC's there is a reflected field, but it is significantly smaller in the case of the second-order ABC.

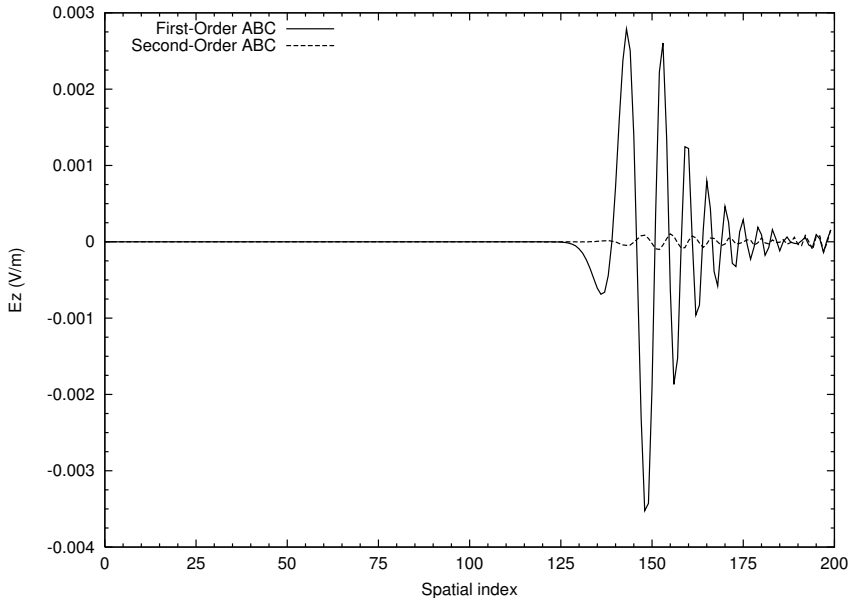


Figure 6.4: Plot of the fields at time step 550 when the grid is terminated with either a first- or second-order ABC. This snapshot is taken after the transmitted pulse has encountered the right edge of the grid. Hence this shows the field reflected by the ABC. Ideally the reflected fields should be zero.

6.7 Implementation of a Second-Order ABC

In order to implement a second-order ABC, one merely has to modify the functions `abcInit()` and `abc()`. Thus, none of the code depicted in Fig. 6.2 needs to change except for that which is directly related to the ABC itself. If instead of using the file `abcfirst.c`, one uses the code shown in Program 6.4, which we assume is stored in a file called `abcsecond.c`, then a second-order ABC will be realized.

Program 6.4 `abcsecond.c` The `abcInit()` and `abc()` functions for implementation of a second-orde ABC.

```
1  /* Functions to implement a second-order ABC. */
2
3  #include "fdtd3.h"
4  #include <math.h>
5
6  static int initDone = 0;
7  static double *ezOldLeft1, *ezOldLeft2,
8                *ezOldRight1, *ezOldRight2;
9  static double *abcCoefLeft, *abcCoefRight;
10
11 /* Initizalization function for second-order ABC. */
```

```
12 void abcInit(Grid *g) {
13   double temp1, temp2;
14
15   initDone = 1;
16
17   ALLOC_1D(ezOldLeft1, 3, double);
18   ALLOC_1D(ezOldLeft2, 3, double);
19   ALLOC_1D(ezOldRight1, 3, double);
20   ALLOC_1D(ezOldRight2, 3, double);
21
22   ALLOC_1D(abcCoefLeft, 3, double);
23   ALLOC_1D(abcCoefRight, 3, double);
24
25   /* calculate coefficients on left end of grid */
26   temp1 = sqrt(Cezh(0) * Chye(0));
27   temp2 = 1.0 / temp1 + 2.0 + temp1;
28   abcCoefLeft[0] = -(1.0 / temp1 - 2.0 + temp1) / temp2;
29   abcCoefLeft[1] = -2.0 * (temp1 - 1.0 / temp1) / temp2;
30   abcCoefLeft[2] = 4.0 * (temp1 + 1.0 / temp1) / temp2;
31
32   /* calculate coefficients on right end of grid */
33   temp1 = sqrt(Cezh(SizeX - 1) * Chye(SizeX - 2));
34   temp2 = 1.0 / temp1 + 2.0 + temp1;
35   abcCoefRight[0] = -(1.0 / temp1 - 2.0 + temp1) / temp2;
36   abcCoefRight[1] = -2.0 * (temp1 - 1.0 / temp1) / temp2;
37   abcCoefRight[2] = 4.0 * (temp1 + 1.0 / temp1) / temp2;
38
39   return;
40 }
41
42 /* Second-order ABC. */
43 void abc(Grid *g) {
44   int mm;
45
46   /* check if abcInit() has been called */
47   if (!initDone) {
48     fprintf(stderr,
49             "abc: abcInit must be called before abc.\n");
50     exit(-1);
51   }
52
53   /* ABC for left side of grid */
54   Ez(0) = abcCoefLeft[0] * (Ez(2) + ezOldLeft2[0])
55     + abcCoefLeft[1] * (ezOldLeft1[0] + ezOldLeft1[2] -
56                         Ez(1) - ezOldLeft2[1])
57     + abcCoefLeft[2] * ezOldLeft1[1] - ezOldLeft2[2];
58
```

```
59    /* ABC for right side of grid */
60    Ez(SizeX-1) =
61      abcCoefRight[0] * (Ez(SizeX - 3) + ezOldRight2[0])
62      + abcCoefRight[1] * (ezOldRight1[0] + ezOldRight1[2] -
63                           Ez(SizeX - 2) - ezOldRight2[1])
64      + abcCoefRight[2] * ezOldRight1[1] - ezOldRight2[2];
65
66    /* update stored fields */
67    for (mm = 0; mm < 3; mm++) {
68      ezOldLeft2[mm] = ezOldLeft1[mm];
69      ezOldLeft1[mm] = Ez(mm);
70
71      ezOldRight2[mm] = ezOldRight1[mm];
72      ezOldRight1[mm] = Ez(SizeX - 1 - mm);
73    }
74
75    return;
76  }
```

Lines 7–9 declare six static, global pointers, each of which will ultimately serve as an array of three points. Four of the arrays will store previous values of the electric field (two arrays dedicated to the left side and two to the right). The remaining two arrays store the coefficients used in the ABC (one array for the left, one for the right). The memory for these arrays is allocated in the `abcInit()` function in lines 17–23. That is followed by calculation of the coefficients on the two side of the grid.

Within the function `abc()`, which is called once per time-step, the values of the nodes on the edge of the grid are updated by the statements starting on lines 54 and 60. Finally, starting on line 67, the stored values are updated. The array `exOldLeft1` represents the field one time-step in the past while `exOldLeft2` represents the field two time-steps in the past. Similar naming is employed on the right.

Chapter 7

Dispersion, Impedance, Reflection, and Transmission

7.1 Introduction

A dispersion relation gives the relationship between frequency and the speed of propagation. This relationship is rather simple in the continuous world and is reviewed in the next section. Unfortunately dispersion in the FDTD is not as simple. Nevertheless, it provides a great deal of insight into the inherent limitations of the FDTD method and hence it is important that one have at least a basic understanding of it.

The tools developed in the analysis of FDTD dispersion can also be used to determine the characteristic impedance of the grid. Furthermore, as will be shown, knowing the dispersion relationship one can obtain exact analytic expressions for the reflection and transmission coefficients in the FDTD grid.

7.2 Dispersion in the Continuous World

Consider a plane wave propagating in the $+x$ direction in a lossless medium. In time-harmonic form the temporal and spatial dependence of the wave are given by $\exp(j[\omega t - \beta x])$ where ω is the frequency and β is the phase constant (wave number). The speed of the wave can be found by determining how fast a given point on the wave travels. In this context "point" is taken to mean a point of constant phase. The phase is dictated by $\omega t - \beta x$. Setting this equal to a constant and differentiating with respect to time gives

$$\frac{d}{dt}(\omega t - \beta x) = \frac{d}{dt}(\text{constant}), \tag{7.1}$$

$$\omega - \beta \frac{dx}{dt} = 0. \tag{7.2}$$

In this expression x is taken to be the position which provides a particular phase. In that sense it is not an independent variable. The location x which yields the desired phase will change as a

[†]Lecture notes by John Schneider. `fdtd-dispersion.tex`

function of time. Therefore, dx/dt is the speed of the wave, or, more properly, the phase speed c_p. Solving (7.2) for the phase speed yields

$$c_p = \frac{dx}{dt} = \frac{\omega}{\beta}. \tag{7.3}$$

This is apparently a function of frequency, but for a plane wave the phase constant β is given by $\omega\sqrt{\mu\epsilon}$. Thus the phase speed is

$$c_p = \frac{\omega}{\omega\sqrt{\mu\epsilon}} = \frac{1}{\sqrt{\mu_r\mu_0\epsilon_r\epsilon_0}} = \frac{c}{\sqrt{\mu_r\epsilon_r}} \tag{7.4}$$

where c is the speed of light in free space. Note that, in the continuous world for a lossless medium, the phase speed is independent of frequency and the dispersion relationship is

$$c_p = \frac{\omega}{\beta} = \frac{c}{\sqrt{\mu_r\epsilon_r}}. \tag{7.5}$$

Since c is a constant, and we are assuming μ_r and ϵ_r contact for the given material, all frequencies propagate at the same speed. Unfortunately this is not the case in the discretized FDTD world—different frequencies have different phase speeds.*

7.3 Harmonic Representation of the FDTD Method

The spatial shift-operator s_x and the temporal shift-operator s_t were introduced in Sec. 6.5. Generalizing these slightly, let a fractional superscript represent a corresponding fractional step. For example,

$$s_x^{1/2} H_y^q[m] = H_y^q\left[m + \frac{1}{2}\right], \tag{7.6}$$

$$s_t^{1/2} E_z^{q+\frac{1}{2}}[m] = E_z^{q+1}[m], \tag{7.7}$$

$$s_t^{-1/2} E_z^{q+\frac{1}{2}}[m] = E_z^q[m]. \tag{7.8}$$

Using these shift operators the finite-difference version of Ampere's law (ref. (3.17)) can be written

$$\epsilon\left(\frac{s_t^{1/2} - s_t^{-1/2}}{\Delta_t}\right) E_z^{q+\frac{1}{2}}[m] = \left(\frac{s_x^{1/2} - s_x^{-1/2}}{\Delta_x}\right) H_y^{q+\frac{1}{2}}[m]. \tag{7.9}$$

Note that both fields have a temporal index of $q + 1/2$. The index can be changed to q if the $1/2$ is accounted for by a temporal shift. Thus Ampere's law can also be written

$$s_t^{1/2}\epsilon\left(\frac{s_t^{1/2} - s_t^{-1/2}}{\Delta_t}\right) E_z^q[m] = s_t^{1/2}\left(\frac{s_x^{1/2} - s_x^{-1/2}}{\Delta_x}\right) H_y^q[m]. \tag{7.10}$$

*Later we will consider FDTD models of materials that are dispersive in the continuous world, i.e., materials for which ϵ or μ are functions of frequency. In fact, we have already considered dispersive behavior to some degree since lossy materials have phase speeds that are a function of frequency.

Let us define the finite-difference operator $\tilde{\partial}_i$ as

$$\tilde{\partial}_i = \left(\frac{s_i^{1/2} - s_i^{-1/2}}{\Delta_i} \right) \tag{7.11}$$

where i is either x or t. Using this notation the Yee version of Ampere's law can be written

$$\epsilon s_t^{1/2} \tilde{\partial}_t E_z^q[m] = s_t^{1/2} \tilde{\partial}_x H_y^q[m] \,. \tag{7.12}$$

Rather than obtaining an update equation from this, the goal is to determine the phase speed for a given frequency. To that end, we assume there is a single harmonic wave propagating such that

$$\hat{E}_z^q[m] = \hat{E}_0 e^{j(\omega q \Delta_t - \tilde{\beta} m \Delta_x)}, \tag{7.13}$$

$$\hat{H}_y^q[m] = \hat{H}_0 e^{j(\omega q \Delta_t - \tilde{\beta} m \Delta_x)}, \tag{7.14}$$

where $\tilde{\beta}$ is the phase constant which exists in the FDTD grid and $\hat{E}_0$ and $\hat{H}_0$ are constant amplitudes. A tilde will be used to indicate quantities in the FDTD grid which will typically (but not always!) differ from the corresponding value in the continuous world. Thus, the phase constant $\tilde{\beta}$ in the FDTD grid will differ, in general, from the phase constant β in the continuous world. As was done in Sec. 5.8 a caret (hat) will be used to indicate a harmonic quantity.

We will assume the frequency ω is the same in both the FDTD grid and the continuous world. Note that one has complete control over the frequency of the excitation—one merely has to ensure that the phase of the source changes a particular number of radians every time step. However, one does not have control over the phase constant, i.e., the spatial frequency. The grid dictates what $\tilde{\beta}$ will be for a given temporal frequency.

The plane-wave space-time dependence which appears in (7.13) and (7.14) essentially serves as an eigenfunction for the FDTD governing equations. If the governing equations operate on a function with this dependence, they will yield another function which has the same space-time dependence, albeit scaled by some value. To illustrate this, consider the temporal shift-operator acting on the electric field

$$\begin{aligned} s_t^{\pm 1/2} \hat{E}_z^q[m] &= \hat{E}_0 e^{j[\omega(q \pm 1/2)\Delta_t - \tilde{\beta} m \Delta_x]} \\ &= e^{\pm j\omega\Delta_t/2} \hat{E}_0 e^{j[\omega q \Delta_t - \tilde{\beta} m \Delta_x]} \\ &= e^{\pm j\omega\Delta_t/2} \hat{E}_z^q[m] \,. \end{aligned} \tag{7.15}$$

Similarly, the spatial shift-operator acting on the electric field yields

$$\begin{aligned} s_x^{\pm 1/2} \hat{E}_z^q[m] &= \hat{E}_0 e^{j[\omega q \Delta_t - \tilde{\beta}(m \pm 1/2)\Delta_x]} \\ &= e^{\mp j\tilde{\beta}\Delta_x/2} \hat{E}_0 e^{j[\omega q \Delta_t - \tilde{\beta} m \Delta_x]} \\ &= e^{\mp j\tilde{\beta}\Delta_x/2} \hat{E}_z^q[m] \,. \end{aligned} \tag{7.16}$$

Thus, for a plane wave, one can equate the shift operators with multiplication by an appropriate term:

$$s_t^{\pm 1/2} \Leftrightarrow e^{\pm j\omega\Delta_t/2}, \tag{7.17}$$

$$s_x^{\pm 1/2} \Leftrightarrow e^{\mp j\tilde{\beta}\Delta_x/2}. \tag{7.18}$$

Carrying this a step further, for plane-wave propagation the finite-difference operators $\tilde{\partial}_t$ and $\tilde{\partial}_x$ are equivalent to

$$\tilde{\partial}_t = \frac{e^{+j\omega\Delta_t/2} - e^{-j\omega\Delta_t/2}}{\Delta_t} = j\frac{2}{\Delta_t}\sin\left(\frac{\omega\Delta_t}{2}\right), \tag{7.19}$$

$$\tilde{\partial}_x = \frac{e^{-j\tilde{\beta}\Delta_x/2} - e^{+j\tilde{\beta}\Delta_x/2}}{\Delta_x} = -j\frac{2}{\Delta_x}\sin\left(\frac{\tilde{\beta}\Delta_x}{2}\right). \tag{7.20}$$

We define Ω and K_x as

$$\Omega = \frac{2}{\Delta_t}\sin\left(\frac{\omega\Delta_t}{2}\right), \tag{7.21}$$

$$K_x = \frac{2}{\Delta_x}\sin\left(\frac{\tilde{\beta}\Delta_x}{2}\right). \tag{7.22}$$

Note that as the discretization goes to zero, Ω approaches ω and K_x approaches $\tilde{\beta}$ (and, in fact, $\tilde{\beta}$ would approach β, the phase constant in the continuous world). Using this notation, taking a finite-difference with respect to time is equivalent to multiplication by $j\Omega$ while a finite difference with respect to space is equivalent to multiplication by $-jK_x$, i.e.,

$$\tilde{\partial}_t \Leftrightarrow j\Omega \tag{7.23}$$

$$\tilde{\partial}_x \Leftrightarrow -jK_x. \tag{7.24}$$

Using (7.17), (7.23), and (7.24) in Ampere's law (7.12) yields

$$j\epsilon\Omega e^{j\omega\Delta_t/2}\hat{E}_z^q[m] = -jK_x e^{j\omega\Delta_t/2}\hat{H}_y^q[m]\,. \tag{7.25}$$

The temporal shift $\exp(j\omega\Delta_t/2)$ is common to both side and hence can be canceled. Using the assumed form of the electric and magnetic fields from (7.13) and (7.14) in (7.25) yields

$$\epsilon\Omega\hat{E}_0 e^{j(\omega q\Delta_t - \tilde{\beta}m\Delta_x)} = -K_x\hat{H}_0 e^{j(\omega q\Delta_t - \tilde{\beta}m\Delta_x)}. \tag{7.26}$$

Canceling the exponential space-time dependence which is common to both sides produces

$$\epsilon\Omega\hat{E}_0 = -K_x\hat{H}_0. \tag{7.27}$$

Solving for the ratio of the electric and magnetic field amplitudes yields

$$\frac{\hat{E}_0}{\hat{H}_0} = -\frac{K_x}{\epsilon\Omega} = -\frac{\Delta_t}{\epsilon\Delta_x}\frac{\sin\left(\frac{\tilde{\beta}\Delta_x}{2}\right)}{\sin\left(\frac{\omega\Delta_t}{2}\right)}. \tag{7.28}$$

It appears that (7.28) is the "numeric impedance" since it is the ratio of the electric field to the magnetic field. In fact it *is* the numeric impedance, but it is only part of the story. As will be shown, the impedance in the FDTD method is exact. This fact is far from obvious if one only considers (7.28). It is also worth considering the corresponding continuous-world quantity $\beta/(\epsilon\omega)$:

$$\frac{\beta}{\epsilon\omega} = \frac{\omega/c_p}{\epsilon\omega} = \frac{1}{\epsilon c_p} = \frac{\sqrt{\mu\epsilon}}{\epsilon} = \sqrt{\frac{\mu}{\epsilon}} = \eta \tag{7.29}$$

Thus the fact that the grid numeric impedance is given by $K_x/(\epsilon\Omega)$ is consistent with continuous-world behavior. (The negative sign in (7.28) merely accounts for the orientation of the fields.)

7.4 Dispersion in the FDTD Grid

Another equation relating $\hat{E}_0$ and $\hat{H}_0$ can be obtained from Faraday's law. Expressed in terms of shift operators, the finite-difference form of Faraday's law (ref. (3.14)) is

$$\mu s_x^{1/2}\tilde{\partial}_t \hat{H}_y^q[m] = s_x^{1/2}\tilde{\partial}_x \hat{E}_z^q[m]\,. \tag{7.30}$$

As before, assuming plane-wave propagation, the shift operators can be replaced with multiplicative equivalents. The resulting equation is

$$j\mu\Omega e^{-j\omega\Delta_x/2}\hat{H}_y^q[m] = -jK_x e^{-j\omega\Delta_x/2}\hat{E}_z^q[m]\,. \tag{7.31}$$

Canceling terms common to both sides and rearranging yields

$$\frac{\hat{E}_0}{\hat{H}_0} = -\frac{\mu\Omega}{K_x} = -\frac{\mu\Delta_x}{\Delta_t}\frac{\sin\left(\frac{\omega\Delta_t}{2}\right)}{\sin\left(\frac{\tilde{\beta}\Delta_x}{2}\right)}. \tag{7.32}$$

Equating (7.28) and (7.32) and cross-multiplying gives

$$\mu\epsilon\Omega^2 = K_x^2. \tag{7.33}$$

This is the FDTD dispersion relation. Alternatively, expanding terms and rearranging slightly yields

$$\sin^2\left(\frac{\omega\Delta_t}{2}\right) = \frac{\Delta_t^2}{\epsilon\mu\Delta_x^2}\sin^2\left(\frac{\tilde{\beta}\Delta_x}{2}\right). \tag{7.34}$$

Taking the square root of both sides of either form of the dispersion relation yields

$$\sqrt{\mu\epsilon}\Omega = K_x, \tag{7.35}$$

or

$$\sin\left(\frac{\omega\Delta_t}{2}\right) = \frac{\Delta_t}{\sqrt{\epsilon\mu}\Delta_x}\sin\left(\frac{\tilde{\beta}\Delta_x}{2}\right). \tag{7.36}$$

These equations dictate the relationship between ω and $\tilde{\beta}$. Contrast this to the dispersion relation (7.5) which pertains to the continuous world. The two appear quite dissimilar! However, the two equations do agree in the limit as the discretization gets small.

The first term in the Taylor series expansion of $\sin(\xi)$ is ξ. Thus ξ provides a good approximation of $\sin(\xi)$ when ξ is small. Assume that the spatial and temporal steps are small enough so that the arguments of the sine functions in (7.36) are small. Retaining the first-order term in the Taylor-series expansion of the sine functions in (7.36) yields

$$\frac{\omega\Delta_t}{2} = \frac{\Delta_t}{\sqrt{\epsilon\mu}\Delta_x}\frac{\tilde{\beta}\Delta_x}{2}. \tag{7.37}$$

From this $\tilde{\beta}$ is seen to be

$$\tilde{\beta} = \omega\sqrt{\mu\epsilon}, \tag{7.38}$$

which is exactly the same as in the continuous world. However, this is only true when the discretization goes to zero. For finite discretization, the phase speed in the FDTD grid and in the continuous world differ.

In the continuous world the phase speed is $c_p = \omega/\beta$. In the FDTD world the same relation holds, i.e., $\tilde{c}_p = \omega/\tilde{\beta}$ where the tilde indicates this is the phase speed in the discretized world. In one dimension, a closed form solution for $\tilde{\beta}$ is possible (a similar dispersion relation holds in two and three dimensions, but there a closed-form solution is not possible). Bringing the coefficient to the other side of (7.36) and taking the arc sine yields

$$\frac{\tilde{\beta}\Delta_x}{2} = \sin^{-1}\left[\frac{\Delta_x\sqrt{\mu\epsilon}}{\Delta_t}\sin\left(\frac{\omega\Delta_t}{2}\right)\right]. \tag{7.39}$$

As was shown in Sec. 5.2.2 (ref. (5.7) and (5.8)), the factor $\omega\Delta_t/2$ is equivalent to $\pi S_c/N_\lambda$ where $S_c = c\Delta_t/\Delta_x$. Thus (7.39) can be written

$$\frac{\tilde{\beta}\Delta_x}{2} = \sin^{-1}\left[\frac{\sqrt{\mu_r\epsilon_r}}{S_c}\sin\left(\frac{\pi S_c}{N_\lambda}\right)\right]. \tag{7.40}$$

Consider the ratio of the phase speed in the grid to the true phase speed

$$\frac{\tilde{c}_p}{c_p} = \frac{\omega/\tilde{\beta}}{\omega/\beta} = \frac{\beta}{\tilde{\beta}} = \frac{\frac{\beta\Delta_x}{2}}{\frac{\tilde{\beta}\Delta_x}{2}}. \tag{7.41}$$

The phase constant in the continuous world can be written

$$\beta = \omega\sqrt{\mu\epsilon} = 2\pi\frac{c}{\lambda}\sqrt{\mu_0\epsilon_0\mu_r\epsilon_r} = \frac{2\pi}{\lambda}\sqrt{\mu_r\epsilon_r} = \frac{2\pi}{N_\lambda\Delta_x}\sqrt{\mu_r\epsilon_r} \tag{7.42}$$

where N_λ is the number of points per free-space wavelength. Using this in the numerator of the last term on the right-hand side of (7.41) and using (7.40) in the denominator, this ratio becomes

$$\frac{\tilde{c}_p}{c_p} = \frac{\pi\sqrt{\mu_r\epsilon_r}}{N_\lambda\sin^{-1}\left[\frac{\sqrt{\mu_r\epsilon_r}}{S_c}\sin\left(\frac{\pi S_c}{N_\lambda}\right)\right]}. \tag{7.43}$$

This equation is a function of the material parameters (ϵ_r and μ_r), the Courant number (S_c), and the number of points per wavelength (N_λ).

For propagation in free space, i.e., $\epsilon_r = \mu_r = 1$, when there are 20 points per wavelength and the Courant number S_c is $1/2$, the ratio of the numeric to the exact phase speed is approximately 0.9969 representing an error of 0.31 percent. Thus, for every wavelength of travel, the FDTD wave will accumulate about 1.12 degrees of phase error (0.0031×360). If the discretization is lowered to 10 points per wavelength, the ratio drops to 0.9873, or about a 1.27 percent error. Note that as the discretization was halved, the error increased by roughly a factor of four. This is as should be expected for a second-order method.

Consider the case of propagation in free space and a Courant number of 1. In that case the ratio collapses to

$$\frac{\tilde{c}_p}{c_p} = \frac{\pi}{N_\lambda\sin^{-1}\left[\sin\left(\frac{\pi}{N_\lambda}\right)\right]} = 1. \tag{7.44}$$

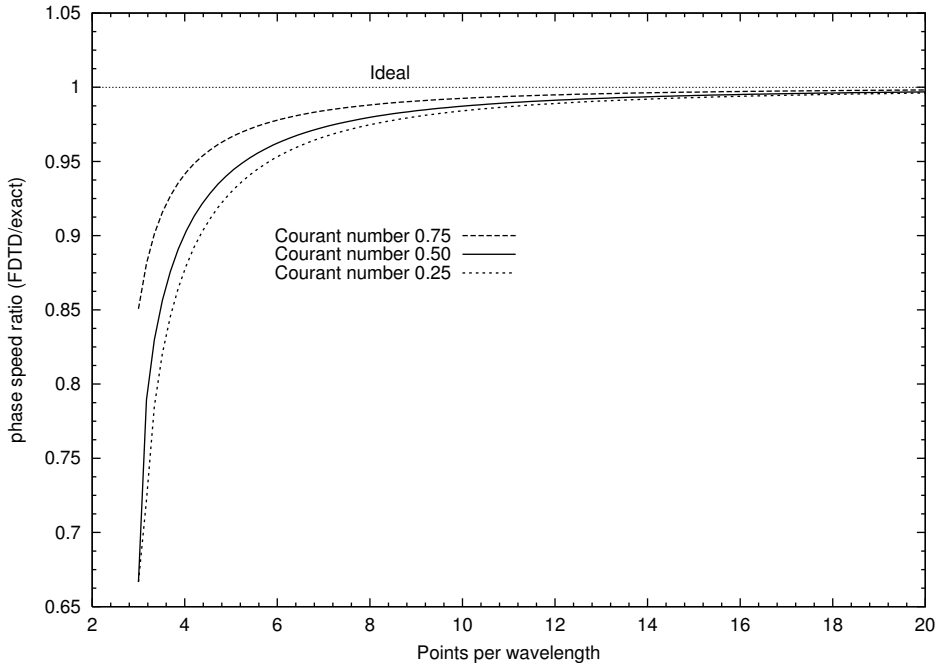
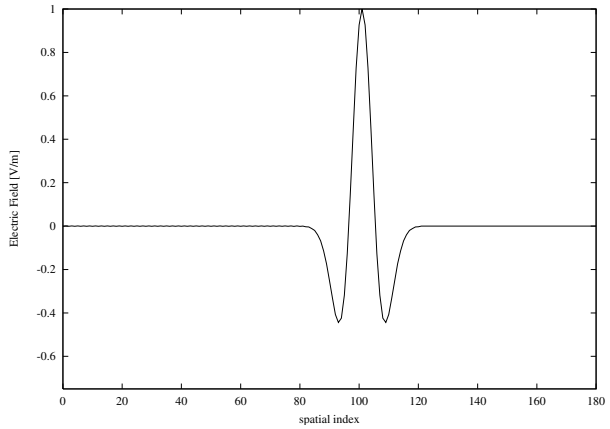


Figure 7.1: Ratio of the FDTD and exact phase speeds ($\tilde{c}_p/c_p$) versus the discretization. Propagation in free space is assumed. Ideally the ratio would be unity for all discretizations. Courant numbers of $1/4$, $1/2$, and $3/4$ are considered.

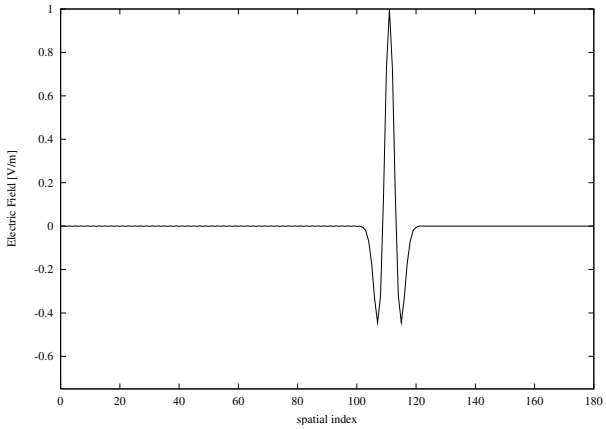
Thus the phase speed in the FDTD grid is exactly what it is in the continuous world! This is true for all discretization.

Figure 7.1 shows a plot of the ratio of the FDTD and exact phase speeds as a function of discretization. Three different Courant numbers are used. Ideally the ratio would be unity for all discretizations. As can be seen, the greater the Courant number, the closer the curve is to ideal. A large Courant number is thus desirable for two reasons. First, the larger the Courant number, the greater the temporal step and hence the more quickly a simulation advances (i.e., each update represents a greater advance in time). Second, the larger the Courant number, the smaller the dispersion error. In one dimension a Courant number of unity is the greatest possible and, since there is no dispersion error with this Courant number, the corresponding time step is known as the magic time-step. Unfortunately a magic time-step does not exist in higher dimensions.

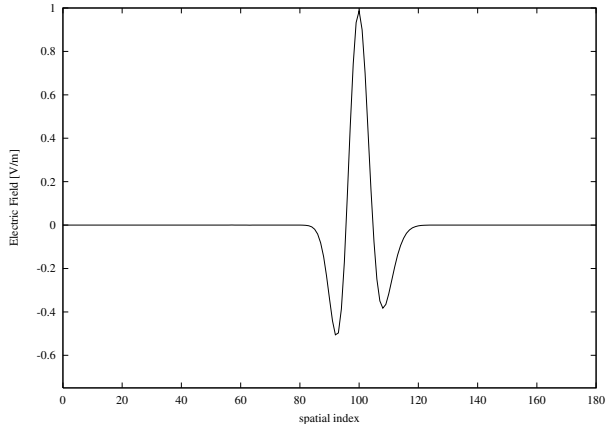
Figure 7.2 shows snapshots of Ricker wavelets propagating to the right that have been discretized at either 20 or 10 points per wavelength at the most energetic frequency (i.e., the parameter N_P discussed in Sec. 5.2.3 is either 20 or 10). The wavelets are propagating in grids that have a Courant number of either 1 or 0.5. In Figs. 7.2(a) and (b) the discretizations are 20 and 10, respectively, and the Courant number is unity. Since this corresponds to the magic time-step, the wavelets propagate without distortion. The discretizations in Figs. 7.2(c) and (d) are also 20 and 10, respectively, but now the Courant number is 0.5. The snapshots in (a) and (b) were taken after 100 time-steps while the snapshots in (c) and (d) were taken after 200 time-steps (since the time step is half as large in (c) and (d) as it is in (a) and (b), this ensures the snapshots are depicting the field at the same time). In Fig. 7.2(c) the distortion of the Ricker wavelet is visible in that the



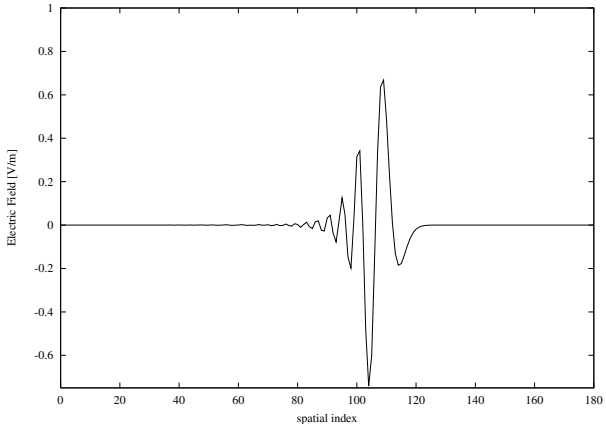
(a) $N_P = 20$, $S_c = 1$



(b) $N_P = 10$, $S_c = 1$



(c) $N_P = 20$, $S_c = 0.5$



(d) $N_P = 10$, $S_c = 0.5$

Figure 7.2: Snapshots of Ricker wavelets with different discretization propagating in grids with different Courant numbers. (a) 20 points per wavelength at the most energetic frequency, i.e., $N_P = 20$, $S_c = 1$, (b) $N_P = 10$, $S_c = 1$, (c) $N_P = 20$, $S_c = 0.5$, and (d) $N_P = 10$, $S_c = 0.5$. The snapshots were taken after 100 time-steps for (a) and (b), and after 200 time-steps for (c) and (d).

function is no longer symmetric about the peak. In Fig. 7.2(d) the distortion caused by dispersion is rather extreme and the function is no longer recognizable as a Ricker wavelet. The reason that Fig. 7.2(d) is so much more distorted than Fig. 7.2(c) is that the spectral energy lies at a coarser discretization. The more coarsely a harmonic is discretized, the more dispersion it will suffer—the higher frequencies propagate more slowly than the lower frequencies. This causes the ringing that is evident on the trailing side of the pulse.

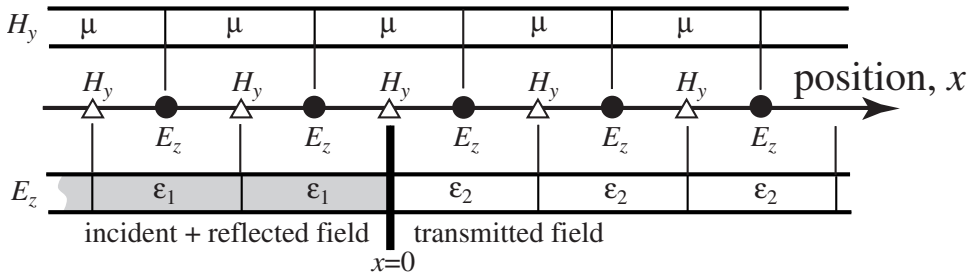


Figure 7.3: One-dimensional simulation where the permittivity changes abruptly at $x = 0$. The interface between the two media coincides with a magnetic-field node. The permeability is assumed to be constant throughout the computational domain. The figure depicts the nodes and the material values associated with their updates. The field to the left of the interface is the sum of the incident and reflected fields. The transmitted field exists to the right of the boundary.

7.5 Numeric Impedance

Let us return to (7.28) which nominally gave the numeric impedance and which is repeated below

$$\frac{\hat{E}_0}{\hat{H}_0} = -\frac{K_x}{\epsilon\Omega}. \tag{7.45}$$

The dispersion relation (7.35) expressed K_x in terms of Ω. Plugging this into (7.45) yields

$$\frac{\hat{E}_0}{\hat{H}_0} = -\frac{\sqrt{\mu\epsilon}\Omega}{\epsilon\Omega} = \sqrt{\frac{\mu}{\epsilon}} = \eta. \tag{7.46}$$

Thus, despite the inherent approximations in the FDTD method, the impedance in the grid is exactly the same as in the continuous world (this also holds in higher dimensions).

7.6 Analytic FDTD Reflection and Transmission Coefficients

Section 5.8.2 discussed the way in which an FDTD simulation could be used to measure the transmission coefficient associated with a planar interface. In this section, instead of using a simulation to measure the transmission coefficient, an expression will be derived that gives the transmission coefficient for the FDTD grid.

Consider a one-dimensional FDTD simulation as shown in Fig. 7.3. The permittivity changes abruptly at the magnetic field which is assumed to coincide with the interface at $x = 0$. The permeability is constant throughout the computational domain.

Assume there is an incident unit-amplitude plane wave propagating in the $+x$ direction. Because of the change in permittivity, a reflected field will exist to the left of the boundary which propagates in the $-x$ direction. Additionally, there will be a transmitted field which propagates to

the right of the boundary. The incident, reflected, and transmitted electrics fields are given by

$$\hat{E}_z^i[m,q] = \hat{E}_z^i[m]e^{j\omega q\Delta_t} = e^{-j\tilde{\beta}_1 m\Delta_x}e^{j\omega q\Delta_t}, \tag{7.47}$$

$$\hat{E}_z^r[m,q] = \hat{E}_z^r[m]e^{j\omega q\Delta_t} = \hat{\tilde{\Gamma}}e^{j\tilde{\beta}_1 m\Delta_x}e^{j\omega q\Delta_t}, \tag{7.48}$$

$$\hat{E}_z^t[m,q] = \hat{E}_z^t[m]e^{j\omega q\Delta_t} = \hat{\tilde{T}}e^{-j\tilde{\beta}_2 m\Delta_x}e^{j\omega q\Delta_t} \tag{7.49}$$

where $\hat{\tilde{\Gamma}}$ and $\hat{\tilde{T}}$ are the FDTD reflection and transmission coefficients, respectively. Note that all fields will have the common temporal phase factor $\exp(j\omega q\Delta_t)$. Therefore this term will not be explicitly written (its existence is implicit in the fact that we are doing harmonic analysis).

As shown in the previous section, the characteristic impedance in the FDTD grid is exact. Therefore the magnetic fields can be related to the electric fields in the same manner as they are in the continuous world, i.e.,

$$\hat{H}_y^i[m] = -\frac{1}{\eta_1}e^{-j\tilde{\beta}_1 m\Delta_x}, \tag{7.50}$$

$$\hat{H}_y^r[m] = \frac{\hat{\tilde{\Gamma}}}{\eta_1}e^{j\tilde{\beta}_1 m\Delta_x}, \tag{7.51}$$

$$\hat{H}_y^t[m] = -\frac{\hat{\tilde{T}}}{\eta_2}e^{-j\tilde{\beta}_2 m\Delta_x}. \tag{7.52}$$

The incident and reflected waves exist to the left of the interface and the transmitted field exists to the right of the interface. The magnetic field at the interface, i.e., the node at $x = 0$ is, in a sense, common to both the left and the right half-spaces. The field at this node can be described equally well as the sum of the incident and reflected waves or by the transmitted wave. This node enforces the continuity of the magnetic field across the boundary. Hence, just as in the continuous world, the fields are governed by the equation

$$-\frac{1}{\eta_1} + \frac{\hat{\tilde{\Gamma}}}{\eta_1} = -\frac{\hat{\tilde{T}}}{\eta_2}. \tag{7.53}$$

In the continuous world enforcing the continuity of the tangential electric and magnetic fields at a boundary yields two equations. These two equations can be used to solve for the reflection and transmission coefficients as was shown in Sec. 5.8.1. In FDTD, however, there are no electric field nodes that coincide with the interface (at least not with this geometry). Therefore continuity of the electric fields cannot be enforced directly nor is there a readily available second equation with which to solve for the reflection and transmission coefficients.

The necessary second equation is obtained by considering the update equation for the magnetic field at the interface. This node depends on the electric field to either side of the interface and hence ties the two half-spaces together. Specifically, the harmonic form of Faraday's law which governs the magnetic-field node at the interface can be used to write the following

$$\mu s_x^{1/2}\tilde{\partial}_t\hat{H}_y^q[m]\Big|_{x=0} = s_x^{1/2}\tilde{\partial}_x\hat{E}_z[m]\Big|_{x=0}, \tag{7.54}$$

$$\mu j\Omega\hat{H}_y[m]\Big|_{x=0} = \frac{1}{\Delta_x}\left(s_x^{1/2} - s_x^{-1/2}\right)\hat{E}_z^q[m]\Big|_{x=0}, \tag{7.55}$$

$$j\Omega\hat{H}_y[m]\Big|_{x=0} = \frac{1}{\mu\Delta_x}\left(\hat{\tilde{T}}e^{-j\tilde{\beta}_2\Delta_x/2} - \left[e^{j\tilde{\beta}_1\Delta_x/2} + \hat{\tilde{\Gamma}}e^{-j\tilde{\beta}_1\Delta_x/2}\right]\right). \tag{7.56}$$

In the last form of the equation, we have used the fact that the transmitted field is present when space is shifted a half spatial-step in the $+x$ direction relative to the boundary. Conversely the field is the sum of the incident and reflected waves when space is shifted a half spatial-step in the $-x$ direction relative to the boundary.

Equation (7.56) provides the second equation which, when coupled to (7.53), can be used to solve for the transmission and reflection coefficients. However, what is $\hat{H}_y[m]\Big|_{x=0}$? Since this node is on the interface, the expression for either the transmitted field or the sum of the incident and reflected field can be used. Using the transmitted field (evaluated at $m = 0$), the equation becomes

$$j\Omega\left(-\frac{\hat{\tilde{T}}}{\eta_2}\right) = \frac{1}{\mu\Delta_x}\left(\hat{\tilde{T}}e^{-j\tilde{\beta}_2\Delta_x/2} - \left[e^{j\tilde{\beta}_1\Delta_x/2} + \hat{\tilde{\Gamma}}e^{-j\tilde{\beta}_1\Delta_x/2}\right]\right). \tag{7.57}$$

Regrouping terms yields

$$e^{j\tilde{\beta}_1\Delta_x/2} + \hat{\tilde{\Gamma}}e^{-j\tilde{\beta}_1\Delta_x/2} = \left(e^{-j\tilde{\beta}_2\Delta_x/2} + j\frac{\Omega\mu\Delta_x}{\eta_2}\right)\hat{\tilde{T}}. \tag{7.58}$$

After multiplying through by $-\eta_1\exp(-j\tilde{\beta}_1\Delta_x/2)$, (7.53) becomes

$$e^{-j\tilde{\beta}_1\Delta_x/2} - \hat{\tilde{\Gamma}}e^{-j\tilde{\beta}_1\Delta_x/2} = \frac{\eta_1}{\eta_2}e^{-j\tilde{\beta}_1\Delta_x/2}\hat{\tilde{T}}. \tag{7.59}$$

Adding the left- and right-hand sides of (7.58) and (7.59) yields an expression that does not depend on the reflection coefficient. Multiplying this expression through by η_2 yields

$$\eta_2\left(e^{j\tilde{\beta}_1\Delta_x/2} + e^{-j\tilde{\beta}_1\Delta_x/2}\right) = \left(\eta_1 e^{-j\tilde{\beta}_1\Delta_x} + \eta_2 e^{-j\tilde{\beta}_2\Delta_x/2} + j\Omega\mu\Delta_x\right)\hat{\tilde{T}}. \tag{7.60}$$

Let us consider the third term in parentheses on the right-hand side. Recall from (7.35) that $\Omega = K_x/\sqrt{\mu\epsilon}$. Taking the material and propagation constants that pertain in the second medium,[†] we can write

$$\begin{aligned} j\Omega\mu\Delta_x &= j\frac{K_{x2}}{\sqrt{\mu\epsilon_2}}\mu\Delta_x, & (7.61)\\ &= j\sqrt{\frac{\mu}{\epsilon_2}}\frac{2}{\Delta_x}\sin\left(\frac{\tilde{\beta}_2\Delta_x}{2}\right)\Delta_x, & (7.62)\\ &= j\eta_2 2\left(\frac{e^{j\tilde{\beta}_2\Delta_x/2} - e^{-j\tilde{\beta}_2\Delta_x/2}}{j2}\right), & (7.63)\\ &= \eta_2\left(e^{j\tilde{\beta}_2\Delta_x/2} - e^{-j\tilde{\beta}_2\Delta_x/2}\right), & (7.64) \end{aligned}$$

where (7.22) was used to go from (7.61) to (7.62). Plugging this final form into (7.60), employing Euler's formula, and solving for the transmission coefficient yields

$$\hat{\tilde{T}} = \frac{2\eta_2\cos\left(\frac{\tilde{\beta}_1\Delta_x}{2}\right)}{\eta_1 e^{-j\tilde{\beta}_1\Delta_x/2} + \eta_2 e^{j\tilde{\beta}_2\Delta_x/2}}. \tag{7.65}$$

[†]As will be more obvious at the end of this derivation, we could instead select the material properties that pertain in the first medium and still obtain the same final result.

This can be compared to the exact transmission coefficient which is given in (5.88). At first it may appear that these are quite dissimilar. However, if the discretization is sufficiently small, the cosine and complex exponentials are close to unity (and become one as the discretization goes to zero). Hence the FDTD reflection coefficient reduces to the exact reflection coefficient as the discretization goes to zero.

The complex exponentials in (7.65) make it appear that the FDTD transmission coefficient is complex. This would impart a phase shift to the transmitted field that is not present in the continuous world. However, this is not the case—(7.65) can be simplified further. The one-dimensional dispersion relation (7.36) dictates that

$$\sin\left(\frac{\tilde{\beta}\Delta_x}{2}\right) = \frac{\sqrt{\epsilon\mu}\Delta_x}{\Delta_t}\sin\left(\frac{\omega\Delta_t}{2}\right). \tag{7.66}$$

Now consider the denominator of (7.65)

$$\begin{aligned}\eta_1 e^{-j\tilde{\beta}_1\Delta_x/2} + \eta_2 e^{j\tilde{\beta}_2\Delta_x/2} \\ = \quad \eta_1\left(\cos\left(\frac{\tilde{\beta}_1\Delta_x}{2}\right) - j\sin\left(\frac{\tilde{\beta}_1\Delta_x}{2}\right)\right) + \eta_2\left(\cos\left(\frac{\tilde{\beta}_2\Delta_x}{2}\right) + j\sin\left(\frac{\tilde{\beta}_2\Delta_x}{2}\right)\right)\end{aligned} \tag{7.67}$$

Using (7.66) to convert the sine terms and employing the definition of impedance, the denominator can be written

$$\begin{aligned}\eta_1\cos\left(\frac{\tilde{\beta}_1\Delta_x}{2}\right) + \eta_2\cos\left(\frac{\tilde{\beta}_2\Delta_x}{2}\right) \\ -j\sqrt{\frac{\mu}{\epsilon_1}}\frac{\sqrt{\epsilon_1\mu}\Delta_x}{\Delta_t}\sin\left(\frac{\omega\Delta_t}{2}\right) + j\sqrt{\frac{\mu}{\epsilon_2}}\frac{\sqrt{\epsilon_2\mu}\Delta_x}{\Delta_t}\sin\left(\frac{\omega\Delta_t}{2}\right).\end{aligned} \tag{7.68}$$

Upon canceling the ϵ's, the imaginary parts cancel and hence the denominator is purely real. Therefore another expression for the FDTD transmission coefficient is

$$\hat{\tilde{T}} = \frac{2\eta_2\cos\left(\frac{\tilde{\beta}_1\Delta_x}{2}\right)}{\eta_2\cos\left(\frac{\tilde{\beta}_2\Delta_x}{2}\right) + \eta_1\cos\left(\frac{\tilde{\beta}_1\Delta_x}{2}\right)}. \tag{7.69}$$

Combining this with (7.53) yields the reflection coefficient

$$\hat{\tilde{\Gamma}} = \frac{\eta_2\cos\left(\frac{\tilde{\beta}_2\Delta_x}{2}\right) - \eta_1\cos\left(\frac{\tilde{\beta}_1\Delta_x}{2}\right)}{\eta_2\cos\left(\frac{\tilde{\beta}_2\Delta_x}{2}\right) + \eta_1\cos\left(\frac{\tilde{\beta}_1\Delta_x}{2}\right)}. \tag{7.70}$$

Because the permeability is assumed to be constant throughout the computational domain, the reflection and transmission coefficients can be written as

$$\hat{\tilde{T}} = \frac{2\sqrt{\epsilon_1}\cos\left(\frac{\tilde{\beta}_1\Delta_x}{2}\right)}{\sqrt{\epsilon_1}\cos\left(\frac{\tilde{\beta}_2\Delta_x}{2}\right) + \sqrt{\epsilon_2}\cos\left(\frac{\tilde{\beta}_1\Delta_x}{2}\right)}, \tag{7.71}$$

$$\hat{\tilde{\Gamma}} = \frac{\sqrt{\epsilon_1}\cos\left(\frac{\tilde{\beta}_2\Delta_x}{2}\right) - \sqrt{\epsilon_2}\cos\left(\frac{\tilde{\beta}_1\Delta_x}{2}\right)}{\sqrt{\epsilon_1}\cos\left(\frac{\tilde{\beta}_2\Delta_x}{2}\right) + \sqrt{\epsilon_2}\cos\left(\frac{\tilde{\beta}_1\Delta_x}{2}\right)}. \tag{7.72}$$

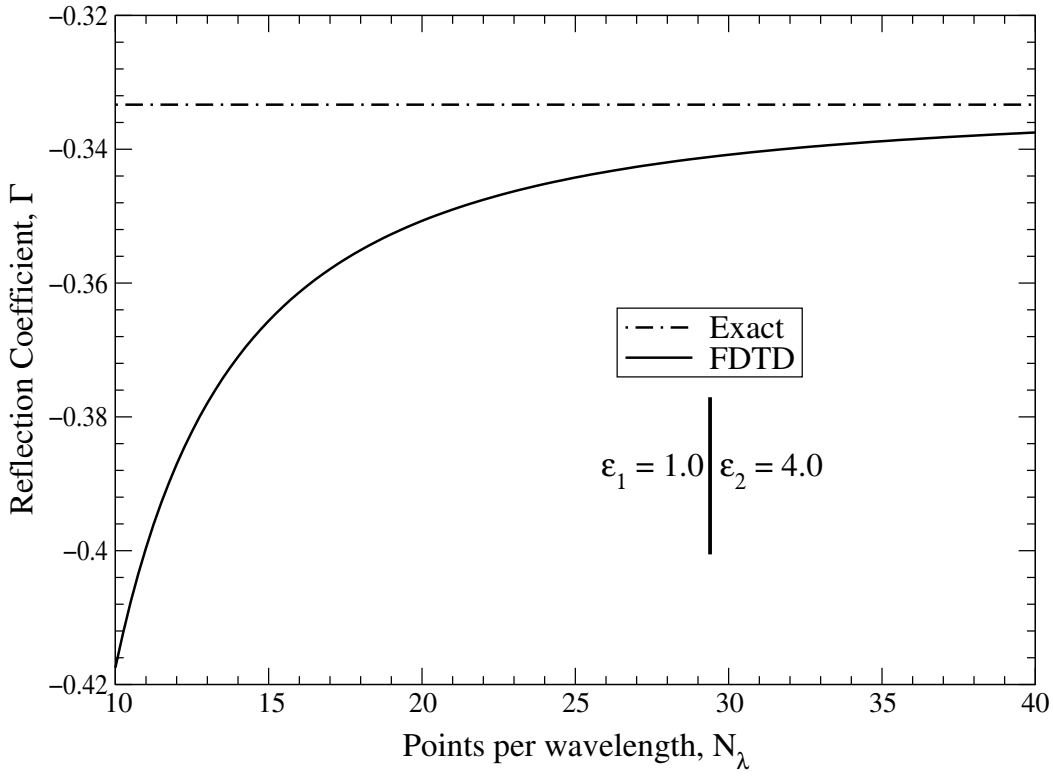


Figure 7.4: Reflection coefficient versus discretization for a wave traveling from free space to a dielectric with a relative permittivity of 4.0 (i.e., $\epsilon_{r1} = 1.0$ and $\epsilon_{r2} = 4.0$). The discretization N_λ shown on the horizontal axis is that which pertains to free space. (These values should be halved to give the discretization pertaining to the dielectric.)

Figure 7.4 shows a plot of the FDTD reflection coefficient versus the points per wavelength (in free space) when a wave is incident from free space to a dielectric with a relative permittivity of 4.0. For these materials the exact reflection coefficient, which is independent of the discretization and is also shown in the plot, is $\hat{\Gamma} = (1-2)/(1+2) = -1/3$. When the discretization is 10 points per wavelength the FDTD reflection coefficient is nearly -0.42 which corresponds to an error of approximately 26 percent. This error is rather large, but one must keep in mind that in the dielectric the discretization is only five points per wavelength. This coarse discretization affects the quality of the results throughout the computational domain, not just in the dielectric. Thus, even though one may ultimately be interested in the fields over only a portion of the computational domain, nevertheless, one must assure that a proper level of discretization is maintained throughout the grid.

7.7 Reflection from a PEC

A perfect electric conductor is realized by setting to zero electric field nodes. Let us assume that one wants to continue to define the interface as shown in Fig. 7.3, i.e., the boundary is assumed to coincide with a magnetic-field node. The new scenario is shown in Fig. 7.5. The electric-field

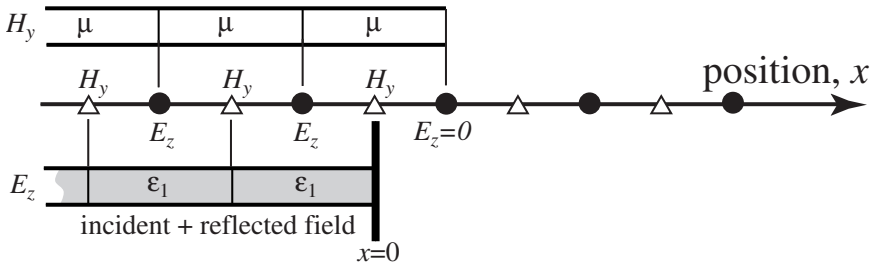


Figure 7.5: One-dimensional space with a perfect electric conductor realized by setting an electric field node to zero. No fields propagate beyond the zeroed node. The reference point $x = 0$ is still assumed to coincide with a magnetic-field node.

node to the right of the interface is set to zero. We now seek to find the reflection coefficient for this case. There is no transmitted field so the transmission coefficient $\hat{\tilde{T}}$ must be zero. That might lead one to think, given (7.53), that the reflection coefficient must be -1 as it is in the continuous world for a PEC boundary. However, that is not correct. Implicit in (7.53) is the assumption that the magnetic field is continuous across the interface. When a PEC is present, this is no longer the case. In the continuous world the discontinuity in the magnetic field is accounted for by a surface current.

When a PEC is present there is only one unknown, the reflection coefficient can be obtained from a single equation since the transmission coefficient is known *a priori*. This equation is provided, as before, by the update equation of the magnetic-field node at the interface. To this end, (7.56) is used with the transmission coefficient set to zero. Also, instead of using the transmitted form of the magnetic field at the interface as was done to obtain (7.57), the sum of the incident and reflected fields is used to obtain:

$$j\Omega\left(-\frac{1}{\eta_1}+\frac{\hat{\tilde{\Gamma}}}{\eta_1}\right)=\frac{1}{\mu\Delta_x}\left(0-\left[e^{j\tilde{\beta}_1\Delta_x/2}+\hat{\tilde{\Gamma}}e^{-j\tilde{\beta}_1\Delta_x/2}\right]\right). \tag{7.73}$$

Solving for the reflection coefficient yields

$$\hat{\tilde{\Gamma}}_{\text{PEC}}=\frac{j\Omega\mu\Delta_x-\eta_1 e^{j\tilde{\beta}_1\Delta_x/2}}{j\Omega\mu\Delta_x+\eta_1 e^{-j\tilde{\beta}_1\Delta_x/2}}. \tag{7.74}$$

As was shown in (7.61)–(7.64), the factor $j\Omega\mu\Delta_x$ can be expressed in terms of the impedance and a difference of complex exponentials. Employing such a conversion allows the reflection coefficient to be written as

$$\hat{\tilde{\Gamma}}_{\text{PEC}}=-e^{-j\tilde{\beta}_1\Delta_x}. \tag{7.75}$$

Note that the magnitude of the reflection coefficient is unity so that the entire incident field, regardless of the frequency, is reflected from the interface.

It appears that the FDTD reflection coefficient for a PEC is introducing a shift that is not present in the continuous world. However, this is being a bit unfair to the FDTD method. The location of the PEC boundary really corresponds to the electric-field node that was set to zero. Thus, one should really think of the PEC boundary existing at $x = \Delta_x/2$.

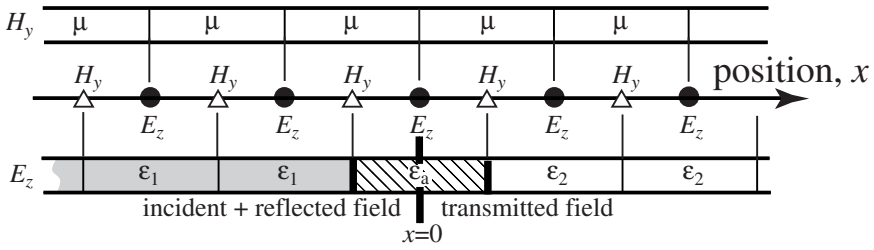


Figure 7.6: One-dimensional space with a discontinuity in permittivity. The interface in the continuous world corresponds to an electric-field node in the FDTD grid. The permittivity to the left of the interface is ϵ_1 and is ϵ_2 to the right. These values are dictated by those in the continuous world. The permittivity of the node at the interface is ϵ_a.

In the continuous world, let us consider a scenario where the origin is located a distance $\Delta_x/2$ in front of a PEC boundary. In that case the incident and reflected fields must sum to zero at $x = \Delta_x/2$, i.e.,

$$E^{\text{inc}}(x) + E^{\text{ref}}(x)\big|_{x=\Delta_x/2} = 0, \tag{7.76}$$

$$e^{-j\beta_1\Delta_x/2} + \hat{\Gamma}e^{j\beta_1\Delta_x/2} = 0. \tag{7.77}$$

Thus, in the continuous world the reflection coefficient is

$$\hat{\Gamma} = -e^{-j\beta_1\Delta_x}. \tag{7.78}$$

When first comparing (7.75) and (7.78) it may appear that in this case the FDTD reflection coefficient is exact. However the two differ owing to the fact that phase constants β_1 and $\tilde{\beta}_1$ are different in the two domains.

7.8 Interface Aligned with an Electric-Field Node

There are situations which necessitate that a discontinuity in permittivity be modeled as coinciding with an electric field node. The permittivity that should be used to either side of the interface is unambiguous but the permittivity of the node at the interface is open to question since it is neither in one half-space nor the other. This scenario was mentioned in Sec. 3.11 where it was suggested that the average permittivity be used for the node at the interface. In this section we wish to provide a more rigorous analysis to justify this permittivity. For now, let the permittivity of the node at the interface be ϵ_a as depicted in Fig. 7.6. The goal now is to find the reflection or transmission coefficients for this geometry and find the value of ϵ_a which yields the best agreement with the continuous-world values.

Although the origin $x = 0$ has shifted from that which was assumed in the previous section, the incident, reflected, and transmitted fields are still assumed to be given by (7.47)–(7.49) and (7.50)–(7.52). The electric-field node at the interface is a member of both half-spaces, i.e., the field is the same whether considered to be the sum of the incident and reflected field or simply to be the transmitted field. This yields a boundary condition of

$$1 + \hat{\tilde{\Gamma}} = \hat{\tilde{T}}. \tag{7.79}$$

Another equation relating the transmission and reflection coefficients is obtained via the update equation for the electric-field node at the interface. Ampere's law evaluated at the interface is

$$j\epsilon_a\Omega\hat{\tilde{T}} = \frac{1}{\Delta_x}\left(-\frac{\hat{\tilde{T}}}{\eta_2}e^{-j\tilde{\beta}_2\Delta_x/2} - \left[-\frac{1}{\eta_1}e^{j\tilde{\beta}_1\Delta_x/2} + \frac{\hat{\tilde{\Gamma}}}{\eta_1}e^{-j\tilde{\beta}_1\Delta_x/2}\right]\right). \tag{7.80}$$

Using (7.79) $\hat{\tilde{T}}$ can be replaced with $1+\hat{\tilde{\Gamma}}$. Then solving for $\hat{\tilde{\Gamma}}$ yields

$$\hat{\tilde{\Gamma}} = \frac{\eta_2 e^{j\tilde{\beta}_1\Delta_x/2} - \eta_1 e^{-j\tilde{\beta}_2\Delta_x/2} - j\eta_1\eta_2\epsilon_a\Omega\Delta_x}{\eta_2 e^{-j\tilde{\beta}_1\Delta_x/2} + \eta_1 e^{-j\tilde{\beta}_2\Delta_x/2} + j\eta_1\eta_2\epsilon_a\Omega\Delta_x}. \tag{7.81}$$

The term $\Omega\Delta_x$ can be written as

$$\begin{aligned}
\Omega\Delta_x &= \frac{2}{\Delta_t}\sin\left(\frac{\omega\Delta_t}{2}\right)\Delta_x, &(7.82)\\
&= 2c\frac{\Delta_x}{c\Delta_t}\sin\left(\frac{\omega\Delta_t}{2}\right), &(7.83)\\
&= 2c\frac{1}{S_c}\sin\left(\frac{2\pi c}{N_\lambda\Delta_x}\frac{\Delta_t}{2}\right), &(7.84)\\
&= 2c\frac{1}{S_c}\sin\left(\frac{\pi}{N_\lambda}S_c\right). &(7.85)
\end{aligned}$$

The conversion of $\omega\Delta_t/2$ to $\pi S_c/N_\lambda$ was discussed in connection with (5.74). Using this last form of $\Omega\Delta_x$, the third term in the numerator and denominator of (7.81) can be written

$$\begin{aligned}
j\eta_1\eta_2\epsilon_a\Omega\Delta_x &= j\sqrt{\frac{\mu}{\epsilon_0\epsilon_{r1}}}\sqrt{\frac{\mu}{\epsilon_0\epsilon_{r2}}}2c\epsilon_0\epsilon_{ra}\frac{1}{S_c}\sin\left(\frac{\pi}{N_\lambda}S_c\right), &(7.86)\\
&= j\frac{\epsilon_{ra}}{\sqrt{\epsilon_{r1}\epsilon_{r2}}}\frac{\mu}{\epsilon_0}2c\epsilon_0\frac{1}{S_c}\sin\left(\frac{\pi}{N_\lambda}S_c\right), &(7.87)\\
&= j\frac{\epsilon_{ra}}{\sqrt{\epsilon_{r1}\epsilon_{r2}}}2\mu c\frac{1}{S_c}\sin\left(\frac{\pi}{N_\lambda}S_c\right), &(7.88)
\end{aligned}$$

where ϵ_{ra} is the relative permittivity of the node at the interface.

Assume that the permeability throughout the computational domain is the permeability of free space so that μc in (7.88) can be written $\mu_0 c = \mu_0/\sqrt{\mu_0\epsilon_0} = \sqrt{\mu_0/\epsilon_0} = \eta_0$. The reflection coefficient (7.81) can now be written as

$$\hat{\tilde{\Gamma}} = \frac{\frac{1}{\sqrt{\epsilon_{r2}}}\eta_0 e^{j\tilde{\beta}_1\Delta_x/2} - \frac{1}{\sqrt{\epsilon_{r1}}}\eta_0 e^{-j\tilde{\beta}_2\Delta_x/2} - j\frac{\epsilon_{ra}}{\sqrt{\epsilon_{r1}\epsilon_{r2}}}\eta_0\frac{2}{S_c}\sin\left(\frac{\pi}{N_\lambda}S_c\right)}{\frac{1}{\sqrt{\epsilon_{r2}}}\eta_0 e^{-j\tilde{\beta}_1\Delta_x/2} + \frac{1}{\sqrt{\epsilon_{r1}}}\eta_0 e^{-j\tilde{\beta}_2\Delta_x/2} + j\frac{\epsilon_{ra}}{\sqrt{\epsilon_{r1}\epsilon_{r2}}}\eta_0\frac{2}{S_c}\sin\left(\frac{\pi}{N_\lambda}S_c\right)}. \tag{7.89}$$

Multiplying numerator and denominator by $\sqrt{\epsilon_{r1}\epsilon_{r2}}/\eta_0$ yields

$$\hat{\tilde{\Gamma}} = \frac{\sqrt{\epsilon_{r1}}e^{j\tilde{\beta}_1\Delta_x/2} - \sqrt{\epsilon_{r2}}e^{-j\tilde{\beta}_2\Delta_x/2} - j\epsilon_{ra}\frac{2}{S_c}\sin\left(\frac{\pi}{N_\lambda}S_c\right)}{\sqrt{\epsilon_{r1}}e^{-j\tilde{\beta}_1\Delta_x/2} + \sqrt{\epsilon_{r2}}e^{-j\tilde{\beta}_2\Delta_x/2} + j\epsilon_{ra}\frac{2}{S_c}\sin\left(\frac{\pi}{N_\lambda}S_c\right)}. \tag{7.90}$$

As is often the case, the expression for a quantity in the FDTD grid bears little resemblance to that in the continuous world. However, as the discretization goes to zero (i.e., N_λ goes to infinity), (7.90) does indeed reduce to the reflection coefficient in the continuous world.

The first two terms in the numerator and denominator of (7.90) depend on the material constants to either side of the interface while the third term depends on the permittivity at the interface, the Courant number, and the number of points per wavelength (in continuous-world free space). We now seek the value of ϵ_a (or the corresponding relative permittivity ϵ_{ra}) which minimizes the difference between the reflection coefficient in the continuous world and the one in the FDTD world. It is important to note that in general the FDTD reflection coefficient in this case is truly complex—there will be a phase shift imparted that does not exist in the continuous world.

Before going further with the analysis, let us consider an example with specific parameters and graphically solve for the optimum value of ϵ_a. Let $\epsilon_{r1} = 1$, $\epsilon_{r2} = 4$, and the discretization be 10 points per wavelength. In this case the reflection coefficient in the continuous world is $-1/3$ (independent of frequency). Figure 7.7 shows the FDTD reflection coefficient in the complex plane for various values of ϵ_{ra}. The continuous-world result, i.e., the exact result, is a single point on the negative real axis. As ϵ_{ra} varies a curve is obtained in the complex plane which is closest to the exact value when ϵ_{ra} is 2.5 which is the arithmetic average of 1 and 4. Thus the optimum value for the interface permittivity is the average of the permittivities to either side. However, these results are only for a specific discretization and for one set of permittivities. Is the average value the optimum one for all permittivities and discretizations?

To answer this question, let us return to (7.81) and separate the numerator and denominator into real and imaginary parts. The result is

$$\hat{\tilde{\Gamma}} = \frac{\eta_2 \cos(\kappa_1) - \eta_1 \cos(\kappa_2) + j\left[\eta_2 \sin(\kappa_1) + \eta_1 \sin(\kappa_2) - \eta_1\eta_2\epsilon_a\Omega\Delta_x\right]}{\eta_2 \cos(\kappa_1) + \eta_1 \cos(\kappa_2) - j\left[\eta_2 \sin(\kappa_1) + \eta_1 \sin(\kappa_2) - \eta_1\eta_2\epsilon_a\Omega\Delta_x\right]} \tag{7.91}$$

where

$$\kappa_1 = \tilde{\beta}_1\Delta_x/2, \tag{7.92}$$

$$\kappa_2 = \tilde{\beta}_2\Delta_x/2. \tag{7.93}$$

The continuous-world reflection coefficient is purely real and thus any imaginary part is an error. Furthermore, the imaginary part goes to zero as the discretization goes to zero. Let us consider just this imaginary part of the numerator and denominator. The $\sin(\kappa)$ terms can be related to the K terms which have been used previously—one merely had to multiply (and divide) by $2/\Delta_x$. Thus the imaginary part can be written

$$\eta_2\frac{2}{\Delta_x}\sin(\kappa_1)\frac{\Delta_x}{2} + \eta_1\frac{2}{\Delta_x}\sin(\kappa_2)\frac{\Delta_x}{2} - \eta_1\eta_2\epsilon_a\Omega\Delta_x = \eta_1\eta_2\frac{\Delta_x}{2}\left(\frac{1}{\eta_1}K_1 + \frac{1}{\eta_2}K_2 - 2\epsilon_a\Omega\right). \tag{7.94}$$

From the dispersion relation 7.33 we know that $K_1 = \sqrt{\mu\epsilon_1}\Omega$ and $K_2 = \sqrt{\mu\epsilon_2}\Omega$. This allows the imaginary part of the numerator and denominator of the reflection coefficient to be written

$$\eta_1\eta_2\frac{\Delta_x}{2}\Omega\left(\epsilon_1 + \epsilon_2 - 2\epsilon_a\right). \tag{7.95}$$

Setting ϵ_a equal to $(\epsilon_1 + \epsilon_2)/2$ will yield zero for this imaginary term. Hence the average permittivity is the optimum value for all permittivities and discretizations! Using the average permittivity

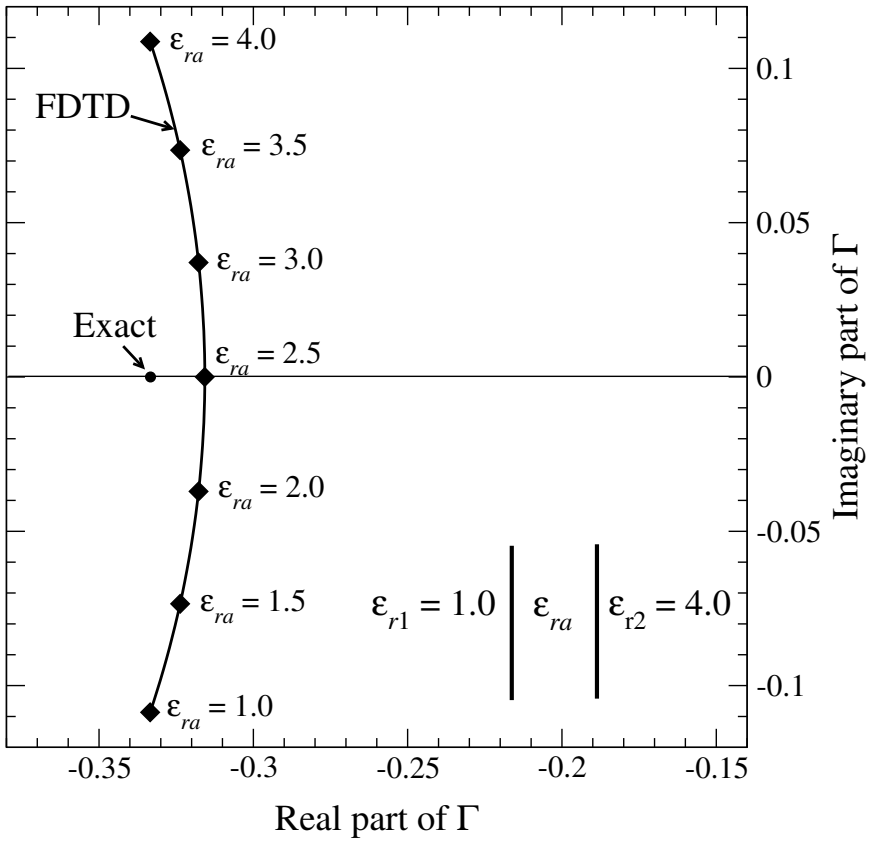


Figure 7.7: Curve showing the FDTD reflection coefficient in the complex plane as a function of ϵ_{ra} as ϵ_{ra} varies between the relative permittivity to the left of the interface ($\epsilon_{ra} = 1$) and the relative permittivity to the right of the interface ($\epsilon_{ra} = 4$). The value of ϵ_{ra} which minimizes the difference between the FDTD reflection coefficient and the exact value of $-1/3$ is the average permittivity, i.e., $\epsilon_{ra} = 2.5$.

yields reflection and transmission coefficients of

$$\hat{\tilde{\Gamma}} = \frac{\eta_2 \cos(\kappa_1) - \eta_1 \cos(\kappa_2)}{\eta_2 \cos(\kappa_1) + \eta_1 \cos(\kappa_2)}, \tag{7.96}$$

$$\hat{\tilde{T}} = \frac{2\eta_2 \cos(\kappa_1)}{\eta_2 \cos(\kappa_1) + \eta_1 \cos(\kappa_2)}. \tag{7.97}$$

Note that these equations appear almost identical to those which pertained to the case of an abrupt interface (i.e., when no averaging is done and the resulting reflection and transmission coefficients are (7.70) and (7.69)). However, these equations differ in the arguments of the cosine terms and, in fact, it can be shown that the abrupt boundary is slightly more accurate than the one which is implemented with the average permittivity at the interface. Nevertheless, when the situation calls for the interface to coincide with a tangential electric field, the average permittivity is the optimum one to use.

Chapter 8

Two-Dimensional FDTD Simulations

8.1 Introduction

One of the truly compelling features of the FDTD method is that the simplicity the method enjoys in one dimension is largely maintained in higher dimensions. The complexity of other numerical techniques often increases substantially as the number of dimensions increases. With the FDTD method, if you understand the algorithm in one dimension, you should have no problem understanding the basic algorithm in two or three dimensions. Nevertheless, introducing additional dimensions has to come at the price of some additional complexity.

This chapter provides details concerning the implementation of two-dimensional simulations with either the magnetic field or the electric field orthogonal to the normal to the plane of propagation, i.e., TM^z or TE^z polarization, respectively. Multidimensional simulations require that we think in terms of a multidimensional grid and thus we require multidimensional arrays to store the fields. Since we are primarily interested in writing programs in C, we begin with a discussion of multidimensional arrays in the C language.

8.2 Multidimensional Arrays

Whether an array is one-, two-, or three-dimensional, it ultimately is using a block of contiguous memory where each element has a single address. The distinction between the different dimensions is primarily a question of how we want to think about, and access, array elements. For higher-dimensional arrays, we want to specify two, or more, indices to dictate the element of interest. However, regardless of the number of indices, ultimately a single address dictates which memory is being accessed. The translation of multiple indices to a single address can be left to the compiler or that translation is something we can do ourselves.

The code shown in Fragment 8.1 illustrates how one can think and work with what is effectively a two-dimensional array even though the memory allocation is essentially the same as was used with the one-dimensional array considered in Fragment 4.1. In Fragment 8.1 the user is prompted to enter the desired number of rows and columns which are stored in `num_rows` and `num_columns`, respectively. In line 8 the pointer `ez` is set to the start of a block of memory

[†]Lecture notes by John Schneider. `fdtd-multidimensional.tex`

which can hold num_rows × num_columns doubles. Thus sufficient space is available to store the desired number of rows and columns, but this pointer is dereferenced with a single index (or offset).

Fragment 8.1 Fragment of code demonstrating the construction of a two-dimensional array.

```
1   #define Ez(I, J) ez[(I) * num_columns + (J)]
        ⋮
2   double *ez;
3   int num_rows, num_columns, m, n;
4
5   printf("Enter the number of row and columns: ");
6   scanf("%d %d", &num_rows, &num_columns);
7
8   ez = calloc(num_rows * num_columns, sizeof(double));
9
10  for (m=0; m < num_rows; m++)
11    for (n=0; n < num_columns; n++)
12      Ez(m, n) = m * n;
```

In this code the trick to thinking and working in two dimensions, i.e., working with two indices instead of one, is the macro which is shown in line 1. This macro tells the preprocessor that every time the string Ez appears with two arguments—here the first argument is labeled I and the second argument is labeled J—the compiler should translate that to ez[(I) * num_columns + (J)]. Note the uppercase E in Ez distinguishes this from the pointer ez. I and J in the macro are just dummy arguments. They merely specify how the first and second arguments should be used in the translated code. Thus, if Ez(2, 3) appears in the source code, the macro tells the preprocessor to translate that to ez[(2) * num_columns + (3)]. In this way we have specified two indices, but the macro translates those indices into an expression which evaluates to a single number which represents the offset into a one-dimensional array (the array ez). Even though Ez is technically a macro, we will refer to it as an array. Note that, as seen in line 12, Ez's indices are enclosed in parentheses, not brackets.

To illustrate this further, assume the user specified that there are 4 columns (num_columns is 4) and 3 rows. When the row number increments by one, that corresponds to moving 4 locations forward in memory. The following shows the arrangement of the two-dimensional array Ez(m, n) where m is used for the row index and n is used for the column index.

	n=0	n=1	n=2	n=3
m=0	Ez(0, 0)	Ez(0, 1)	Ez(0, 2)	Ez(0, 3)
m=1	Ez(1, 0)	Ez(1, 1)	Ez(1, 2)	Ez(1, 3)
m=2	Ez(2, 0)	Ez(2, 1)	Ez(2, 2)	Ez(2, 3)

The Ez array really corresponds to the one-dimensional ez array. The macro calculates the index for the ez array based on the arguments (i.e., indices) of the Ez array. The following shows the same table of values, but in terms of the ez array.

	n=0	n=1	n=2	n=3
m=0	ez[0]	ez[1]	ez[2]	ez[3]
m=1	ez[4]	ez[5]	ez[6]	ez[7]
m=2	ez[8]	ez[9]	ez[10]	ez[11]

Again, in this example, when the row is incremented by one, the array index is incremented by 4 (which is the number of columns). This is due to the fact that we are storing the elements by rows. An entire row of values is stored, then the next row, and so on. Each row contains `num_columns` elements.

Instead of storing things by rows, we could have easily employed "column-centric storage" where an entire column is stored contiguously. This would be accomplished by using the macro

```
#define Ez(I, J)   ez[(I) + (J) * num_rows]
```

This would be used instead of line 1 in Fragment 8.1. If this were used, each time the row is incremented by one the index of `ez` is incremented by one. If the column is incremented by one, the index of `ez` would be incremented by `num_rows`. In this case the elements of the `ez` array would correspond to elements of the `Ez` array as shown below:

	n=0	n=1	n=2	n=3
m=0	ez[0]	ez[3]	ez[6]	ez[9]
m=1	ez[1]	ez[4]	ez[7]	ez[10]
m=2	ez[2]	ez[5]	ez[8]	ez[11]

Note that when the row index is incremented by one, the index of `ez` is also incremented by one. However, when the column is incremented by one, the index of `ez` is incremented by 3, which is the number of rows. This type of column-centric storage is used in FORTRAN. However, multidimensional arrays in C are generally assumed to be stored in row-order. Thus column-centric storage will *not* be considered further and we will use row-centric macros similar to the one presented in Fragment 8.1.

When an array is stored by rows, it is most efficient to access the array one row at a time—not one column at a time. Lines 10 through 12 of Fragment 8.1 demonstrate this by using two loops to set the elements of `Ez` to the product of the row and column indices. The inner loop is over the row and the outer loop sets the column. This is more efficient than if these loops were interchanged (although there is likely to be no difference for small arrays). This is a consequence of the way memory is stored both on the disk and in the CPU cache.

Memory is accessed in small chunks known as pages. If the CPU needs access to a certain variable that is not already in the cache, it will generate a page fault (and servicing a page fault takes more time than if the variable were already in the cache). When the page gets to the cache it contains more than just the variable the CPU wanted—it contains other variables which were stored in memory adjacent to the variable of interest (the page may contain many variables). If the subsequent CPU operations involve a variable that is already in the cache, that operation can be done very quickly. It is most likely that that variable will be in the cache, and the CPU will not have to generate a page fault, if that variable is adjacent in memory to the one used in the previous operation. Thus, assuming row-centric storage, working with arrays row-by-row is the best way to avoid needless page faults.

It is important to note that we should not feel constrained to visualize our arrays in terms of the standard way in which arrays are displayed! Typically two-dimensional arrays are displayed in a table with the first element in the upper, left-hand corner of the table. The first index gives the row number and the second index gives the column number. FDTD simulations are modeling a physical space, not a table of numbers. In two dimensions we will be concerned with spatial dimensions x and y. It is convention to give the x location as the first argument and the y location as the second argument, i.e., $E_z(x, y)$. It is also often the case that it is convenient to think of the lower left-hand corner of some finite rectangular region of space as the origin. It is perfectly acceptable to use the array mechanics which have been discussed so far but to imagine them corresponding to an arrangement in space which corresponds to our usual notion of variation in the x and y directions. So, for example, in the case of a 3 by 4 array, one can visualize the nodes as being arranged in the following way:

n=3	Ez(0,3)	Ez(1,3)	Ez(2,3)
n=2	Ez(0,2)	Ez(1,2)	Ez(2,2)
n=1	Ez(0,1)	Ez(1,1)	Ez(2,1)
n=0	Ez(0,0)	Ez(1,0)	Ez(2,0)
	m=0	m=1	m=2

$\Longleftrightarrow$

n=3	ez[3]	ez[7]	ez[11]
n=2	ez[2]	ez[6]	ez[10]
n=1	ez[1]	ez[5]	ez[9]
n=0	ez[0]	ez[4]	ez[8]
	m=0	m=1	m=2

Nothing has changed in terms of the implementation of the macro to realize this two-dimensional array—the only difference is the way the elements have been displayed. The depiction here is natural when thinking in terms of variations in x and y, where the first index corresponds to x and the second index corresponds to y. The previous depiction was natural to the way most people discuss tables of data. Regardless of how we think of the arrays, it is still true that the second index is the one that should vary most rapidly in the case of nested loops (i.e., one should strive to have consecutive operations access consecutive elements in memory).

As mentioned in Sec. 4.2, it is always best to check that an allocation of memory was successful. If `calloc()` is unable to allocated the requested memory, it will return `NULL`. After every allocation we could add code to check that the request was successful. However, as we did in one-dimension, a better approach is again offered with the use of macros. Fragment 8.2 shows a macro that can be used to allocate memory for a two-dimensional array.

Fragment 8.2 Macro for allocating memory for a two-dimensional array.

```
#define ALLOC_2D(PNTR, NUMX, NUMY, TYPE)                              \
    PNTR = (TYPE *)calloc((NUMX) * (NUMY), sizeof(TYPE));             \
    if (!PNTR) {                                                      \
      perror("ALLOC_2D");                                             \
      fprintf(stderr,                                                 \
          "Allocation failed for " #PNTR ".  Terminating...\n");      \
      exit(-1);                                                       \
    }
```

The macro `ALLOC_2D()` is similar to `ALLOC_1D()`, which was presented in Fragment 4.2, except it takes four arguments instead of three. The first argument is a pointer, the second is the number

of rows, the third is the number of columns, and the fourth is the data type. As an example of how this macro could be used, line 8 of Fragment 8.1 could be replaced with

```
ALLOC_2D(ez, num_rows, num_columns, double);
```

8.3 Two Dimensions: TM^z Polarization

The one-dimensional problems considered thus far assumed a non-zero z component of the electric field and variation only in the x direction. This required the existence of a non-zero y component of the magnetic field. Here the field is assumed to vary in both the x and y directions, but not the z direction. From the outset we will include the possibility of a magnetic conductivity σ_m. With these assumptions Faraday's law becomes

$$-\sigma_m \mathbf{H} - \mu \frac{\partial \mathbf{H}}{\partial t} = \nabla \times \mathbf{E} = \begin{vmatrix} \hat{\mathbf{a}}_x & \hat{\mathbf{a}}_y & \hat{\mathbf{a}}_z \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & E_z \end{vmatrix} = \hat{\mathbf{a}}_x \frac{\partial E_z}{\partial y} - \hat{\mathbf{a}}_y \frac{\partial E_z}{\partial x}. \tag{8.1}$$

Since the right-hand side only has non-zero components in the x and y directions, the time-varying components of the magnetic field can only have non-zero x and y components (we are not concerned with static fields nor a rather pathological case where the magnetic current $\sigma_m \mathbf{H}$ cancels the time-varying field $\partial \mathbf{H}/\partial t$). The magnetic field is transverse to the z direction and hence this is designated the TM^z case. Ampere's law becomes

$$\sigma \mathbf{E} + \epsilon \frac{\partial \mathbf{E}}{\partial t} = \nabla \times \mathbf{H} = \begin{vmatrix} \hat{\mathbf{a}}_x & \hat{\mathbf{a}}_y & \hat{\mathbf{a}}_z \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & 0 \\ H_x & H_y & 0 \end{vmatrix} = \hat{\mathbf{a}}_z \left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} \right). \tag{8.2}$$

The scalar equations obtained from (8.1) and (8.2) are

$$-\sigma_m H_x - \mu \frac{\partial H_x}{\partial t} = \frac{\partial E_z}{\partial y}, \tag{8.3}$$

$$\sigma_m H_y + \mu \frac{\partial H_y}{\partial t} = \frac{\partial E_z}{\partial x}, \tag{8.4}$$

$$\sigma E_z + \epsilon \frac{\partial E_z}{\partial t} = \frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y}. \tag{8.5}$$

Note that, ignoring the conduction terms for a moment, the temporal derivative of the magnetic field is related to the spatial derivative of the electric field and vice versa. The only difference from the one-dimensional case is the additional field component H_x and the derivatives in the y direction.

Space-time is now discretized so that (8.3)–(8.5) can be expressed in terms of finite-differences. From these difference equations the future fields can be expressed in terms of past fields. Following the notation used in Sec. 3.3, the following notation will be used:

$$H_x(x, y, t) = H_x(m\Delta_x, n\Delta_y, q\Delta_t) = H_x^q[m, n], \tag{8.6}$$

$$H_y(x, y, t) = H_y(m\Delta_x, n\Delta_y, q\Delta_t) = H_y^q[m, n], \tag{8.7}$$

$$E_z(x, y, t) = E_z(m\Delta_x, n\Delta_y, q\Delta_t) = E_z^q[m, n]. \tag{8.8}$$

As before, the index m corresponds to the spatial step in the x direction while the index q corresponds to the temporal step. Additionally the index n represents the spatial step in the y direction. The spatial step sizes in the x and y directions are Δ_x and Δ_y, respectively (these need not be equal).

In order to obtain the necessary update-equations, each of the field components must be staggered in space. However, it is not necessary to stagger all the field components in time. The electric field must be offset from the magnetic field, but the magnetic field components do not need to be staggered relative to each other—all the magnetic field components can exist at the same time. A suitable spatial staggering of the electric and magnetic field components is shown in Fig. 8.1.

When we say the dimensions of a TM^z grid is $M \times N$, that corresponds to the dimensions of the E_z array. We will ensure the grid is terminated such that there are electric-field nodes on the edge of the grid. Thus, the H_x array would by $M \times (N-1)$ while the H_y array would be $(M-1) \times N$.

In the arrangement of nodes shown in Fig. 8.1 we will assume the electric field nodes fall at integer spatial steps and the magnetic field nodes are offset a half spatial step in either the x or y direction. As with one dimension, the electric field is assumed to exist at integer multiples of the temporal step while both magnetic fields components are offset a half time-step from the electric fields. With this arrangement in mind, the finite difference approximation of (8.3) expanded about the space-time point $(m\Delta_x, (n+1/2)\Delta_y, q\Delta_t)$ is

$$-\sigma_m \frac{H_x^{q+\frac{1}{2}}\left[m, n+\frac{1}{2}\right] + H_x^{q-\frac{1}{2}}\left[m, n+\frac{1}{2}\right]}{2} - \mu \frac{H_x^{q+\frac{1}{2}}\left[m, n+\frac{1}{2}\right] - H_x^{q-\frac{1}{2}}\left[m, n+\frac{1}{2}\right]}{\Delta_t} = \frac{E_z^q[m, n+1] - E_z^q[m, n]}{\Delta_y}. \tag{8.9}$$

This can be solved for the future value $H_x^{q+\frac{1}{2}}\left[m, n+\frac{1}{2}\right]$ in terms of the "past" values. The resulting update equation is

$$H_x^{q+\frac{1}{2}}\left[m, n+\frac{1}{2}\right] = \frac{1-\frac{\sigma_m \Delta_t}{2\mu}}{1+\frac{\sigma_m \Delta_t}{2\mu}} H_x^{q-\frac{1}{2}}\left[m, n+\frac{1}{2}\right] - \frac{1}{1+\frac{\sigma_m \Delta_t}{2\mu}} \frac{\Delta_t}{\mu \Delta_y} \left(E_z^q[m, n+1] - E_z^q[m, n]\right). \tag{8.10}$$

As was the case in one dimension, the material parameters μ and σ_m are those which pertain at the given evaluation point.

The update equation for the y component of the magnetic field is obtained by the finite-difference approximation of (8.4) expanded about the space-time point $((m+1/2)\Delta_x, n\Delta_y, q\Delta_t)$. The resulting equation is

$$H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}, n\right] = \frac{1-\frac{\sigma_m \Delta_t}{2\mu}}{1+\frac{\sigma_m \Delta_t}{2\mu}} H_y^{q-\frac{1}{2}}\left[m+\frac{1}{2}, n\right] + \frac{1}{1+\frac{\sigma_m \Delta_t}{2\mu}} \frac{\Delta_t}{\mu \Delta_x} \left(E_z^q[m+1, n] - E_z^q[m, n]\right). \tag{8.11}$$

Again, the material parameters μ and σ_m are those which pertain at the given evaluation point. Note that H_y nodes are offset in space from H_x nodes. Hence the μ and σ_m appearing in (8.10) and (8.11) are not necessarily the same even when m and n are the same.

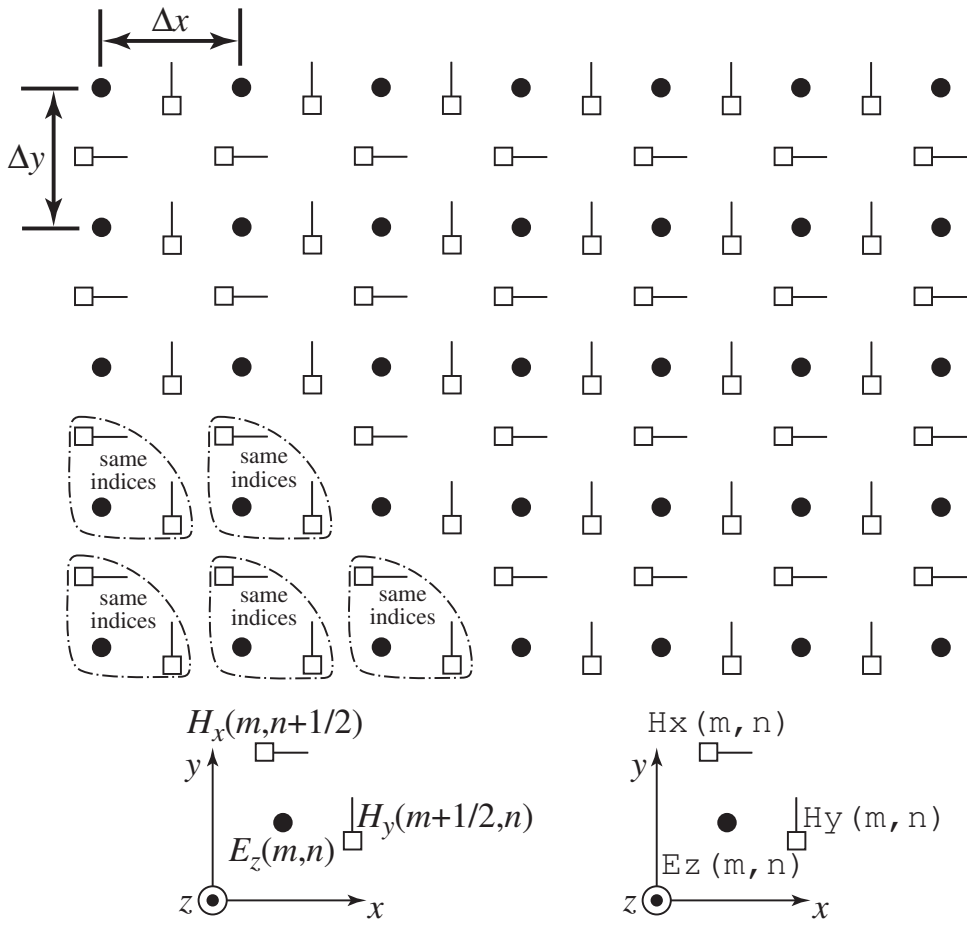


Figure 8.1: Spatial arrangement of electric- and magnetic-field nodes for TM^z polarization. The electric-field nodes are shown as circles and the magnetic-field nodes as squares with a line that indicates the orientation of the field component. The somewhat triangularly shaped dashed lines indicate groupings of nodes that have the same array indices. For example, in the lower left corner of the grid all the nodes would have indices in a computer program of $(m = 0, n = 0)$. In this case the spatial offset of the fields is implicitly understood. This grouping is repeated throughout the grid. However, groups at the top of the grid lack an H_x node and groups at the right edge lack an H_y node. The diagram at the bottom left of the figure indicates nodes with their offsets given explicitly in the spatial arguments whereas the diagram at the bottom right indicates how the same nodes would be specified in a computer program where the offsets are understood implicitly.

The electric-field update equation is obtained via the finite-difference approximation of (8.5) expanded about $(m\Delta_x, n\Delta_y, (q+1/2)\Delta_t)$:

$$E_z^{q+1}[m,n] = \frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}}E_z^q[m,n] + \frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\left(\frac{\Delta_t}{\epsilon\Delta_x}\left\{H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n\right] - H_y^{q+\frac{1}{2}}\left[m-\frac{1}{2},n\right]\right\} - \frac{\Delta_t}{\epsilon\Delta_y}\left\{H_x^{q+\frac{1}{2}}\left[m,n+\frac{1}{2}\right] - H_x^{q+\frac{1}{2}}\left[m,n-\frac{1}{2}\right]\right\}\right). \quad (8.12)$$

A uniform grid is one in which the spatial step size is the same in all directions. Assuming a uniform grid such that $\Delta_x = \Delta_y = \delta$, we define the following quantities

$$C_{hxh}(m,n+1/2) = \left.\frac{1-\frac{\sigma_m\Delta_t}{2\mu}}{1+\frac{\sigma_m\Delta_t}{2\mu}}\right|_{m\delta,(n+1/2)\delta}, \quad (8.13)$$

$$C_{hxe}(m,n+1/2) = \left.\frac{1}{1+\frac{\sigma_m\Delta_t}{2\mu}}\frac{\Delta_t}{\mu\delta}\right|_{m\delta,(n+1/2)\delta}, \quad (8.14)$$

$$C_{hyh}(m+1/2,n) = \left.\frac{1-\frac{\sigma_m\Delta_t}{2\mu}}{1+\frac{\sigma_m\Delta_t}{2\mu}}\right|_{(m+1/2)\delta,n\delta}, \quad (8.15)$$

$$C_{hye}(m+1/2,n) = \left.\frac{1}{1+\frac{\sigma_m\Delta_t}{2\mu}}\frac{\Delta_t}{\mu\delta}\right|_{(m+1/2)\delta,n\delta}, \quad (8.16)$$

$$C_{eze}(m,n) = \left.\frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}}\right|_{m\delta,n\delta}, \quad (8.17)$$

$$C_{ezh}(m,n) = \left.\frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\frac{\Delta_t}{\epsilon\delta}\right|_{m\delta,n\delta}. \quad (8.18)$$

These quantities appear in the update equations and employ the following naming convention: the first letter identifies the quantity as a constant which does not vary in time (one can also think of this C as representing the word coefficient), the next two letters indicate the field being updated, and the last letter indicates the type of field this quantity multiplies. For example, C_{hxh} appears in the update equation for H_x and it multiples the previous value of the magnetic field. On the other hand, C_{hxe}, which also appears in the update equation for H_x, multiplies the electric fields.

To translate these update equations into a form that is suitable for use in a computer program, we follow the approach that was used in one dimension: explicit references to the time step are dropped and the spatial offsets are understood. As illustrated in Fig. 8.1, an H_y node is assumed to be a half spatial step further in the x direction than the corresponding E_z node with the same indices. Similarly, an H_x node is assumed to be a half spatial step further in the y direction than the corresponding E_z node with the same indices. Thus, in C, the update equations could be written

```
Hx(m, n) = Chxh(m, n) * Hx(m, n) -
   Chxe(m, n) * (Ez(m, n + 1) - Ez(m, n));
Hy(m, n) = Chyh(m, n) * Hy(m, n) +
   Chye(m, n) * (Ez(m + 1, n) - Ez(m, n));
```

```
Ez(m, n) = Ceze(m, n) * Ez(m, n) +
   Cezh(m, n) * ((Hy(m, n) - Hy(m - 1, n)) - (Hx(m, n) - Hx(m, n - 1)));
```

The reason that the "arrays" appearing in these equations start with an uppercase letter and use parentheses (instead of two pairs of brackets that would be used with traditional two-dimensional arrays in C) is because these terms are actually macros consistent with the usage described in Sec. 8.2. In order for these equations to be useful, they have to be contained within loops that cycle over the spatial indices and these loops must themselves be contained within a time-stepping loop. Additional considerations are initialization of the arrays, the introduction of energy, and termination of the grid. This issues are covered in the following sections.

8.4 TM^z Example

To illustrate the simplicity of the FDTD method in two dimensions, let us consider a simulation of a TM^z grid which is 101 nodes by 81 nodes and filled with free space. The grid will be terminated on electric field nodes which will be left at zero (so that the simulation is effectively of a rectangular resonator with PEC walls). A Ricker wavelet with 20 points per wavelength at its most energetic frequency is hardwired to the electric-field node at the center of the grid.

Before we get to the core of the code, we are now at a point where it is convenient to split the main header file into multiple header files: one defining the `Grid` structure, one defining various macros, one giving the allocation macros, and one providing the function prototypes. Not all the "`.c`" files need to include each of these header files.

The arrangement of the code is shown in Fig. 8.2. In this figure the header files `fdtd-grid1.h`, `fdtd-alloc1.h`, `fdtd-macro-tmz.h`, and `fdtd-proto1.h` are shown in a single box but they exist as four separate files (as will be shown below).

The contents of `fdtd-grid1.h` are shown in Program 8.3. The `Grid` structure, which begins on line 6, now has elements for any of the possible electric or magnetic field components as well as their associated coefficient arrays. Note that just because all the pointers are declared, they do not have to be used or point to anything useful. The `Grid` structure shown here could be used for a 1D simulation—it provides elements for everything that was needed to do a 1D simulation—but most of the pointers would be unused, i.e., those elements that pertain to anything other than a 1D simulation would be ignored.

The way we will distinguish between what different grids are being used for is by setting the "`type`" field of the grid. Note that line 4 creates a `GRIDTYPE` enumeration. This command merely serves to set the value of `oneDGrid` to zero, the value of `teZGrid` to one, and the value of `tmZGrid` to two. (The value of `threeDGrid` would be three, but we are not yet concerned with three-dimensional grids.) A `Grid` will have its `type` set to one of these values. Functions can then check the `type` and act accordingly.

Program 8.3 `fdtd-grid1.h` Contents of the header file that defines the `Grid` structure. This structure now contains pointers for each of the possible field values. However, not all these pointers would be used for any particular grid. The pointers that are meaningful would be determined by the "`type`" of the grid. The `type` takes on one of the values of the `GRIDTYPE` enumeration.

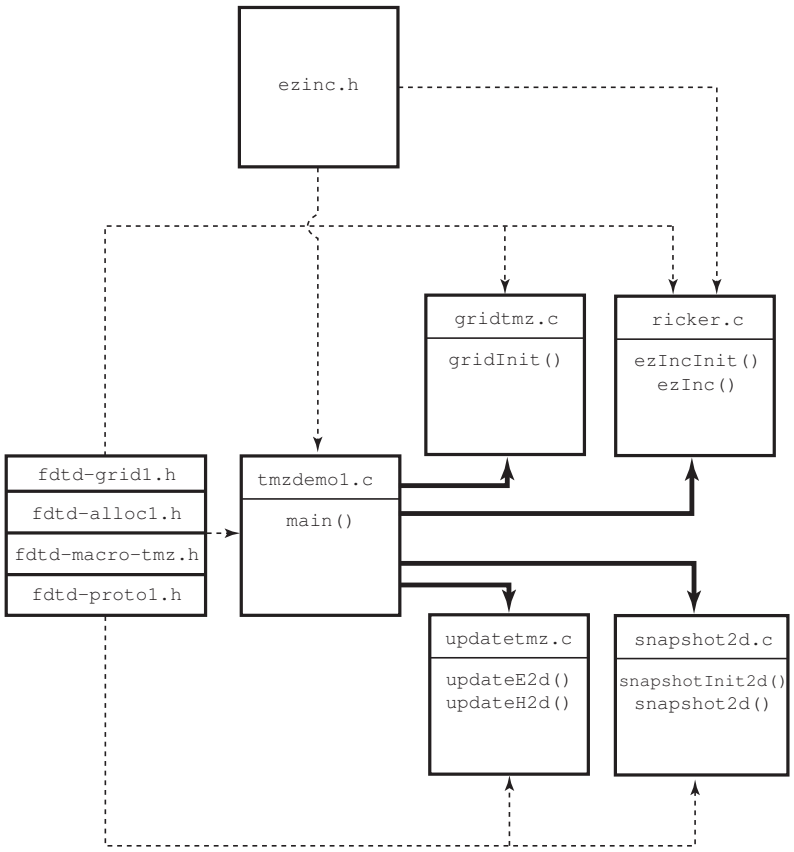


Figure 8.2: The files associated with a simple TM^z simulation with a hard source at the center of the grid. The four header files with an `fdtd-` prefix are lumped into a single box. Not all these files are included in each of the files to which this box is linked. See the code for the specifics related to the inclusion of these files.

```
#ifndef _FDTD_GRID1_H
#define _FDTD_GRID1_H

enum GRIDTYPE {oneDGrid, teZGrid, tmZGrid, threeDGrid};

struct Grid {
  double *hx, *chxh, *chxe;
  double *hy, *chyh, *chye;
  double *hz, *chzh, *chze;
  double *ex, *cexe, *cexh;
  double *ey, *ceye, *ceyh;
  double *ez, *ceze, *cezh;
  int sizeX, sizeY, sizeZ;
  int time, maxTime;
  int type;
  double cdtds;
};

typedef struct Grid Grid;

#endif
```

The contents of `fdtd-alloc1.h` are shown in Program 8.4. This header file merely provides the memory-allocation macros that have been discussed previously.

Program 8.4 `fdtd-alloc1.h` Contents of the header file that defines the memory allocation macros suitable for 1D and 2D arrays.

```
#ifndef _FDTD_ALLOC1_H
#define _FDTD_ALLOC1_H

#include <stdio.h>
#include <stdlib.h>

/* memory allocation macros */
#define ALLOC_1D(PNTR, NUM, TYPE)                                    \
    PNTR = (TYPE *)calloc(NUM, sizeof(TYPE));                        \
    if (!PNTR) {                                                     \
      perror("ALLOC_1D");                                            \
      fprintf(stderr,                                                \
          "Allocation failed for " #PNTR ".  Terminating...\n");\
      exit(-1);                                                      \
    }

```

```
17 #define ALLOC_2D(PNTR, NUMX, NUMY, TYPE)                                \
18     PNTR = (TYPE *)calloc((NUMX) * (NUMY), sizeof(TYPE));            \
19     if (!PNTR) {                                                     \
20       perror("ALLOC_2D");                                            \
21       fprintf(stderr,                                                \
22               "Allocation failed for " #PNTR ".  Terminating...\n");\
23       exit(-1);                                                      \
24     }
25
26 #endif
```

The contents of `fdtd-macro-tmz.h` are shown in Program 8.5. This file provides the macros used to access the field arrays and elements of a pointer to a `Grid` structure. Thus far all the macros we have used assumed the `Grid` pointer was called `g`. The macros provided in lines 8–35 no longer make this assumption. Instead, one specifies the name of the pointer as the first argument. To this point in our code there is no need for this added degree of freedom. We only considered code that has one pointer to a `Grid` and we have consistently named it `g`. However, as we will see when we discuss the TFSF boundary, it is convenient to have the ability to refer to different grids.

The macros in lines 39–66 do assume the `Grid` pointer is named `g`. These macros are actually defined in terms of the first set of macros where the first argument has been set to `g`. Note that although we are discussing a 2D TM^z problem, this file still provides macros that can be used for a 1D array. Again, we will see, when we implement a 2D TFSF boundary, that there are valid reasons for doing this. Since any function that is using these macros will also need to know about a `Grid` structure, line 4 ensures that the `fdtd-grid1.h` header file is also included.

Program 8.5 `fdtd-macro-tmz.h` Header file providing macros suitable for accessing the elements and arrays of either a 1D or 2D `Grid`. There are two distinct sets of macros. The first set takes an argument that speficies the name of the pointer to the `Grid` structure. The second set assumes the name of the pointer is `g`.

```
1  #ifndef _FDTD_MACRO_TMZ_H
2  #define _FDTD_MACRO_TMZ_H
3
4  #include "fdtd-grid1.h"
5
6  /* macros that permit the "Grid" to be specified */
7  /* one-dimensional grid */
8  #define Hy1G(G, M)      G->hy[M]
9  #define Chyh1G(G, M)    G->chyh[M]
10 #define Chye1G(G, M)    G->chye[M]
11
12 #define Ez1G(G, M)      G->ez[M]
13 #define Ceze1G(G, M)    G->ceze[M]
```

```
14 #define Cezh1G(G, M)    G->cezh[M]
15
16 /* TMz grid */
17 #define HxG(G, M, N)     G->hx[(M) * (SizeYG(G)-1) + (N)]
18 #define ChxhG(G, M, N) G->chxh[(M) * (SizeYG(G)-1) + (N)]
19 #define ChxeG(G, M, N) G->chxe[(M) * (SizeYG(G)-1) + (N)]
20
21 #define HyG(G, M, N)     G->hy[(M) * SizeYG(G) + (N)]
22 #define ChyhG(G, M, N) G->chyh[(M) * SizeYG(G) + (N)]
23 #define ChyeG(G, M, N) G->chye[(M) * SizeYG(G) + (N)]
24
25 #define EzG(G, M, N)     G->ez[(M) * SizeYG(G) + (N)]
26 #define CezeG(G, M, N) G->ceze[(M) * SizeYG(G) + (N)]
27 #define CezhG(G, M, N) G->cezh[(M) * SizeYG(G) + (N)]
28
29 #define SizeXG(G)        G->sizeX
30 #define SizeYG(G)        G->sizeY
31 #define SizeZG(G)        G->sizeZ
32 #define TimeG(G)         G->time
33 #define MaxTimeG(G)      G->maxTime
34 #define CdtdsG(G)        G->cdtds
35 #define TypeG(G)         G->type
36
37 /* macros that assume the "Grid" is "g" */
38 /* one-dimensional grid */
39 #define Hy1(M)      Hy1G(g, M)
40 #define Chyh1(M)    Chyh1G(g, M)
41 #define Chye1(M)    Chye1G(g, M)
42
43 #define Ez1(M)      Ez1G(g, M)
44 #define Ceze1(M)    Ceze1G(g, M)
45 #define Cezh1(M)    Cezh1G(g, M)
46
47 /* TMz grid */
48 #define Hx(M, N)   HxG(g, M, N)
49 #define Chxh(M, N) ChxhG(g, M, N)
50 #define Chxe(M, N) ChxeG(g, M, N)
51
52 #define Hy(M, N)   HyG(g, M, N)
53 #define Chyh(M, N) ChyhG(g, M, N)
54 #define Chye(M, N) ChyeG(g, M, N)
55
56 #define Ez(M, N)   EzG(g, M, N)
57 #define Ceze(M, N) CezeG(g, M, N)
58 #define Cezh(M, N) CezhG(g, M, N)
59
60 #define SizeX        SizeXG(g)
```

```
61 #define SizeY        SizeYG(g)
62 #define SizeZ        SizeZG(g)
63 #define Time         TimeG(g)
64 #define MaxTime      MaxTimeG(g)
65 #define Cdtds        CdtdsG(g)
66 #define Type         TypeG(g)
67
68 #endif   /* matches #ifndef _FDTD_MACRO_TMZ_H */
```

Finally, the contents of `fdtd-proto1.h` are shown in Program 8.6. This file provides the prototypes for the various functions associated with the simulation. Since a pointer to a `Grid` appears as an argument to these functions, any file that includes this header will also need to include `fdtd-grid1.h` as is done in line 4.

Program 8.6 `fdtd-proto1.h` Header file providing the function prototypes.

```
1  #ifndef _FDTD_PROTO1_H
2  #define _FDTD_PROTO1_H
3
4  #include "fdtd-grid1.h"
5
6  /* Function prototypes */
7  void gridInit(Grid *g);
8
9  void snapshotInit2d(Grid *g);
10 void snapshot2d(Grid *g);
11
12 void updateE2d(Grid *g);
13 void updateH2d(Grid *g);
14
15 #endif
```

The file `tmzdemo1.c`, which contains the `main()` function, is shown in Program 8.7. The program begins with the inclusion of the necessary header files. Note that only three of the four `fdtd-` header files are explicitly included. However, both the header files `fdtd-macro-tmz.h` and `fdtd-proto1.h` ensure that the "missing" file, `fdtd-grid1.h`, is included.

Fields are introduced into the grid by hardwiring the value of an electric-field node as shown in line 22. Because the source function is used in `main()`, the header file `ezinc.h` had to be included in this file. Other than those small changes, this program looks similar to many of the 1D programs which we have previously considered.

Program 8.7 `tmzdemo1.c` FDTD implementation of a TM^z grid with a Ricker wavelet source at the center of the grid. No ABC have been implemented so the simulation is effectively of a

resonator.

```
1  /* TMz simulation with Ricker source at center of grid. */
2
3  #include "fdtd-alloc1.h"
4  #include "fdtd-macro-tmz.h"
5  #include "fdtd-proto1.h"
6  #include "ezinc.h"
7
8  int main()
9  {
10   Grid *g;
11
12   ALLOC_1D(g, 1, Grid); // allocate memory for Grid
13
14   gridInit(g);         // initialize the grid
15   ezIncInit(g);
16   snapshotInit2d(g);   // initialize snapshots
17
18   /* do time stepping */
19   for (Time = 0; Time < MaxTime; Time++) {
20     updateH2d(g);      // update magnetic field
21     updateE2d(g);      // update electric field
22     Ez(SizeX / 2, SizeY / 2) = ezInc(Time, 0.0); // add a source
23     snapshot2d(g);     // take a snapshot (if appropriate)
24   } // end of time-stepping
25
26   return 0;
27 }
```

The contents of `gridtmz.c`, which contains the grid initialization function `gridInit()`, is shown in Program 8.8. On line 9 the type of grid is defined. This is followed by statements which set the size of the grid, in both the x and y directions, the duration of the simulation, and the Courant number. Then, on lines 15 through 23, space is allocated for the field arrays and their associated coefficients array. Note that although the E_z array is `SizeX`×`SizeY`, H_x is `SizeX`×(`SizeY` − 1), and H_y is (`SizeX` − 1)×`SizeY`. The remainder of the program merely sets the coefficient arrays. Here there is no need to include the header file `fdtd-proto1.h` since this function does not call any of the functions listed in that file.

Program 8.8 `gridtmz.c` Grid initialization function for a TM^z simulation. Here the grid is simply homogeneous free space.

```
1  #include "fdtd-macro-tmz.h"
2  #include "fdtd-alloc1.h"
```

```
3  #include <math.h>
4
5  void gridInit(Grid *g) {
6    double imp0 = 377.0;
7    int mm, nn;
8
9    Type = tmZGrid;
10   SizeX = 101;              // x size of domain
11   SizeY = 81;               // y size of domain
12   MaxTime = 300;            // duration of simulation
13   Cdtds = 1.0 / sqrt(2.0); // Courant number
14
15   ALLOC_2D(g->hx,   SizeX, SizeY - 1, double);
16   ALLOC_2D(g->chxh, SizeX, SizeY - 1, double);
17   ALLOC_2D(g->chxe, SizeX, SizeY - 1, double);
18   ALLOC_2D(g->hy,   SizeX - 1, SizeY, double);
19   ALLOC_2D(g->chyh, SizeX - 1, SizeY, double);
20   ALLOC_2D(g->chye, SizeX - 1, SizeY, double);
21   ALLOC_2D(g->ez,   SizeX, SizeY, double);
22   ALLOC_2D(g->ceze, SizeX, SizeY, double);
23   ALLOC_2D(g->cezh, SizeX, SizeY, double);
24
25   /* set electric-field update coefficients */
26   for (mm = 0; mm < SizeX; mm++)
27     for (nn = 0; nn < SizeY; nn++) {
28       Ceze(mm, nn) = 1.0;
29       Cezh(mm, nn) = Cdtds * imp0;
30     }
31
32   /* set magnetic-field update coefficients */
33   for (mm = 0; mm < SizeX; mm++)
34     for (nn = 0; nn < SizeY - 1; nn++) {
35       Chxh(mm, nn) = 1.0;
36       Chxe(mm, nn) = Cdtds / imp0;
37     }
38
39   for (mm = 0; mm < SizeX - 1; mm++)
40     for (nn = 0; nn < SizeY; nn++) {
41       Chyh(mm, nn) = 1.0;
42       Chye(mm, nn) = Cdtds / imp0;
43     }
44
45   return;
46 }
```

The functions for updating the fields are contained in the file `updatetmz.c` which is shown in Program 8.9. In line 7 the `Type` is checked (i.e., `g->type` is checked). If it is `oneDGrid`

then only the H_y field is updated and it only has a single spatial index. If the grid is not a 1D grid, it is assumed to be a TM^z grid. Thus, starting on line 12, H_x and H_y are updated and they now have two spatial indices.

Program 8.9 `updatetmz.c` Functions to update the fields. Depending on the type of grid, the fields can be treated as either one- or two-dimensional.

```
1  #include "fdtd-macro-tmz.h"
2
3  /* update magnetic field */
4  void updateH2d(Grid *g) {
5    int mm, nn;
6
7    if (Type == oneDGrid) {
8      for (mm = 0; mm < SizeX - 1; mm++)
9        Hy1(mm) = Chyh1(mm) * Hy1(mm)
10         + Chye1(mm) * (Ez1(mm + 1) - Ez1(mm));
11   } else {
12     for (mm = 0; mm < SizeX; mm++)
13       for (nn = 0; nn < SizeY - 1; nn++)
14         Hx(mm, nn) = Chxh(mm, nn) * Hx(mm, nn)
15           - Chxe(mm, nn) * (Ez(mm, nn + 1) - Ez(mm, nn));
16
17     for (mm = 0; mm < SizeX - 1; mm++)
18       for (nn = 0; nn < SizeY; nn++)
19         Hy(mm, nn) = Chyh(mm, nn) * Hy(mm, nn)
20           + Chye(mm, nn) * (Ez(mm + 1, nn) - Ez(mm, nn));
21   }
22
23   return;
24 }
25
26 /* update electric field */
27 void updateE2d(Grid *g) {
28   int mm, nn;
29
30   if (Type == oneDGrid) {
31     for (mm = 1; mm < SizeX - 1; mm++)
32       Ez1(mm) = Ceze1(mm) * Ez1(mm)
33         + Cezh1(mm) * (Hy1(mm) - Hy1(mm - 1));
34   } else {
35     for (mm = 1; mm < SizeX - 1; mm++)
36       for (nn = 1; nn < SizeY - 1; nn++)
37         Ez(mm, nn) = Ceze(mm, nn) * Ez(mm, nn) +
38           Cezh(mm, nn) * ((Hy(mm, nn) - Hy(mm - 1, nn)) -
39                           (Hx(mm, nn) - Hx(mm, nn - 1)));
```

```
40    }
41
42    return;
43  }
```

The function for updating the electric field, `updateE2d()`, only is responsible for updating the E_z field. However, as shown in line 30, it still must check the grid type. If this is a 1D grid, E_z only has a single spatial index and only depends on H_y. If it is not a 1D grid, it is assumed to be a TM^z grid and E_z now depends on both H_x and H_y.

The function to implement the Ricker wavelet is shown in Program 8.10. The header file `ezinc.h` is virtually unchanged from Program 4.16. The one minor change is that instead of including `fdtd2.h`, now the file `fdtd-macro-tmz.h` is included. Thus `ezinc.h` is not shown. The initialization function `ezIncInit()` prompts the user to enter the points per wavelength at which the Ricker wavelet has maximum energy. In line 10 it also makes a local copy of the Courant number (since the `Grid` is not passed to the `ezInc()` function and would not otherwise know this value).

Program 8.10 `ricker.c` Function to implement a Ricker wavelet. This is a traveling-wave version of the function so `ezInc()` takes arguments of both time and space.

```
1  #include "ezinc.h"
2
3  static double cdtds, ppw = 0;
4
5  /* initialize source-function variables */
6  void ezIncInit(Grid *g){
7
8    printf("Enter the points per wavelength for Ricker source: ");
9    scanf(" %lf", &ppw);
10   cdtds = Cdtds;
11   return;
12 }
13
14 /* calculate source function at given time and location */
15 double ezInc(double time, double location) {
16   double arg;
17
18   if (ppw <= 0) {
19     fprintf(stderr,
20        "ezInc: ezIncInit() must be called before ezInc.\n"
21        "       Points per wavelength must be positive.\n");
22     exit(-1);
23   }
24
25   arg = M_PI * ((cdtds * time - location) / ppw - 1.0);
```

```
26   arg = arg * arg;
27
28   return (1.0 - 2.0 * arg) * exp(-arg);
29 }
```

Finally, `snapshot2d.c` is shown in Program 8.11. The function `snapshotInit2d()` obtains information from the user about the output that is desired. The goal is to write the data so that the electric field can be visualized over the entire 2D computational domain.

Program 8.11 `snapshot2d.c` Function to record the 2D field to a file. The data is stored as binary data.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "fdtd-macro-tmz.h"
4
5  static int temporalStride = -2, frame = 0, startTime,
6    startNodeX, endNodeX, spatialStrideX,
7    startNodeY, endNodeY, spatialStrideY;
8  static char basename[80];
9
10 void snapshotInit2d(Grid *g) {
11
12   int choice;
13
14   printf("Do you want 2D snapshots? (1=yes, 0=no) ");
15   scanf("%d", &choice);
16   if (choice == 0) {
17     temporalStride = -1;
18     return;
19   }
20
21   printf("Duration of simulation is %d steps.\n", MaxTime);
22   printf("Enter start time and temporal stride: ");
23   scanf(" %d %d", &startTime, &temporalStride);
24   printf("In x direction grid has %d total nodes"
25          " (ranging from 0 to %d).\n", SizeX, SizeX - 1);
26   printf("Enter first node, last node, and spatial stride: ");
27   scanf(" %d %d %d", &startNodeX, &endNodeX, &spatialStrideX);
28   printf("In y direction grid has %d total nodes"
29          " (ranging from 0 to %d).\n", SizeY, SizeY - 1);
30   printf("Enter first node, last node, and spatial stride: ");
31   scanf(" %d %d %d", &startNodeY, &endNodeY, &spatialStrideY);
32   printf("Enter the base name: ");
33   scanf(" %s", basename);
```

```
34
35    return;
36  }
37
38  void snapshot2d(Grid *g) {
39    int mm, nn;
40    float dim1, dim2, temp;
41    char filename[100];
42    FILE *out;
43
44    /* ensure temporal stride set to a reasonable value */
45    if (temporalStride == -1) {
46      return;
47    } if (temporalStride < -1) {
48      fprintf(stderr,
49        "snapshot2d: snapshotInit2d must be called before snapshot.\n"
50        "            Temporal stride must be set to positive value.\n");
51      exit(-1);
52    }
53
54    /* get snapshot if temporal conditions met */
55    if (Time >= startTime &&
56        (Time - startTime) % temporalStride == 0) {
57      sprintf(filename, "%s.%d", basename, frame++);
58      out = fopen(filename, "wb");
59
60      /* write dimensions to output file --
61       * express dimensions as floats */
62      dim1 = (endNodeX - startNodeX) / spatialStrideX + 1;
63      dim2 = (endNodeY - startNodeY) / spatialStrideY + 1;
64      fwrite(&dim1, sizeof(float), 1, out);
65      fwrite(&dim2, sizeof(float), 1, out);
66
67      /* write remaining data */
68      for (nn = endNodeY; nn >= startNodeY; nn -= spatialStrideY)
69        for (mm = startNodeX; mm <= endNodeX; mm += spatialStrideX) {
70          temp = (float)Ez(mm, nn); // store data as a float
71          fwrite(&temp, sizeof(float), 1, out); // write the float
72        }
73
74      fclose(out);  // close file
75    }
76
77    return;
78  }
```

Similar to the snapshot code in one dimension, the E_z field is merely recorded (in binary

format) to a file at the appropriate time-steps. It is up to some other program or software to render this data in a suitable way. In order to understand what is happening in the two-dimensional grid, it is extremely helpful to display the fields in a manner that is consistent with the underlying two-dimensional format. This can potentially be quite a bit of data. To deal with it efficiently, it is often best to store the data directly in binary format, which we will refer to as "raw" data. In line 58 the output file is opened as a binary file (hence "`b`" which appears in the second argument of the call to `fopen()`).

The arrays being used to store the fields are doubles. However, storing a complete double can be considered overkill when it comes to generating graphics. We certainly do not need 15 digits of precision when viewing the fields. Instead of writing doubles, the output is converted to a float. (By using floats instead of doubles, the file size is reduced by a factor of two.) Within each output data file, first the dimensions of the array are written, as floats, as shown in lines 64 and 65. After that, starting in line 68 of Program 8.11, two nested loops are used to write each element of the array. Note that the elements are not written in what might be considered a standard way. The elements are written consistent with how you would read a book in English: from left to right, top to bottom. As mentioned previously, this is not the most efficient way to access arrays, but there are some image-processing tools which prefer that data be stored this way.

Once this data is generated, there are several ways in which the data can be displayed. It is possible to read the data directly using Matlab and even create an animation of the field. Appendix C presents a Matlab function that can be used to generate a movie from the data generated by Program 8.7.

After compiling Program 8.7 in accordance with all the files shown in Fig. 8.2, let us assume the executable is named `tmzdemo1`. The following shows a typical session where this program is run on a UNIX system (where the executable is entered at the command-line prompt of ">"). The user's entries are shown in bold.

```
> tmzdemo1
Enter the points per wavelength for Ricker source: 20
Do you want 2D snapshots? (1=yes, 0=no) 1
Duration of simulation is 300 steps.
Enter start time and temporal stride: 10 10
In x direction grid has 101 total nodes (ranging from 0 to 100).
Enter first node, last node, and spatial stride: 0 100 1
In y direction grid has 81 total nodes (ranging from 0 to 80).
Enter first node, last node, and spatial stride: 0 80 1
Enter the base name: sim
```

In this case the user set the Ricker wavelet to have 20 points per wavelength at the most energetic frequency. Snapshots were generated every 10 time-steps beginning at the 10th time-step. The snapshots were taken of the entire computational domain since the start- and stop-points were the first and last nodes in the x and y directions and the spatial stride was unity. The snapshots had a common base name of `sim`.

Figure 8.3 shows three snapshots of the electric field that are generated by Program 8.7. These images are individual frames generated by the code presented in Appendix C (the frames are in color when viewed on a suitable output device). These frames correspond to snapshots taken at time-steps 30, 70, and 110. Logarithmic scaling is used so that the maximum normalized value of one corresponds to the color identified as zero on the color-bar to the right of each image. A

normalization value of unity was used for these images. Three decades are displayed so that the minimum visible normalized field is 10^{-3}. This value is shown with a color corresponding to -3 on the color-bar (any values less than the minimum are also displayed using this same color).

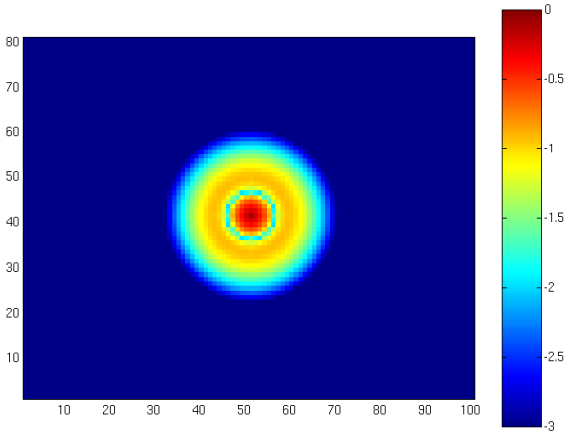
At time-step 30, the field is seen to be radiating concentrically away from the source at the center of the grid. At time-step 70 the field is just starting to reach the top and bottom edges of the computational domain. Since the electric-field nodes along the edge of the computational domain are not updated (due to these nodes lacking a neighboring magnetic-field node in their update equations), these edges behave as PEC boundaries. Hence the field is reflected back from these walls. The reflection of the field in clearly evident at time-step 110. As the simulation progresses, the field bounces back and forth. (The field at a given point can be recorded and then Fourier transformed. The peaks in the transform correspond to the resonant modes of this particular structure.)

To model an infinite domain, the second-order ABC discussed in Sec. 6.6 can be applied to every electric field node on the boundary. In one dimension the ABC needed to be applied to only two nodes. In two dimensions, there would essentially be four lines of nodes to which the ABC must be applied: nodes along the left, right, top, and bottom. However, in all cases the form of the ABC is the same. For a second-order ABC, a node on the boundary depends on two interior nodes as well as the field at the boundary and those same two interior nodes at two previous time steps. As before, the old values would have to be stored in supplementary arrays—six old values for each node on the boundary. This is accomplished fairly easily by extrapolating the 1D case so that there are now four storage arrays (one for the left, right, top, and bottom). These would be three-dimensional arrays. In addition to two indices which indicate displacement from the edge (i.e., displacement into the interior) and the time step, there would be a third index to indicate displacement *along* the edge. So, for example, this third index would specify the particular node along the top or bottom (and hence would vary between 0 and "`SizeX - 1`") or the node along the left or right (and hence would vary between 0 and "`SizeY - 1`").

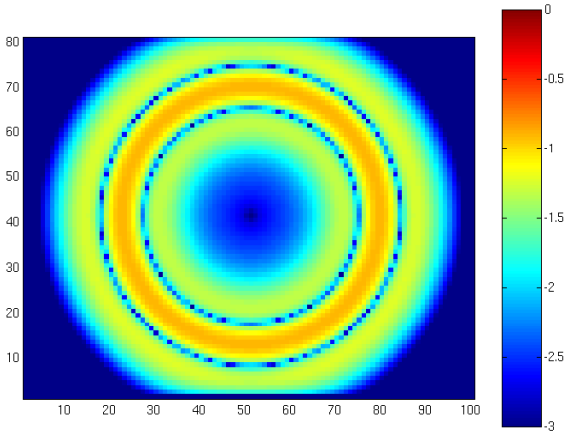
For nodes in the corner of the computational domain, there is some ambiguity as to which nodes are the neighboring "interior" nodes which should be used by the ABC. However, the corner nodes never couple back to the interior and hence it does not matter what one does with these nodes. They can be left zero or assumed to contain meaningless numbers and that will not affect the values in the interior of the grid. The magnetic fields that are adjacent to corner nodes are affected by the values of the field in the corners. However, these nodes themselves are not used be any other nodes in their updates. The electric fields which are adjacent to these magnetic fields are updated using the ABC; they ignore the field at the neighboring magnetic-field nodes. Therefore no special consideration will be given to resolving the corner ambiguity.

8.5 The TFSF Boundary for TM^z Polarization

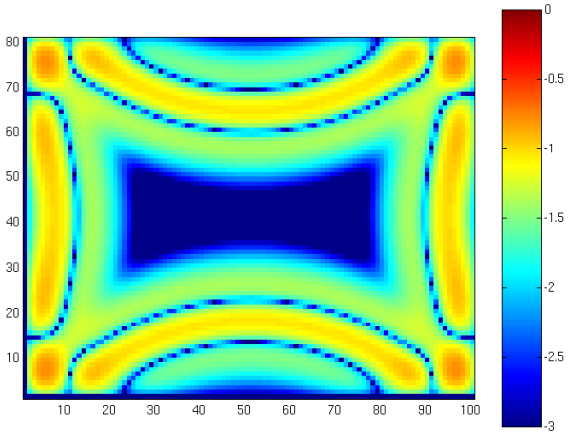
For a distant source illuminating a scatterer, it is not feasible to discretize the space surrounding the source, discretize the space between the source and the scatterer, and discretize the space surrounding the scatterer. Even if a large enough computer could be obtained that was capable of storing all that discretized space, one simply would not want to use the FDTD grid to propagate the field from the source to the scatterer. Such an endeavor would be slow, incredibly inefficient, and suffer from needless numerical artifacts. Instead, one should discretize the space surrounding



(a)



(b)



(c)

Figure 8.3: Display of E_z field generated by Program 8.7 at time steps (a) 30, (b) 70, and (c) 110. A Ricker source with 20 points per wavelength at its most energetic frequency is hard-wired to the E_z node at the center of the grid.

the scatterer and introduce the incident field via a total-field/scattered-field boundary. When the source is distant from the scatterer, the incident field is nearly planar and thus we will restrict consideration to incident plane waves.

Section 3.10 showed how the TFSF concept could be implemented in a one-dimensional problem. The TFSF boundary separated the grid into two regions: a total-field (TF) region and a scattered-field (SF) region. There were two nodes adjacent to this boundary. One was in the SF region and depended on a node in the TF region. The other was in the TF region and depended on a node in the SF region. To obtain self-consistent update equations, when updating nodes in the TF region, one must use the total field which pertains at the neighboring nodes. Conversely, when updating nodes in the SF region, one must use the scattered field which pertains at neighboring nodes. In one dimension, the two nodes adjacent to the boundary must have the incident field either added to or subtracted from the field which exists at their neighbor on the other side of the boundary. Thus, in one dimension we required knowledge of the incident field at two locations for every time step.

In two dimensions, the grid is again divided into a TF region and a SF region. In this case the boundary between the two regions is no longer a point. Figure 8.4 shows a TM^z grid with a rectangular TFSF boundary. (The boundary does not have to be rectangular, but the implementation details are simplest when the boundary has straight sides and hence we will restrict ourselves to TFSF boundaries which are rectangular.) In this figure the TF region is enclosed within the TFSF boundary which is drawn with a dashed line. The SF region is any portion of the grid that is outside this boundary. Nodes that have a neighbor on the other side of the boundary are enclosed in a solid rectangle with rounded corners. Note that these encircled nodes are tangential to the TFSF boundary (we consider the E_z field, which points out of the page, to be tangential to the boundary if we envision the boundary extending into the third dimension). The fields that are normal to the boundary, such as the H_y nodes along the top and bottom of the TFSF boundary, do not have neighbors which are across the boundary (even though the field could be considered adjacent to the boundary).

In the implementation used here, the TF region is defined by the indices of the "first" and "last" electric-field nodes which are in the TF region. These nodes are shown in Fig. 8.4 where the "first" node is the one in the lower left corner and the "last" one is in the upper right corner. Note that electric fields and magnetic fields with the same indices are not necessarily on the same side of the boundary. For example, the E_z nodes on the right side of the TF region have one of their neighboring H_y nodes in the SF region. This is true despite the fact that these H_y nodes share the same x-index as the E_z nodes.

Further note that in this particular construction of a TFSF boundary, the electric fields tantential to the TFSF boundary are always in the TF region. These nodes will have at least one neighboring magnetic field node that is in the SF region. Thus, the correction necessary to obtain a consistent update of these electric field nodes would involve adding the incident field the neighboring magnetic fields on the other side of the TFSF boundary. Conversely, the magnetic field nodes that are tangential to the TFSF boundary are always in the SF region. These nodes will have one neighboring electric field node that is in the TF region. Thus, the correction necessary to obtain a consistent update of these magnetic field nodes would involve subtracting the incident field from the electric field node on the other side of the TFSF boundary.

As in the one-dimensional case, to implement the TFSF method, one must know the incident field at every node which has a neighbor on the other side of the TFSF boundary. The incident

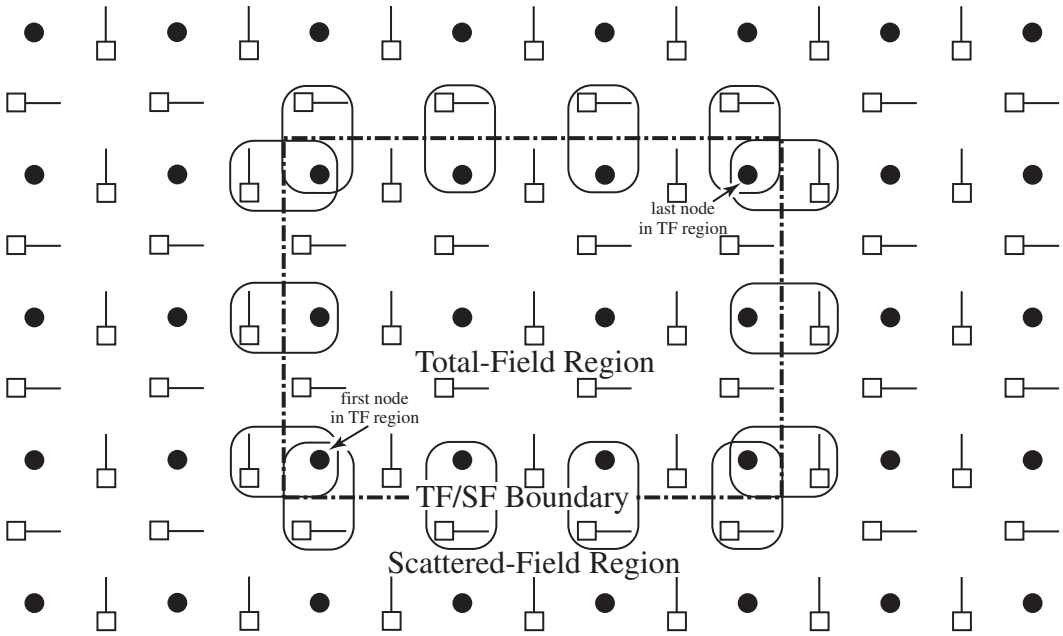


Figure 8.4: Depiction of a total-field/scattered-field boundary in a TM^z grid. The size of the TF region is defined by the indices of the first and last electric field nodes which are within the region. Note that at the right-hand side of the boundary the H_y nodes with the same x-index (i.e., the same "m" index) as the "last" node will be in the SF region. Similarly, at the top of the grid, H_x nodes with the same y-index as the last node will be in the SF region. Therefore one must pay attention to the field component as well as the indices to determine if a node is in the SF or TF region.

field must be known at all these points and for every time-step. In Section 3.10 analytic expressions were used for the incident field, i.e., the expressions that describes propagation of the incident field in the continuous world. However, the incident field does not propagate the same way in the FDTD grid as the continuous world (except in the special case of one-dimensional propagation with a Courant number of unity). Therefore, if the continuous-world expressions were used for the incident field, there would be a mismatch between the fields in the grid and the fields given by the continuous-world expressions. This mismatch would cause fields to leak across the boundary. Another drawback to using the continuous-world expressions is that they typically involve a transcendental function (such as a trigonometric function or an exponential). Calculation of these functions is somewhat computationally expensive—at least compared to a few simple algebraic calculations. If the transcendental functions have to be calculated at numerous points for every time-step, this can impose a potentially significant computational cost. Fortunately, provided the direction of the incident-field propagation coincides with one of the axes of the grid, there is a way to ensure that the incident field exactly matches the way in which the incident field propagates in the two-dimensional FDTD grid. Additionally, the calculation of the incident field can be done efficiently.

The trick to calculating the incident field is to perform an auxiliary one-dimensional FDTD simulation which calculates the incident field. This auxiliary simulation uses the same Courant number and material parameters as pertain in the two-dimensional grid but is otherwise completely separate from the two-dimensional grid. The one-dimensional grid is merely used to find the incident fields needed to implement the TFSF boundary. (Each E_z and H_y node in the 1D grid can be thought as providing E_z^{inc} and H_y^{inc}, respectively, at the appropriate point in space-time as dictated by the discretization and time-stepping.)

Figure 8.5 shows the auxiliary 1D grid together with the 2D grid. The base of the vertical arrows pointing from the 1D grid to the 2D grid indicate the nodes in the 1D grid from which the nodes in the 2D grid obtain the incident field (only nodes in the 2D grid adjacent to the TFSF boundary require knowledge of the incident field). Since the incident field propagates in the $+x$ direction, there is no incident H_x field. Hence nodes that depend on an H_x node on the other side of the TFSF boundary do not need to be corrected since $H_x^{\text{inc}} = 0$.

Despite the representation in Fig. 8.5, the 1D grid does not need to be the same width as the 2D grid, but it must be at least as long as necessary to provide the incident field for all the nodes tangential to the TFSF boundary (i.e., it must be large enough to provide the values associated with the base of each of the vertical arrows shown in Fig. 8.5). Additionally, the 1D grid must include a source on the left and the right side of the grid must be suitably terminated so that the incident field does not reflect back. Here we will assume fields are introduced into the 1D grid via a hard source at the left end.

Using an auxiliary 1D grid, the TFSF boundary could be realized as follows. First, outside of the time-stepping loop, a function would be called to initialize the TFSF code. This initialization would allocate arrays of the necessary size for the 1D auxiliary grid and set all the necessary constants. Then, within the time-stepping loop, the following steps would be taken (where we use the additional subscripts 1D and 2D to distinguish between arrays associated with the 1D and 2D grids):

1. Update the magnetic fields in the two-dimensional grid using the usual update equations (i.e., do not account for the existence of TFSF boundary): $H_{x2\text{D}}^{q-\frac{1}{2}} \Rightarrow H_{x2\text{D}}^{q+\frac{1}{2}}$ and $H_{y2\text{D}}^{q-\frac{1}{2}} \Rightarrow H_{y2\text{D}}^{q+\frac{1}{2}}$.

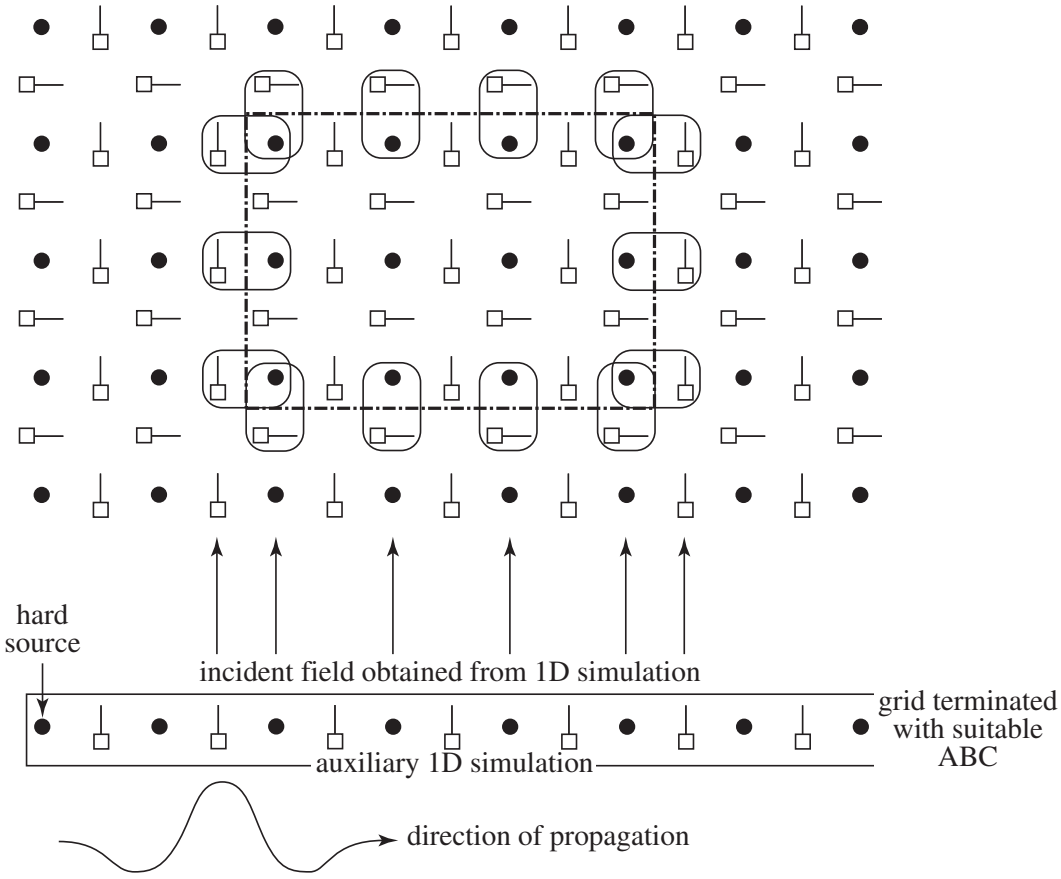


Figure 8.5: A one-dimensional auxiliary grid is used to calculate the incident field which is assumed to be propagating in the $+x$ direction. The vertical arrows indicate the nodes whose values are needed to implement the TFSF boundary. The incident H_x field is zero and hence no correction is needed in association with E_z nodes that have a neighboring H_x node on the other side of the boundary. The 1D grid is driven by a hard source at the left side. The 1D grid must be suitably terminated at the right side to model an infinite domain. The size of the 1D grid is somewhat independent of the size of the 2D grid—it must be large enough to provide incident field associated with each of the vertical arrows shown above but otherwise may be larger or smaller than the overall width of the 2D grid.

2. Call a function to make all calculations and corrections associated with the TFSF boundary:
   (a) Correct the two-dimensional magnetic fields tangential to the TFSF boundary using the incident electric field from the one-dimensional grid, i.e., using $E^q_{z1\mathrm{D}}$.
   (b) Update the magnetic field in the one-dimensional grid: $H^{q-\frac{1}{2}}_{y1\mathrm{D}} \Rightarrow H^{q+\frac{1}{2}}_{y1\mathrm{D}}$.
   (c) Update the electric field in the one-dimensional grid: $E^q_{z1\mathrm{D}} \Rightarrow E^{q+1}_{z1\mathrm{D}}$.
   (d) Correct the electric field in the two-dimensional grid using the incident magnetic field from the one-dimensional grid, i.e., using $H^{q+\frac{1}{2}}_{y1\mathrm{D}}$. (Since there is no $H_{x1\mathrm{D}}$ in this particular case with grid-aligned propagation, no correction is necessary in association with E_z nodes that have a neighboring H_x node on the other side of the TFSF boundary.)
3. Update the electric field in the two-dimensional grid using the usual update equations (i.e., do not account for the existence of TFSF boundary): $E^q_{z2\mathrm{D}} \Rightarrow E^{q+1}_{z2\mathrm{D}}$.

8.6 TM^z TFSF Boundary Example

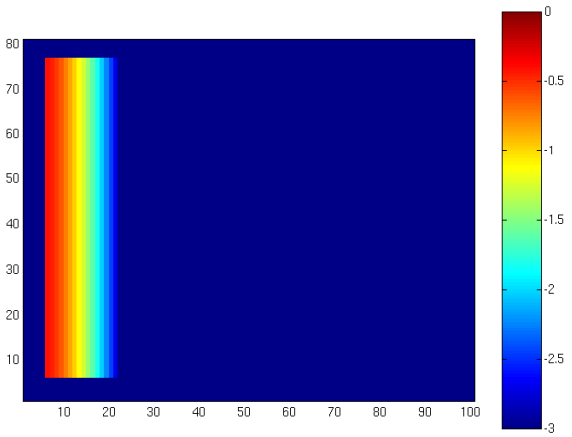
Figure 8.6 shows three snapshots of a computational domain that incorporates a TFSF boundary. The size of the grid is 101 nodes wide and 81 nodes high. The incident field is a Ricker wavelet with 30 points per wavelength at its most energetic frequency. The indices for the first electric-field node in the TF region are $(5, 5)$ and the indices of the last node in the TF region are $(95, 75)$. There is no scatterer present and hence there are no fields visible in the SF region.

In Fig. 8.6(a) the incident field is seen to have entered the left side of the TF region. There is an abrupt discontinuity in the field as one cross the TFSF boundary. This discontinuity is visible to the left of the TF region as well as along a portion of the top and bottom of the region. In Fig. 8.6(b) the pulse is nearly completely within the TF region. In Fig. 8.6(c) the incident pulse has encountered the right side of the TF region. At this point the incident field seemingly disappears! The corrections to the fields at the right side of the boundary are such that the incident field does not escape the TF region.

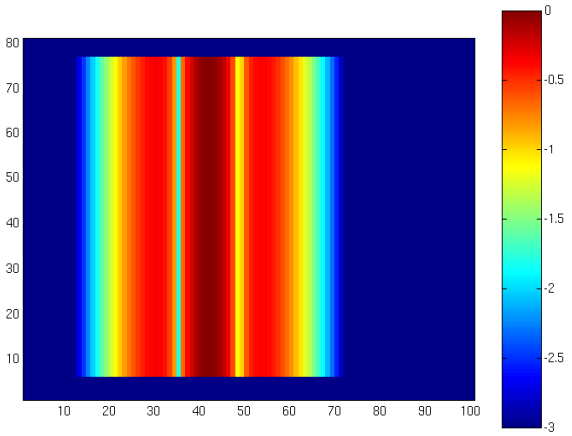
Figure 8.7 shows three snapshots of a computational domain that is similar to the one shown in Fig. 8.6. The only difference is that a PEC plate has been put into the grid. The plate is realized by setting to zero the E_z nodes along a vertical line. This line of nodes is offset 20 cells from the left side of the computational domain and runs vertically from 20 cells from the bottom of the domain to 20 cells from the top. (The way in which one models a PEC in 2D grids will be discussed further in Sec. 8.8.)

Second-order ABC's are used to terminate the grid. (In Fig. 8.6 the fields were normalized to 1.0. In Fig. 8.7 they have been normalized to 2.0.)

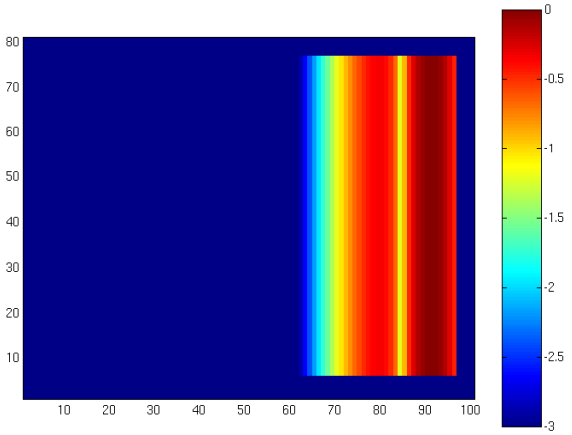
In Fig. 8.7(a) the incident field has just barely reached the plate. There is no scattering evident yet and hence no scattered fields are visible in the SF region. In Fig. 8.7(b) the interaction of the field with the plate is obvious. One can see how the fields have diffracted around the edges of the plate. As can be seen, the field scattered from the plate has had time to propagate into the SF region. Figure 8.7(c) also shows the non-zero field in the SF region (together with the total field throughout the TF region). The ABC's must absorb the scattered field, but they do not have to contend with the incident field since, as shown in Fig. 8.6, the incident field never escapes the TF



(a)

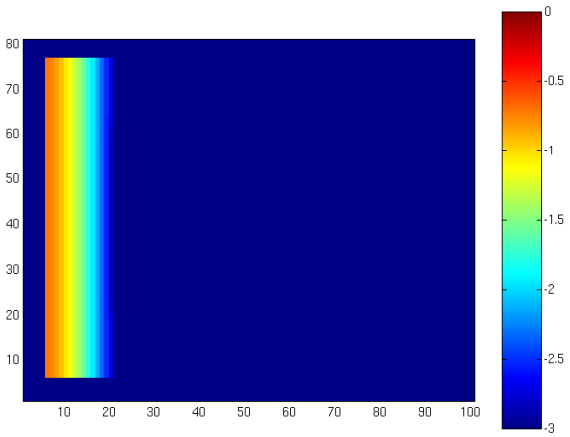


(b)

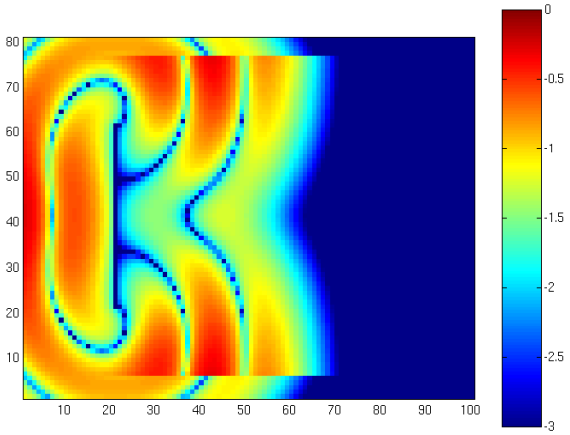


(c)

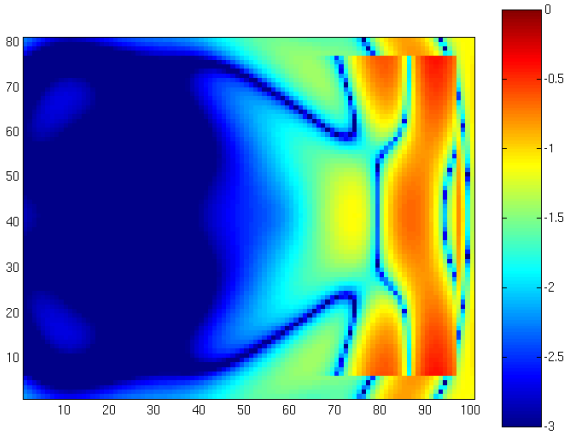
Figure 8.6: Display of E_z field in a computational domain employing a TFSF boundary. Snapshots are taken at time-steps (a) 30, (b) 100, and (c) 170. The pulsed, plane-wave source corresponds to a Ricker wavelet with 30 points per wavelength at its most energetic frequency.



(a)



(b)



(c)

Figure 8.7: Display of E_z field in a computational domain employing a TFSF boundary. There is a PEC vertical plate which is realized by setting to zero the E_z field over a lines that is 41 cells high and 20 cells from the left edge of the computational domain. Snapshots are taken at time steps (a) 30, (b) 100, and (c) 170. A second-order ABC is used to terminate the grid.

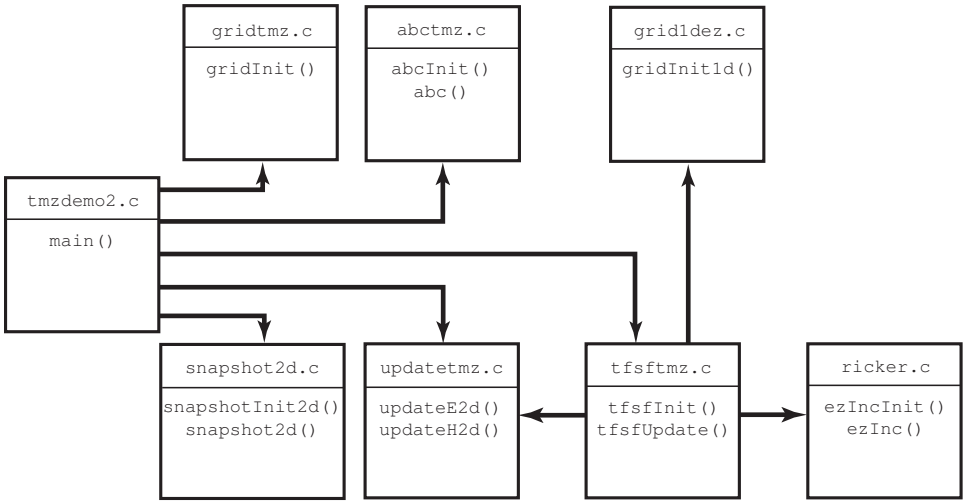


Figure 8.8: Organization of files associated with a TM^z simulation that employs a TFSF boundary and a second-order ABC. The header files are not shown.

region (but, of course, the scattered field at any point along the edge of the computational domain could be as large or larger than the incident field—it depends on how the scatterer scatters the field).

The organization of code used to generate the results shown in Fig. 8.7 is depicted in Fig. 8.8. The header files are not shown. The contents of the files `updatetmz.c`, `ricker.c`, and `snapshot2d.c` are unchanged from the previous section (refer to Programs 8.9, 8.10, and 8.11, respectively). The file `gridtmz.c` has changed only slightly from the code shown in Program 8.8 in that a line of electric-field update coefficients are now set to zero corresponding to the location of the PEC. Since this change is so minor, this file is not presented here. The header files `fdtd-alloc1.h`, `fdtd-grid1.h` and `fdtd-macro-tmz.h` are also unchanged from the previous section (refer to Programs 8.4, 8.3, and 8.5).

The contents of `tmzdemo2.c` are shown in Program 8.12. This program differs from Program 8.7 only in the call to the TFSF and ABC functions. Also, a different prototype header file is included. These difference are shown in bold.

Program 8.12 `tmzdemo2.c` Program to perform a TM^z simulation where the field is introduced via a TFSF boundary and the grid is terminated with a second-order ABC. The differences between this code and Program 8.7 are shown in bold.

```
1 /* TMz simulation with a TFSF boundary and a second-order ABC. */
2
3 #include "fdtd-alloc1.h"
4 #include "fdtd-macro-tmz.h"
5 #include "fdtd-proto2.h"
6
7 int main()
```

```
8  {
9    Grid *g;
10
11   ALLOC_1D(g, 1, Grid); // allocate memory for grid
12   gridInit(g);          // initialize 2D grid
13
14   abcInit(g);           // initialize ABC
15   tfsfInit(g);          // initialize TFSF boundary
16   snapshotInit2d(g);    // initialize snapshots
17
18   /* do time stepping */
19   for (Time = 0; Time < MaxTime; Time++) {
20     updateH2d(g);       // update magnetic fields
21     tfsfUpdate(g);      // apply TFSF boundary
22     updateE2d(g);       // update electric fields
23     abc(g);             // apply ABC
24     snapshot2d(g);      // take a snapshot (if appropriate)
25   } // end of time-stepping
26
27   return 0;
28 }
```

After initialization of the 2D grid in line 11, the ABC, TFSF, and snapshot functions are initialized. Time-stepping begins in line 19. Within the time-stepping loop, first the magnetic fields are updated. As already mentioned, the function `updateH2d()` is unchanged from before. We merely pass to it the the `Grid` pointer `g`. Next, the function `tfsfUpdate()` is used to update the fields adjacent to the TFSF boundary. This function takes (the 2D `Grid` pointer) `g` as an argument. As we will see, the TFSF function also keeps track of an auxiliary 1D that is completely hidden from `main()`. The electric fields are then updated, the ABC is applied, and a snapshot is generated (if the time-step is appropriate).

The header file `fdtd-proto2.h` is shown in Program 8.13. The only substantial changes from Program 8.6 are the addition of the prototypes for the TFSF, ABC functions, and a function used to initialize the 1D grid.

Program 8.13 `fdtd-proto2.h` Header file that now includes the prototypes for the TFSF and ABC functions. The differences between this file and 8.6 are shown in bold.

```
1  #ifndef _FDTD_PROTO2_H
2  #define _FDTD_PROTO2_H
3
4  #include "fdtd-grid1.h"
5
6  /* Function prototypes */
7  void abcInit(Grid *g);
8  void abc(Grid *g);
```

```
9
10 void gridInit1d(Grid *g);
11 void gridInit(Grid *g);
12
13 void snapshotInit2d(Grid *g);
14 void snapshot2d(Grid *g);
15
16 void tfsfInit(Grid *g);
17 void tfsfUpdate(Grid *g);
18
19 void updateE2d(Grid *g);
20 void updateH2d(Grid *g);
21
22 #endif
```

The code to implement the TFSF boundary is shown in Program 8.14. There are five global variables in this program. The four declared on lines 7 and 8 give the indices of the first and last points in the TF region. The global variable `g1`, declared on line 10, is a `Grid` pointer that will be used for the auxiliary 1D grid.

The function `tfsfInit()` starts by allocating space for `g1`. Once that space has been allocated we could set the Courant number, the maximum number of time steps, and the size of the grid. However, it is important that these values match, at least in some ways, the value of the 2D grid that has already been declared. Thus, in line 15, the contents of the 2D grid structure are copy to the 1D grid structure. To accomplish this copying the C function `memcpy()` is used. This function takes three arguments: the destination memory address, the source memory address, and the amount of memory to be copied. After this copying has been completed, there are some things about the `g1` which are incorrect. For example, its `type` corresponds to a 2D TM^z grid. Also, the pointers for its arrays are the same as those for the 2D grid. We do not want the 1D grid writing to the same arrays as the 2D grid! Therefore these values within the `Grid` pointer `g1` need to be fix and this is accomplished with the grid-initialization function `gridInit1D()` called in 16. We will consider the details of that function soon. Just prior to returning, `tfsfInit()` initializes the source function by calling `ezIncInit()`.

As we saw in the `main()` function in Program 8.12, `tfsfUpdate()` is called once per time-step, after the magnetic fields have been updated and before the electric field is updated. Note that the fields throughout the grid are not consistent until after the electric field has been updated in the 2D grid (i.e., after step three in the algorithm described on page 206). This is because just prior to calling `tfsfUpdate()` the magnetic fields have not been corrected to account for the TFSF boundary. Just after `tfsfUpdate()` has returned the electric field has been corrected in anticipation of the next update.

Program 8.14 `tfsftmz.c` Source code to implement a TFSF boundary for a TM^z grid. The incident field is assumed to propagate along the x direction and is calculated using an auxiliary 1D simulation.

```
#include <string.h>  // for memcpy
#include "fdtd-macro-tmz.h"
#include "fdtd-proto2.h"
#include "fdtd-alloc1.h"
#include "ezinc.h"

static int firstX = 0, firstY, // indices for first point in TF region
           lastX, lastY;       // indices for last point in TF region

static Grid *g1;  // 1D auxilliary grid

void tfsfInit(Grid *g) {

  ALLOC_1D(g1, 1, Grid);       // allocate memory for 1D Grid
  memcpy(g1, g, sizeof(Grid)); // copy information from 2D array
  gridInit1d(g1);              // initialize 1d grid

  printf("Grid is %d by %d cell.\n", SizeX, SizeY);
  printf("Enter indices for first point in TF region: ");
  scanf(" %d %d", &firstX, &firstY);
  printf("Enter indices for last point in TF region: ");
  scanf(" %d %d", &lastX, &lastY);

  ezIncInit(g); // initialize source function

  return;
}

void tfsfUpdate(Grid *g) {
  int mm, nn;

  // check if tfsfInit() has been called
  if (firstX <= 0) {
    fprintf(stderr,
      "tfsfUpdate: tfsfInit must be called before tfsfUpdate.\n"
      "            Boundary location must be set to positive value.\n");
    exit(-1);
  }

  // correct Hy along left edge
  mm = firstX - 1;
  for (nn = firstY; nn <= lastY; nn++)
    Hy(mm, nn) -= Chye(mm, nn) * Ez1G(g1, mm + 1);

  // correct Hy along right edge
  mm = lastX;
  for (nn = firstY; nn <= lastY; nn++)
```

```
48      Hy(mm, nn) += Chye(mm, nn) * Ez1G(g1, mm);
49
50    // correct Hx along the bottom
51    nn = firstY - 1;
52    for (mm = firstX; mm <= lastX; mm++)
53      Hx(mm, nn) += Chxe(mm, nn) * Ez1G(g1, mm);
54
55    // correct Hx along the top
56    nn = lastY;
57    for (mm = firstX; mm <= lastX; mm++)
58      Hx(mm, nn) -= Chxe(mm, nn) * Ez1G(g1, mm);
59
60    updateH2d(g1);     // update 1D magnetic field
61    updateE2d(g1);     // update 1D electric field
62    Ez1G(g1, 0) = ezInc(TimeG(g1), 0.0); // set source node
63    TimeG(g1)++;       // increment time in 1D grid
64
65    /* correct Ez adjacent to TFSF boundary */
66    // correct Ez field along left edge
67    mm = firstX;
68    for (nn = firstY; nn <= lastY; nn++)
69      Ez(mm, nn) -= Cezh(mm, nn) * Hy1G(g1, mm - 1);
70
71    // correct Ez field along right edge
72    mm = lastX;
73    for (nn = firstY; nn <= lastY; nn++)
74      Ez(mm, nn) += Cezh(mm, nn) * Hy1G(g1, mm);
75
76    // no need to correct Ez along top and bottom since
77    // incident Hx is zero
78
79    return;
80  }
```

The function `tfsfUpdate()`, which is called once per time-step, starts by ensuring that the initialization function has been called. It then corrects H_y along the left and right edges and H_x along the top and bottom edges. Then, in line 60, the magnetic field in the 1D grid is updated, then the 1D electric field. Then the source is realized by hard-wiring the first electric-field node in the 1D grid to the source function (in this case a Ricker wavelet). This is followed by incrementing the time-step in the 1D grid. Now that the 1D grid has been updated, starting in line 67, the electric fields adjacent to the TFSF boundary are corrected. Throughout `tfsfUpdate()` any macro that pertains to `g1` must explicitly specify the `Grid` as an argument.

The function used to initialize the 1D grid is shown in Program 8.15. After inclusion of the appropriate header files, `NLOSS` is defined to be 20. The 1D grid is terminated with a lossy layer rather than an ABC. `NLOSS` represents the number of nodes in this lossy region.

Program 8.15 `grid1dez.c` Initialization function for the 1D auxilliary grid used by the TFSF function to calculate the incident field.

```
#include <math.h>
#include "fdtd-macro-tmz.h"
#include "fdtd-alloc1.h"

#define NLOSS     20   // number of lossy cells at end of 1D grid
#define MAX_LOSS  0.35 // maximum loss factor in lossy layer

void gridInit1d(Grid *g) {
  double imp0 = 377.0, depthInLayer, lossFactor;
  int mm;

  SizeX += NLOSS;    // size of domain
  Type = oneDGrid;   // set grid type

  ALLOC_1D(g->hy,   SizeX - 1, double);
  ALLOC_1D(g->chyh, SizeX - 1, double);
  ALLOC_1D(g->chye, SizeX - 1, double);
  ALLOC_1D(g->ez,   SizeX, double);
  ALLOC_1D(g->ceze, SizeX, double);
  ALLOC_1D(g->cezh, SizeX, double);

  /* set the electric- and magnetic-field update coefficients */
  for (mm = 0; mm < SizeX - 1; mm++) {
    if (mm < SizeX - 1 - NLOSS) {
      Ceze1(mm) = 1.0;
      Cezh1(mm) = Cdtds * imp0;
      Chyh1(mm) = 1.0;
      Chye1(mm) = Cdtds / imp0;
    } else {
      depthInLayer = mm - (SizeX - 1 - NLOSS) + 0.5;
      lossFactor = MAX_LOSS * pow(depthInLayer / NLOSS, 2);
      Ceze1(mm) = (1.0 - lossFactor) / (1.0 + lossFactor);
      Cezh1(mm) = Cdtds * imp0 / (1.0 + lossFactor);
      depthInLayer += 0.5;
      lossFactor = MAX_LOSS * pow(depthInLayer / NLOSS, 2);
      Chyh1(mm) = (1.0 - lossFactor) / (1.0 + lossFactor);
      Chye1(mm) = Cdtds / imp0 / (1.0 + lossFactor);
    }
  }

  return;
}
```

Recall that in `tfsfInit()` the values from the 2D grid were copied to the 1D grid (ref. line 15 of Program 8.14). Thus at the start of this function the value of `SizeX` is set to that of the 2D grid. (The value of `SizeY` is also set to that of the 2D grid, but this value is ignored in the context of a 1D grid.) In line 12 the size is increased by the number of nodes in the lossy layer. This is the final size of the 1D grid: 20 cells greater than the x dimension of the 2D grid.

The grid type is specified as being a `oneDGrid` in line 13. (There is no need to set the Courant number since that was copied from the 2D grid.) This is followed by memory allocation for the various arrays in lines 15 to 20.

The update-equation coefficients are set by the for-loop that begins on line 23. (The final electric-field node does not have its coefficient set as it will not be updated.) The region of the 1D grid corresponding to the width of the 2D grid is set to free space. Recalling the discussion of Sec. 3.12, the remainder of the grid is set to a lossy layer where the electric and magnetic loss are matched so that the characteristic impedance remains that of free space. However, unlike in Sec. 3.12, here the amount of loss is small at the start of the layer and grows towards the end of the grid: The loss increases quadratically as one approaches the end of the grid. The maximum "loss factor" (which corresponds to $\sigma\Delta_t/2\epsilon$ in the electric-field update equations or $\sigma_m\Delta_t/2\mu$ in the magnetic-field update equations) is set by the `#define` statement on line 6 to 0.35. By gradually ramping up the loss, the reflections associated with having an abrupt change in material constants can be greatly reduced. Further note that although the loss factor associated with the electric and magnetic fields are matches, because the electric and magnetic fields are spatially offset, the loss factor that pertains at electric and magnetic field nodes differ even when they have the same spatial index. The loss factor is based on the variable `depthInLayer` which represents how deep a particular node is within the lossy layer. The greater the depth, the greater the loss.

Finally, the file `abctmz.c` is shown in Program 8.16. There are four arrays used to store the old values of field needed by the ABC—one array for each side of the grid. For each node along the edge of the grid, six values must be stored. Thus the arrays that store values along the left and right sides have a total of $6\times$ `SizeY` elements while the arrays that store values along the top and bottom have $6\times$ `SizeX` elements. Starting on line 17 four macros are defined that simplify accessing the elements of these arrays. The macros take three arguments. One arguments specifies displacement along the edge of the grid. Another specifies the displacement into the interior. The third argument specifies the number of steps back in time.

Program 8.16 `abctmz.c` Function to apply a second-order ABC to a TM^z grid.

```
1  /* Second-order ABC for TMz grid. */
2  #include <math.h>
3  #include "fdtd-alloc1.h"
4  #include "fdtd-macro-tmz.h"
5
6  /* Define macros for arrays that store the previous values of the
7   * fields.  For each one of these arrays the three arguments are as
8   * follows:
9   *
10  *   first argument:  spatial displacement from the boundary
```

```
 *   second argument: displacement back in time
 *   third argument:  distance from either the bottom (if EzLeft or
 *                    EzRight) or left (if EzTop or EzBottom) side
 *                    of grid
 *
 */
#define EzLeft(M, Q, N)   ezLeft[(N) * 6 + (Q) * 3 + (M)]
#define EzRight(M, Q, N) ezRight[(N) * 6 + (Q) * 3 + (M)]
#define EzTop(N, Q, M)       ezTop[(M) * 6 + (Q) * 3 + (N)]
#define EzBottom(N, Q, M) ezBottom[(M) * 6 + (Q) * 3 + (N)]

static int initDone = 0;
static double coef0, coef1, coef2;
static double *ezLeft, *ezRight, *ezTop, *ezBottom;

void abcInit(Grid *g) {
  double temp1, temp2;

  initDone = 1;

  /* allocate memory for ABC arrays */
  ALLOC_1D(ezLeft, SizeY * 6, double);
  ALLOC_1D(ezRight, SizeY * 6, double);
  ALLOC_1D(ezTop, SizeX * 6, double);
  ALLOC_1D(ezBottom, SizeX * 6, double);

  /* calculate ABC coefficients */
  temp1 = sqrt(Cezh(0, 0) * Chye(0, 0));
  temp2 = 1.0 / temp1 + 2.0 + temp1;
  coef0 = -(1.0 / temp1 - 2.0 + temp1) / temp2;
  coef1 = -2.0 * (temp1 - 1.0 / temp1) / temp2;
  coef2 = 4.0 * (temp1 + 1.0 / temp1) / temp2;

  return;
}

void abc(Grid *g)
{
  int mm, nn;

  /* ABC at left side of grid */
  for (nn = 0; nn < SizeY; nn++) {
    Ez(0, nn) = coef0 * (Ez(2, nn) + EzLeft(0, 1, nn))
     + coef1 * (EzLeft(0, 0, nn) + EzLeft(2, 0, nn)
                - Ez(1, nn) - EzLeft(1, 1, nn))
     + coef2 * EzLeft(1, 0, nn) - EzLeft(2, 1, nn);

```

```
58      /* memorize old fields */
59      for (mm = 0; mm < 3; mm++) {
60        EzLeft(mm, 1, nn) = EzLeft(mm, 0, nn);
61        EzLeft(mm, 0, nn) = Ez(mm, nn);
62      }
63    }
64
65    /* ABC at right side of grid */
66    for (nn = 0; nn < SizeY; nn++) {
67      Ez(SizeX - 1, nn) = coef0 * (Ez(SizeX - 3, nn) + EzRight(0, 1, nn))
68       + coef1 * (EzRight(0, 0, nn) + EzRight(2, 0, nn)
69                  - Ez(SizeX - 2, nn) - EzRight(1, 1, nn))
70       + coef2 * EzRight(1, 0, nn) - EzRight(2, 1, nn);
71
72      /* memorize old fields */
73      for (mm = 0; mm < 3; mm++) {
74        EzRight(mm, 1, nn) = EzRight(mm, 0, nn);
75        EzRight(mm, 0, nn) = Ez(SizeX - 1 - mm, nn);
76      }
77    }
78
79    /* ABC at bottom of grid */
80    for (mm = 0; mm < SizeX; mm++) {
81      Ez(mm, 0) = coef0 * (Ez(mm, 2) + EzBottom(0, 1, mm))
82       + coef1 * (EzBottom(0, 0, mm) + EzBottom(2, 0, mm)
83                  - Ez(mm, 1) - EzBottom(1, 1, mm))
84       + coef2 * EzBottom(1, 0, mm) - EzBottom(2, 1, mm);
85
86      /* memorize old fields */
87      for (nn = 0; nn < 3; nn++) {
88        EzBottom(nn, 1, mm) = EzBottom(nn, 0, mm);
89        EzBottom(nn, 0, mm) = Ez(mm, nn);
90      }
91    }
92
93    /* ABC at top of grid */
94    for (mm = 0; mm < SizeX; mm++) {
95      Ez(mm, SizeY - 1) = coef0 * (Ez(mm, SizeY - 3) + EzTop(0, 1, mm))
96       + coef1 * (EzTop(0, 0, mm) + EzTop(2, 0, mm)
97                  - Ez(mm, SizeY - 2) - EzTop(1, 1, mm))
98       + coef2 * EzTop(1, 0, mm) - EzTop(2, 1, mm);
99
100     /* memorize old fields */
101     for (nn = 0; nn < 3; nn++) {
102       EzTop(nn, 1, mm) = EzTop(nn, 0, mm);
103       EzTop(nn, 0, mm) = Ez(mm, SizeY - 1 - nn);
104     }
```

```
105     }
106
107     return;
108 }
```

The initialization function starting on line 26 allocates space for the arrays and calculates the coefficients used by the ABC. It is assumed the grid is uniform along the edge and the coefficients are calculated based on the parameters that pertain at the first node in the grid (as indicated by the statements starting on line 38).

The `abc()` function, which starts on line 47 and is called once per time step, systematically applies the ABC to each node along the edge of the grid. After the ABC is applied to an edge, the "old" stored values are updated.

8.7 TE^z Polarization

In TE^z polarization the non-zero fields are E_x, E_y, and H_z, i.e., the electric field is transverse to the z direction. The fields may vary in the x and y directions but are invariant in z. These fields, and the corresponding governing equations, are completely decoupled from those of TM^z polarization. The governing equations are

$$\sigma E_x + \epsilon \frac{\partial E_x}{\partial t} = \frac{\partial H_z}{\partial y}, \tag{8.19}$$

$$\sigma E_y + \epsilon \frac{\partial E_y}{\partial t} = -\frac{\partial H_z}{\partial x}, \tag{8.20}$$

$$-\sigma_m H_z - \mu \frac{\partial H_z}{\partial t} = \frac{\partial E_y}{\partial x} - \frac{\partial E_x}{\partial y}. \tag{8.21}$$

As usual, space-time is discretized so that (8.19)–(8.21) can be expressed in terms of finite-differences. From these difference equations the future fields can be expressed in terms of past fields. The following notation will be used:

$$E_x(x,y,t) = E_x(m\Delta_x, n\Delta_y, q\Delta_t) = E_x^q[m,n], \tag{8.22}$$

$$E_y(x,y,t) = E_y(m\Delta_x, n\Delta_y, q\Delta_t) = E_y^q[m,n] \tag{8.23}$$

$$H_z(x,y,t) = H_z(m\Delta_x, n\Delta_y, q\Delta_t) = H_z^q[m,n]. \tag{8.24}$$

As before the indices m, n, and q specify the step in the x, y, and t "directions."

A suitable arrangement of nodes is shown in Fig. 8.9. The triangularly shaped dashed lines in the lower left of the grid enclose nodes which would have the same indices in a computer program.

Note that the grid is terminated such that there are tangential electric field nodes adjacent to the boundary. (When it comes to the application of ABC's, these are the nodes to which the ABC would be applied.) When we say a TE^z grid has dimensions $M \times N$, the arrays are dimensioned as follows: E_x is $(M-1) \times N$, E_y is $M \times (N-1)$, and H_z is $(M-1) \times (N-1)$. Therefore, although the grid is described as $M \times N$, no array actually has these dimensions! Each magnetic-field node has four adjacent electric-field nodes that "swirl" about it. One can think of these four nodes as

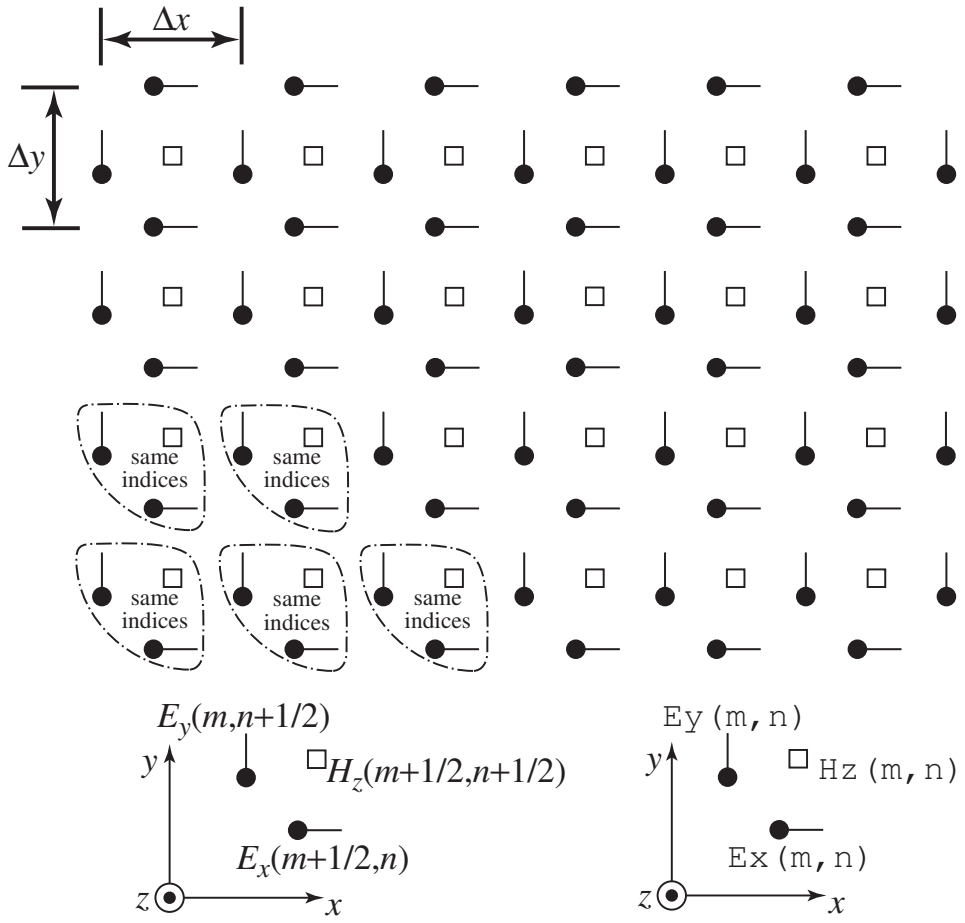


Figure 8.9: Spatial arrangement of electric- and magnetic-field nodes for TE^z polarization. The magnetic-field nodes are shown as squares and the electric-field nodes are circles with a line that indicates the orientation of the field component. The somewhat triangularly shaped dashed lines indicate groupings of nodes which have the same array indices. This grouping is repeated throughout the grid. However, at the top of the grid the "group" only contains an E_x node and on the right side of the grid the group only contains an E_y node. The diagram at the bottom left of the figure indicates nodes with their offsets given explicitly in the spatial arguments whereas the diagram at the bottom right indicates how the same nodes would be specified in a computer program where the offsets are understood implicitly.

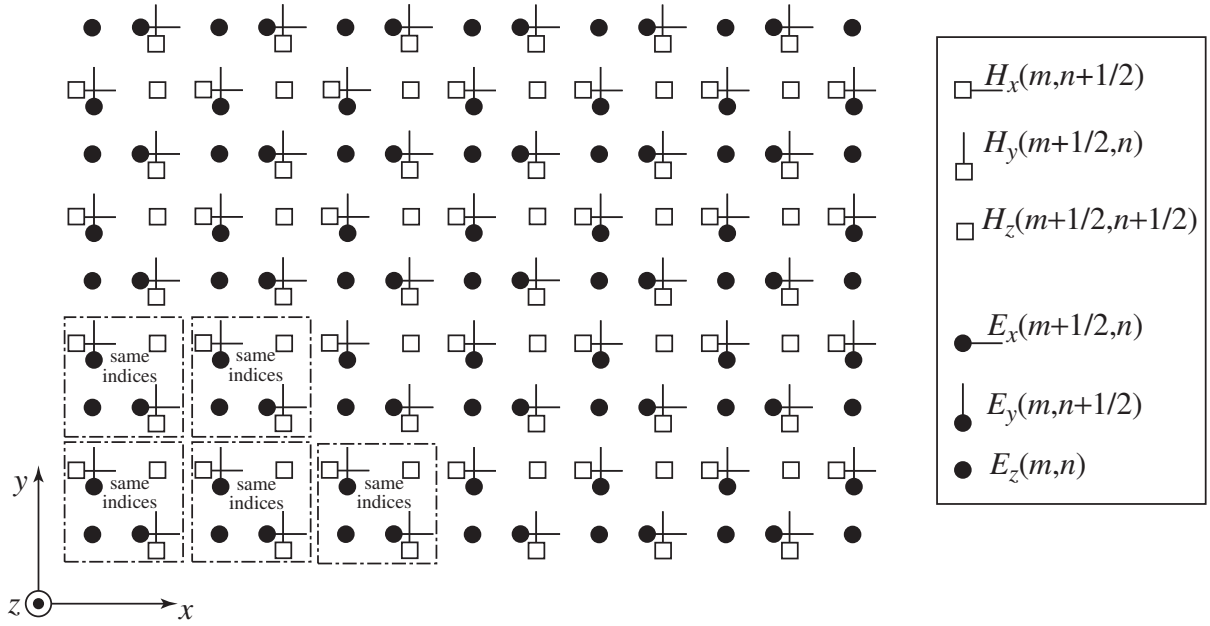


Figure 8.10: Superposition of a TM^z and TE^z grid. The symbols used for the nodes is as before. The dashed boxes enclose nodes which have the same indices. Although this is nominally identified as an $M \times N$ grid, only the E_z array has $M \times N$ nodes.

defining a square with the magnetic field at the center of the square (if the grid is not uniform, the square becomes a rectangle). An $M \times N$ grid would consist of $(M-1) \times (N-1)$ complete squares.

The way in which the TE^z arrays are dimensioned may seem odd but it is done with an eye toward having a consistent grid in three dimensions. As an indication of where we will ultimately end up, we can overlay a TM^z and TE^z grid as shown in Fig. 8.10. As will be shown in the discussion of 3D grids, a 3D grid is essentially layers of TM^z and TE^z grids which are offset from each other in the z direction. The update equations of these offset grids will have to be modified to account for variations in the z directions. This modification will provide the coupling between the TM^z and TE^z grids which is lacking in 2D.

Given the governing equations (8.19)–(8.21) and the arrangement of nodes shown in Fig. 8.9, the H_z update equation is

$$\begin{aligned} H_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n+\frac{1}{2}\right] &= \frac{1-\frac{\sigma_m\Delta_t}{2\mu}}{1+\frac{\sigma_m\Delta_t}{2\mu}} H_z^{q-\frac{1}{2}}\left[m+\frac{1}{2},n+\frac{1}{2}\right] \\ &\quad - \frac{1}{1+\frac{\sigma_m\Delta_t}{2\epsilon}}\left(\frac{\Delta_t}{\epsilon\Delta_x}\left\{E_y^q\left[m+1,n+\frac{1}{2}\right]-E_y^q\left[m,n+\frac{1}{2}\right]\right\}\right. \\ &\quad \left. - \frac{\Delta_t}{\mu\Delta_y}\left\{E_x^q\left[m+\frac{1}{2},n+1\right]-E_x^q\left[m+\frac{1}{2},n\right]\right\}\right) \end{aligned} \quad (8.25)$$

The electric-field update equations are

$$E_x^{q+1}\left[m+\frac{1}{2},n\right] = \frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}}E_x^q\left[m+\frac{1}{2},n\right] + \frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\frac{\Delta_t}{\epsilon\Delta_y}\left(H_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n+\frac{1}{2}\right]-H_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n-\frac{1}{2}\right]\right), \quad (8.26)$$

$$E_y^{q+1}\left[m,n+\frac{1}{2}\right] = \frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}}E_y^q\left[m,n+\frac{1}{2}\right] - \frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\frac{\Delta_t}{\epsilon\Delta_x}\left(H_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n+\frac{1}{2}\right]-H_z^{q+\frac{1}{2}}\left[m-\frac{1}{2},n+\frac{1}{2}\right]\right). \quad (8.27)$$

Similar to the TM^z case, we assume a uniform grid and define the following quantities

$$C_{hxh}(m+1/2,n+1/2) = \left.\frac{1-\frac{\sigma_m\Delta_t}{2\mu}}{1+\frac{\sigma_m\Delta_t}{2\mu}}\right|_{(m+1/2)\Delta_x,(n+1/2)\Delta_y}, \quad (8.28)$$

$$C_{hxe}(m+1/2,n+1/2) = \left.\frac{1}{1+\frac{\sigma_m\Delta_t}{2\mu}}\frac{\Delta_t}{\mu\delta}\right|_{(m+1/2)\Delta_x,(n+1/2)\Delta_y}, \quad (8.29)$$

$$C_{exe}(m+1/2,n) = \left.\frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}}\right|_{(m+1/2)\Delta_x,n\Delta_y}, \quad (8.30)$$

$$C_{exh}(m+1/2,n) = \left.\frac{1}{1+\frac{\sigma_m\Delta_t}{2\epsilon}}\frac{\Delta_t}{\epsilon\delta}\right|_{(m+1/2)\Delta_x,n\Delta_y}, \quad (8.31)$$

$$C_{eye}(m,n+1/2) = \left.\frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}}\right|_{m\Delta_x,(n+1/2)\Delta_y}, \quad (8.32)$$

$$C_{eyh}(m,n+1/2) = \left.\frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\frac{\Delta_t}{\epsilon\delta}\right|_{m\Delta_x,(n+1/2)\Delta_y}. \quad (8.33)$$

By discarding the explicit offsets of one-half (but leaving them as implicitly understood) the update equations can be written in a form suitable for implementation in a computer. Because of the arrangement of the nodes, this "discarding" implies that sometimes the one-half truly is discarded and sometimes it should be replaced with unity. The distinction is whether or not the one-half indicates the nodes on the right side of the update equation are within the same grouping of cells as the node on the left side of the equation. If they are, the one-half is truly discarded. If they are not, the node on the right side of the update equation must have its index reflect which group of cells it is within relative to the node on the left side of the equation. The resulting equations are

```
Hz(m, n) = Chzh(m, n) * Hz(m, n) +
   Chze(m, n) * ((Ex(m, n + 1) - Ex(m, n)) -
                 (Ey(m + 1, n) - Ey(m, n)));
Ex(m, n) = Cexe(m, n) * Ex(m, n) +
   Cexh(m, n) * (Hz(m, n) - Hz(m, n - 1));
```

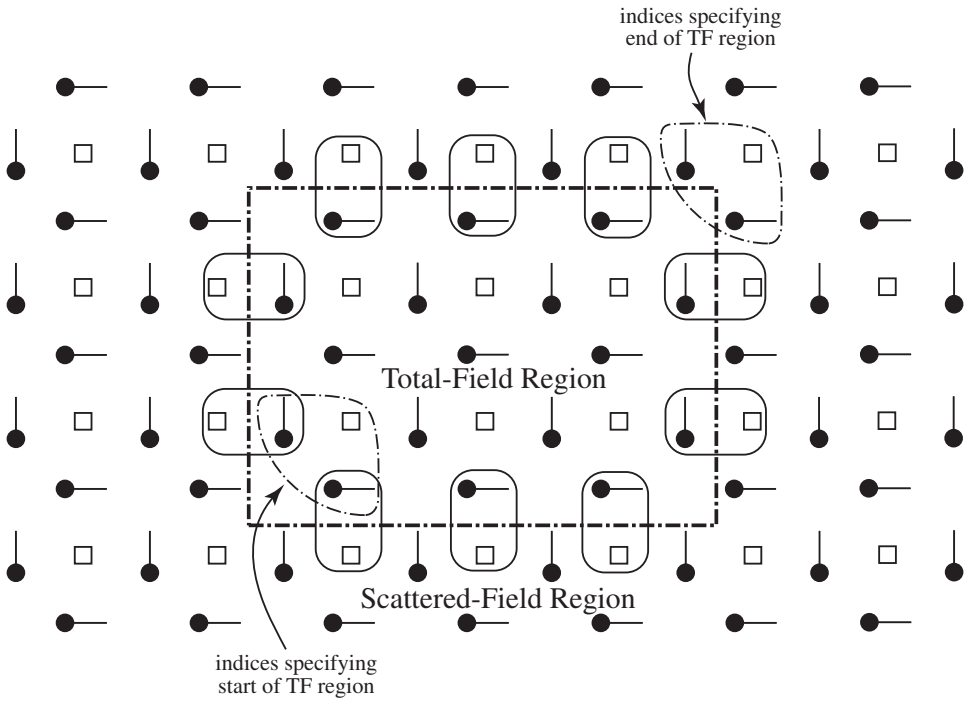


Figure 8.11: TFSF boundary in a TE^z grid. The rounded boxes indicate the nodes that have a neighboring node on the other side of the boundary and hence have to have their update equations corrected.

```
Ey(m, n) = Ceye(m, n) * Ey(m, n) -
   Ceyh(m, n) * (Hz(m, n) - Hz(m - 1, n));
```

A TFSF boundary can be incorporated in a TE^z grid. Conceptually the implementation is the same as has been shown in 1D and in the TM^z grid. Nodes that are tangential to the boundary will have a neighboring node on the other side of the boundary. The incident field will have to be either added to or subtracted from that neighboring node to obtain consistent equations. A portion of a TE^z grid showing the TFSF boundary is shown in Fig. 8.11. We will specify the size of the TF region as indicated in the figure. Indices that specify the start of the TF region correspond to the first E_x and E_y nodes which are in the TF region. Referring to Figs. 8.4 and 8.10, these indices would also correspond to the first E_z node in the TF region. The indices which specify the end of the TF region correspond to E_x and E_y nodes which are actually in the SF region. These two nodes, as shown in Fig. 8.10, are not tangential to the TFSF boundary and hence do to have to be corrected. However, note that the E_z node in the overlain grid that has these indices does lie in the TF region (and does, when dealing with a 3D or TM^z grid, have to be corrected to account for the presence of the boundary).

8.8 PEC's in TE^z and TM^z Simulations

When modeling a PEC in a TM^z grid, if an E_z node falls within the PEC, it is set to zero. Figure 8.12 shows a portion of a TM^z grid that depicts how E_z would set to zero. The curved boundary

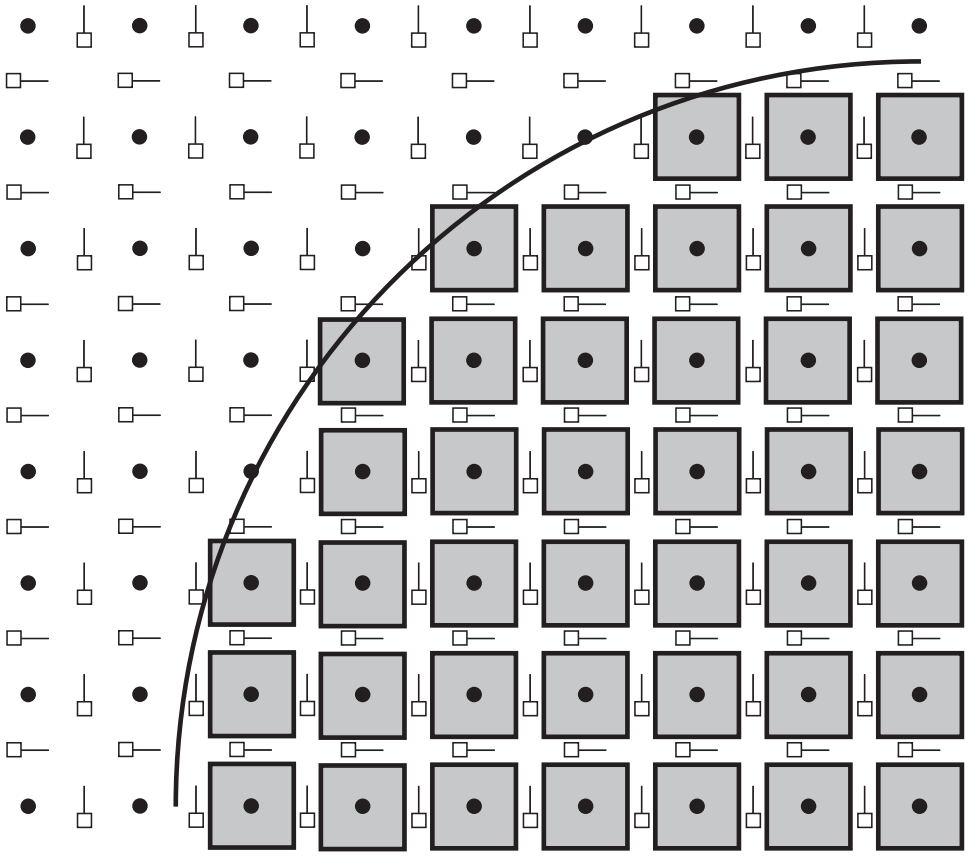


Figure 8.12: TM^z grid with a PEC object. The PEC is assumed to exist below and to the right of the curved boundary. The PEC is realized by setting to zero the E_z nodes that fall within the PEC. The nodes that would be set to zero are surrounded by gray boxes.

is the surface of the PEC and it is assumed that the PEC extends down and to the right of this boundary. The E_z nodes which would be set to zero are indicated with gray boxes. Although the goal is to model a continuously varying boundary, the discrete nature of the FDTD grid gives rise to a "staircased" approximation of the surface.

When we say a node is "set to zero" this could mean various things. For example, it may mean that the field is initially zero and then never updated. It could mean that it is updated, but the update coefficients are set to zero. Or, it could even mean that the field is updated with non-zero coefficients, but then additional code is used to set the field to zero each time-step. The means by which a field is set to zero is not particularly important to us right now.

A thin PEC plate can be modeled in a TM^z grid by setting to zero nodes along a vertical or horizontal line. If the physical plate being modeled is not aligned with the grid, one would have to zero nodes in a manner that approximates the true slope of the plate. Again, this would yield a staircased approximate to the true surface. (One may have to be careful to ensure that there are no "gaps" in the model of a thin PEC that is not aligned with the grid. Fields should only be able to get from one side of the PEC to the other by propagating around the ends of the PEC.)

In a TE^z grid, the realization of a PEC is slightly more complicated. For a PEC object which

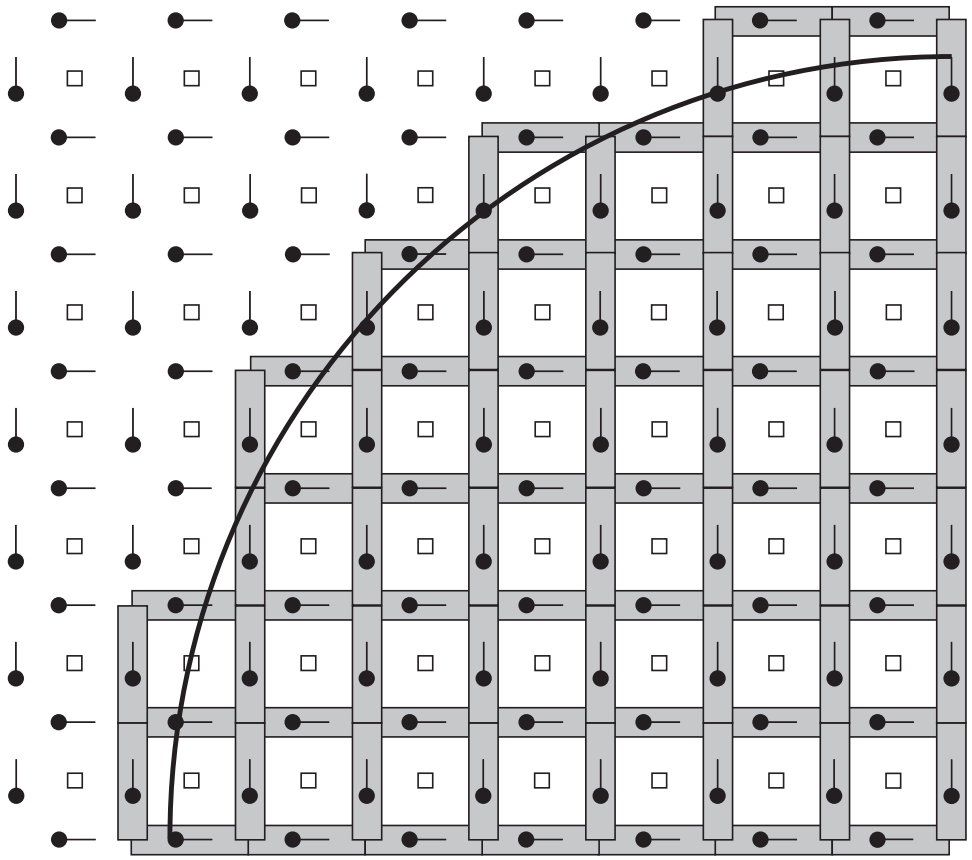


Figure 8.13: TE^z grid with a PEC object. The PEC is assumed to exist below and to the right of the curved boundary. The PEC is realized be setting to zero any electric field which has a neighboring H_z node within the PEC. The nodes that would be set to zero are surrounded by gray rectangles.

has a specified cross section, one should not merely set to zero the electric-field nodes that fall within the boundary of the PEC (as was done in the TM^z case). Instead, one should consider the PEC as consisting of a collection of patches of metal. If an H_z node falls within the PEC, then four surrounding electric-field nodes should be set to zero. Thus, if an H_z node is in the PEC we fill the square surrounding that node with PEC and this causes the four surrounding electric field nodes to be zero. The TE^z representation of a PEC object is depicted in Fig. 8.13. The object is the same as shown in Fig. 8.12. In both figures the curved boundary is a portion of a circle that is center on what would correspond to the location of an E_z node (regardless of whether or not an E_z node is actually present). The nodes that are set to zero are enclosed in gray rectangles.

A horizontal PEC plate would be implemented by zeroing a horizontal line of E_x nodes while a vertical plate would be realized by zeroing a vertical line of E_y nodes. A tilted plate would be realized as a combination of zeroed E_x and E_y nodes.

For both TE^z and TM^z grids, all the magnetic fields are updated in the usual way. Magnetic fields are oblivious to the presence of PEC's.

8.9 TE^z Example

In this section we present the computer code to model a circular PEC scatterer in a TE^z grid. The scatterer is illuminated by a pulsed plane wave that is introduced via a TFSF boundary. We will use a grid that is nominally 92 by 82 (keeping in mind that for TE^z polarization none of the field arrays will actually have these dimensions). The code is organized in essentially the same way as was the TM^z code presented in Sec. 8.6.

The PEC scatterer is assumed to have a radius of 12 cells and be centered on an H_z node. The indices of the center are $(45, 40)$. The PEC is realized by checking if an H_z node is within the circle (specifically, if the distance from the center to the node is less than the radius). As we will see, if an H_z is within the circle, the four surrounding electric-field nodes are set to zero by setting the corresponding update coefficients to zero.

Program 8.17 contains the `main()` function. Other than the difference of one header file, this program is identical to the TM^z code presented in Program 8.12. However, despite similar names, the functions that are called here differ from those used by Program 8.12—different files are linked together for the different simulations.

Program 8.17 `tezdemo.c` The `main()` function for a simulation involving a TE^z grid.

```
1  /* TEz simulation with a TFSF boundary and a second-order ABC. */
2
3  #include "fdtd-alloc1.h"
4  #include "fdtd-macro-tez.h"
5  #include "fdtd-proto2.h"
6
7  int main()
8  {
9    Grid *g;
10
11   ALLOC_1D(g, 1, Grid); // allocate memory for grid
12   gridInit(g);          // initialize 2D grid
13
14   abcInit(g);           // initialize ABC
15   tfsfInit(g);          // initialize TFSF boundary
16   snapshotInit2d(g);    // initialize snapshots
17
18   /* do time stepping */
19   for (Time = 0; Time < MaxTime; Time++) {
20     updateH2d(g);       // update magnetic fields
21     tfsfUpdate(g);      // apply TFSF boundary
22     updateE2d(g);       // update electric fields
23     abc(g);             // apply ABC
24     snapshot2d(g);      // take a snapshot (if appropriate)
25   } // end of time-stepping
26
27   return 0;
```

```
28 }
```

The code to construct the TE^z grid is shown in Program 8.18, i.e., the code to set the elements of the `Grid` pointer `g`. The simulation is run at the Courant limit of $1/\sqrt{2}$ as shown in line 16. Between lines 31 and 41 the update coefficients for all the electric field nodes are set to that of free space. Then, starting at line 49, each H_z node is checked to see if it is within the PEC scatterer. If it is, the coefficients for the surrounding nodes are set to zero. Starting at line 71 all the magnetic-field coefficients are set to that of free space. There is no need to change these coefficients to account for the PEC—the PEC is realized solely by dictating the behavior of the electric field.

Program 8.18 `gridtezpec.c` Function to initialize a TE^z grid. A circular PEC scatterer is present.

```
1  #include "fdtd-macro-tez.h"
2  #include "fdtd-alloc1.h"
3  #include <math.h>
4
5  void gridInit(Grid *g) {
6    double imp0 = 377.0;
7    int mm, nn;
8
9    /* terms for the PEC scatterer */
10   double rad, r2, xLocation, yLocation, xCenter, yCenter;
11
12   Type = teZGrid;
13   SizeX = 92;       // size of domain
14   SizeY = 82;
15   MaxTime = 300;    // duration of simulation
16   Cdtds = 1.0 / sqrt(2.0); // Courant number
17
18   ALLOC_2D(g->hz,   SizeX - 1, SizeY - 1, double);
19   ALLOC_2D(g->chzh, SizeX - 1, SizeY - 1, double);
20   ALLOC_2D(g->chze, SizeX - 1, SizeY - 1, double);
21
22   ALLOC_2D(g->ex,   SizeX - 1, SizeY, double);
23   ALLOC_2D(g->cexh, SizeX - 1, SizeY, double);
24   ALLOC_2D(g->cexe, SizeX - 1, SizeY, double);
25
26   ALLOC_2D(g->ey,   SizeX, SizeY - 1, double);
27   ALLOC_2D(g->ceye, SizeX, SizeY - 1, double);
28   ALLOC_2D(g->ceyh, SizeX, SizeY - 1, double);
29
30   /* set electric-field update coefficients */
31   for (mm = 0; mm < SizeX - 1; mm++)
32     for (nn = 0; nn < SizeY; nn++) {
```

```
33        Cexe(mm, nn) = 1.0;
34        Cexh(mm, nn) = Cdtds * imp0;
35      }
36
37    for (mm = 0; mm < SizeX; mm++)
38      for (nn = 0; nn < SizeY - 1; nn++) {
39        Ceye(mm, nn) = 1.0;
40        Ceyh(mm, nn) = Cdtds * imp0;
41      }
42
43    /* Set to zero nodes associated with PEC scatterer.
44     * Circular scatterer assumed centered on Hz node
45     * at (xCenter, yCenter).  If an Hz node is less than
46     * the radius away from this node, set to zero the
47     * four electric fields that surround that node.
48     */
49    rad = 12;  // radius of circle
50    xCenter = SizeX / 2;
51    yCenter = SizeY / 2;
52    r2 = rad * rad;  // square of radius
53    for (mm = 1; mm < SizeX - 1; mm++) {
54      xLocation = mm - xCenter;
55      for (nn = 1; nn < SizeY - 1; nn++) {
56        yLocation = nn - yCenter;
57        if (xLocation * xLocation + yLocation * yLocation < r2) {
58          Cexe(mm, nn) = 0.0;
59          Cexh(mm, nn) = 0.0;
60          Cexe(mm, nn + 1) = 0.0;
61          Cexh(mm, nn + 1) = 0.0;
62          Ceye(mm + 1, nn) = 0.0;
63          Ceyh(mm + 1, nn) = 0.0;
64          Ceye(mm, nn) = 0.0;
65          Ceyh(mm, nn) = 0.0;
66        }
67      }
68    }
69
70    /* set magnetic-field update coefficients */
71    for (mm = 0; mm < SizeX - 1; mm++)
72      for (nn = 0; nn < SizeY - 1; nn++) {
73        Chzh(mm, nn) = 1.0;
74        Chze(mm, nn) = Cdtds / imp0;
75      }
76
77    return;
78  }
```

The header file `fdtd-macro-tez.h` that defines the macros used in the TE^z simulations is shown in Program 8.19. The header files that define the function prototypes (`fdtd-proto2.h`), the allocation macros (`fdtd-alloc1.h`), and the `Grid` structure (`fdtd-grid1.h`) are unchanged from before and hence are not repeated here (refer to Programs 8.13, 8.4, and 8.3, respectively).

Program 8.19 `fdtd-macro-tez.h` Macros used for TE^z grids.

```
#ifndef _FDTD_MACRO_TEZ_H
#define _FDTD_MACRO_TEZ_H

#include "fdtd-grid1.h"

/* macros that permit the "Grid" to be specified */
/* one-dimensional grid */
#define Hz1G(G, M)     G->hz[M]
#define Chzh1G(G, M)   G->chzh[M]
#define Chze1G(G, M)   G->chze[M]

#define Ey1G(G, M)     G->ey[M]
#define Ceye1G(G, M)   G->ceye[M]
#define Ceyh1G(G, M)   G->ceyh[M]

/* TEz grid */
#define HzG(G, M, N)     G->hz[(M) * (SizeYG(G) - 1) + (N)]
#define ChzhG(G, M, N) G->chzh[(M) * (SizeYG(G) - 1) + (N)]
#define ChzeG(G, M, N) G->chze[(M) * (SizeYG(G) - 1) + (N)]

#define ExG(G, M, N)     G->ex[(M) * SizeYG(G) + (N)]
#define CexeG(G, M, N) G->cexe[(M) * SizeYG(G) + (N)]
#define CexhG(G, M, N) G->cexh[(M) * SizeYG(G) + (N)]

#define EyG(G, M, N)     G->ey[(M) * (SizeYG(G) - 1) + (N)]
#define CeyeG(G, M, N) G->ceye[(M) * (SizeYG(G) - 1) + (N)]
#define CeyhG(G, M, N) G->ceyh[(M) * (SizeYG(G) - 1) + (N)]

#define SizeXG(G)        G->sizeX
#define SizeYG(G)        G->sizeY
#define SizeZG(G)        G->sizeZ
#define TimeG(G)         G->time
#define MaxTimeG(G)      G->maxTime
#define CdtdsG(G)        G->cdtds
#define TypeG(G)         G->type

/* macros that assume the "Grid" is "g" */
/* one-dimensional grid */
```

```
39 #define Hz1(M)      Hz1G(g, M)
40 #define Chzh1(M)    Chzh1G(g, M)
41 #define Chze1(M)    Chze1G(g, M)
42
43 #define Ey1(M)      Ey1G(g, M)
44 #define Ceye1(M)    Ceye1G(g, M)
45 #define Ceyh1(M)    Ceyh1G(g, M)
46
47 /* TEz grid */
48 #define Hz(M, N)   HzG(g, M, N)
49 #define Chzh(M, N) ChzhG(g, M, N)
50 #define Chze(M, N) ChzeG(g, M, N)
51
52 #define Ex(M, N)   ExG(g, M, N)
53 #define Cexh(M, N) CexhG(g, M, N)
54 #define Cexe(M, N) CexeG(g, M, N)
55
56 #define Ey(M, N)   EyG(g, M, N)
57 #define Ceye(M, N) CeyeG(g, M, N)
58 #define Ceyh(M, N) CeyhG(g, M, N)
59
60 #define SizeX        SizeXG(g)
61 #define SizeY        SizeYG(g)
62 #define SizeZ        SizeZG(g)
63 #define Time         TimeG(g)
64 #define MaxTime      MaxTimeG(g)
65 #define Cdtds        CdtdsG(g)
66 #define Type         TypeG(g)
67
68 #endif   /* matches #ifndef _FDTD_MACRO_TEZ_H */
```

The functions to update the fields are shown in Program 8.20. These functions can update fields in either one- or two-dimensional grid. If the grid type is `oneDGrid`, here it is assumed the non-zero fields are E_y and H_z. If that is not the case, it is assume the grid is a TE^z grid with non-zero fields E_x, E_y, and H_z. As has been the case in the past the electric field updates, starting at line 37, update all the nodes except the nodes at the edge of the grid. However, since all the magnetic-field nodes have all their neighbors, as shown starting on line 15, all the magnetic-field nodes in the grid are updated.

Program 8.20 `updatetez.c`: Functions to update fields in a TE^z grid.

```
1 #include "fdtd-macro-tez.h"
2
3 /* update magnetic field */
4 void updateH2d(Grid *g) {
```

```
5    int mm, nn;
6
7    if (Type == oneDGrid) {
8
9      for (mm = 0; mm < SizeX - 1; mm++)
10       Hz1(mm) = Chzh1(mm) * Hz1(mm)
11         - Chze1(mm) * (Ey1(mm + 1) - Ey1(mm));
12
13   } else {
14
15     for (mm = 0; mm < SizeX - 1; mm++)
16       for (nn = 0; nn < SizeY - 1; nn++)
17         Hz(mm, nn) = Chzh(mm, nn) * Hz(mm, nn) +
18           Chze(mm, nn) * ((Ex(mm, nn + 1) - Ex(mm, nn))
19                           - (Ey(mm + 1, nn) - Ey(mm, nn)));
20   }
21
22   return;
23 }
24
25 /* update electric field */
26 void updateE2d(Grid *g) {
27   int mm, nn;
28
29   if (Type == oneDGrid) {
30
31     for (mm = 1; mm < SizeX - 1; mm++)
32       Ey1(mm) = Ceye1(mm) * Ey1(mm)
33         - Ceyh1(mm) * (Hz1(mm) - Hz1(mm - 1));
34
35   } else {
36
37     for (mm = 0; mm < SizeX - 1; mm++)
38       for (nn = 1; nn < SizeY - 1; nn++)
39         Ex(mm, nn) = Cexe(mm, nn) * Ex(mm, nn) +
40           Cexh(mm, nn) * (Hz(mm, nn) - Hz(mm, nn - 1));
41
42     for (mm = 1; mm < SizeX - 1; mm++)
43       for (nn = 0; nn < SizeY - 1; nn++)
44         Ey(mm, nn) = Ceye(mm, nn) * Ey(mm, nn) -
45           Ceyh(mm, nn) * (Hz(mm, nn) - Hz(mm - 1, nn));
46   }
47
48   return;
49 }
```

The second-order absorbing boundary condition is realized with the code in the file `abctez.c`

which is shown in Program 8.21. Because of the way the grid is constructed, the ABC is applied to E_y nodes along the left and right side of the computational domain and to E_x nodes along the top and bottom.

Program 8.21 `abctez.c`: Contents of file that implements the second-order absorbing boundary condition for the TE^z grid.

```
/* Second-order ABC for TEz grid. */
#include <math.h>
#include "fdtd-alloc1.h"
#include "fdtd-macro-tez.h"

/* Define macros for arrays that store the previous values of the
 * fields.  For each one of these arrays the three arguments are as
 * follows:
 *
 *   first argument:  spatial displacement from the boundary
 *   second argument: displacement back in time
 *   third argument:  distance from either the bottom (if EyLeft or
 *                    EyRight) or left (if ExTop or ExBottom) side
 *                    of grid
 *
 */
#define EyLeft(M, Q, N)    eyLeft[(N) * 6 + (Q) * 3 + (M)]
#define EyRight(M, Q, N)  eyRight[(N) * 6 + (Q) * 3 + (M)]
#define ExTop(N, Q, M)       exTop[(M) * 6 + (Q) * 3 + (N)]
#define ExBottom(N, Q, M) exBottom[(M) * 6 + (Q) * 3 + (N)]

static int initDone = 0;
static double coef0, coef1, coef2;
static double *eyLeft, *eyRight, *exTop, *exBottom;

void abcInit(Grid *g) {
  double temp1, temp2;

  initDone = 1;

  /* allocate memory for ABC arrays */
  ALLOC_1D(eyLeft, (SizeY - 1) * 6, double);
  ALLOC_1D(eyRight, (SizeY - 1) * 6, double);
  ALLOC_1D(exTop, (SizeX - 1) * 6, double);
  ALLOC_1D(exBottom, (SizeX - 1) * 6, double);

  /* calculate ABC coefficients */
  temp1  = sqrt(Cexh(0, 0) * Chze(0, 0));
  temp2 = 1.0 / temp1 + 2.0 + temp1;
```

```
40    coef0 = -(1.0 / temp1 - 2.0 + temp1) / temp2;
41    coef1 = -2.0 * (temp1 - 1.0 / temp1) / temp2;
42    coef2 = 4.0 * (temp1 + 1.0 / temp1) / temp2;
43
44    return;
45  }
46
47  void abc(Grid *g)
48  {
49    int mm, nn;
50
51    /* ABC at left side of grid */
52    for (nn = 0; nn < SizeY - 1; nn++) {
53      Ey(0, nn) = coef0 * (Ey(2, nn) + EyLeft(0, 1, nn))
54        + coef1 * (EyLeft(0, 0, nn) + EyLeft(2, 0, nn)
55                   - Ey(1, nn) - EyLeft(1, 1, nn))
56        + coef2 * EyLeft(1, 0, nn) - EyLeft(2, 1, nn);
57
58      /* memorize old fields */
59      for (mm = 0; mm < 3; mm++) {
60        EyLeft(mm, 1, nn) = EyLeft(mm, 0, nn);
61        EyLeft(mm, 0, nn) = Ey(mm, nn);
62      }
63    }
64
65    /* ABC at right side of grid */
66    for (nn = 0; nn < SizeY - 1; nn++) {
67      Ey(SizeX - 1, nn) = coef0 * (Ey(SizeX - 3, nn) + EyRight(0, 1, nn))
68        + coef1 * (EyRight(0, 0, nn) + EyRight(2, 0, nn)
69                   - Ey(SizeX - 2, nn) - EyRight(1, 1, nn))
70        + coef2 * EyRight(1, 0, nn) - EyRight(2, 1, nn);
71
72      /* memorize old fields */
73      for (mm = 0; mm < 3; mm++) {
74        EyRight(mm, 1, nn) = EyRight(mm, 0, nn);
75        EyRight(mm, 0, nn) = Ey(SizeX - 1 - mm, nn);
76      }
77    }
78
79    /* ABC at bottom of grid */
80    for (mm = 0; mm < SizeX - 1; mm++) {
81      Ex(mm, 0) = coef0 * (Ex(mm, 2) + ExBottom(0, 1, mm))
82        + coef1 * (ExBottom(0, 0, mm) + ExBottom(2, 0, mm)
83                   - Ex(mm, 1) - ExBottom(1, 1, mm))
84        + coef2 * ExBottom(1, 0, mm) - ExBottom(2, 1, mm);
85
86      /* memorize old fields */
```

```
87      for (nn = 0; nn < 3; nn++) {
88        ExBottom(nn, 1, mm) = ExBottom(nn, 0, mm);
89        ExBottom(nn, 0, mm) = Ex(mm, nn);
90      }
91    }
92
93    /* ABC at top of grid */
94    for (mm = 0; mm < SizeX - 1; mm++) {
95      Ex(mm, SizeY - 1) = coef0 * (Ex(mm, SizeY - 3) + ExTop(0, 1, mm))
96        + coef1 * (ExTop(0, 0, mm) + ExTop(2, 0, mm)
97                   - Ex(mm, SizeY - 2) - ExTop(1, 1, mm))
98        + coef2 * ExTop(1, 0, mm) - ExTop(2, 1, mm);
99
100     /* memorize old fields */
101     for (nn = 0; nn < 3; nn++) {
102       ExTop(nn, 1, mm) = ExTop(nn, 0, mm);
103       ExTop(nn, 0, mm) = Ex(mm, SizeY - 1-nn);
104     }
105   }
106
107   return;
108 }
```

The contents of the file `tfsftez.c` are shown in Program 8.22. This closely follows the TFSF code that was used for the TM^z grid. Again, a 1D auxiliary grid is used to describe the incident field. The 1D grid is available via in the `Grid` pointer `g1` which is only visible to the functions in this file. Space for the structure is allocated in line 16. In the following line the contents of the 2D structure are copied to the 1D structure. This is done to set the size of the grid and the Courant number. Then, in line 18, the function `gridInit1d()` is called to complete the initialization of the 1D grid.

The function `tfsfUpdate()`, which starts on line 31, is called once per time-step. After ensuring that the initialization function has been called, the magnetic fields adjacent to the TFSF boundary are corrected. Following this, as shown starting on line 52, the magnetic field in the 1D grid is updated, then the 1D electric field is updated, then the source function is applied to the first node in the 1D grid, and finally the time-step of the 1D grid is incremented. Starting on line 58, the electric fields in the 2D grid adjacent to the TFSF boundary are corrected.

The header file `ezinctez.h` differs from `ezinc.h` used in the TM^z code only in that it includes `fdtd-macro-tez.h` instead of `fdtd-macro-tmz.h`. Hence it is not shown here nor is the code used to realize the source function which is a Ricker wavelet (which is also essentially unchanged from before).

Program 8.22 `tfsftez.c`: Implementation of a TFSF boundary for a TE^z grid. The incident field propagates in the x direction and an auxiliary 1D grid is used to compute the incident field.

```
/* TFSF implementation for a TEz grid. */

#include <string.h>  // for memcpy
#include "fdtd-macro-tez.h"
#include "fdtd-proto2.h"
#include "fdtd-alloc1.h"
#include "ezinctez.h"

static int firstX = 0, firstY, // indices for first point in TF region
           lastX, lastY;     // indices for last point in TF region

static Grid *g1;  // 1D auxilliary grid

void tfsfInit(Grid *g) {

  ALLOC_1D(g1, 1, Grid);       // allocate memory for 1D Grid
  memcpy(g1, g, sizeof(Grid)); // copy information from 2D array
  gridInit1d(g1);            // initialize 1d grid

  printf("Grid is %d by %d cell.\n", SizeX, SizeY);
  printf("Enter indices for first point in TF region: ");
  scanf(" %d %d", &firstX, &firstY);
  printf("Enter indices for last point in TF region: ");
  scanf(" %d %d", &lastX, &lastY);

  ezIncInit(g); // initialize source function

  return;
}

void tfsfUpdate(Grid *g) {
  int mm, nn;

  // check if tfsfInit() has been called
  if (firstX <= 0) {
    fprintf(stderr,
      "tfsfUpdate: tfsfInit must be called before tfsfUpdate.\n"
      "            Boundary location must be set to positive value.\n");
    exit(-1);
  }

  // correct Hz along left edge
  mm = firstX - 1;
  for (nn = firstY; nn < lastY; nn++)
    Hz(mm, nn) += Chze(mm, nn) * Ey1G(g1, mm + 1);

  // correct Hz along right edge
```

```
48    mm = lastX;
49    for (nn = firstY; nn < lastY; nn++)
50      Hz(mm, nn) -= Chze(mm, nn) * Ey1G(g1, mm);
51
52    updateH2d(g1);    // update 1D magnetic field
53    updateE2d(g1);    // update 1D electric field
54    Ey1G(g1, 0) = ezInc(TimeG(g1), 0.0); // set source node
55    TimeG(g1)++;      // increment time in 1D grid
56
57    // correct Ex along the bottom
58    nn = firstY;
59    for (mm = firstX; mm < lastX; mm++)
60      Ex(mm, nn) -= Cexh(mm, nn) * Hz1G(g1, mm);
61
62    // correct Ex along the top
63    nn = lastY;
64    for (mm = firstX; mm < lastX; mm++)
65      Ex(mm, nn) += Cexh(mm, nn) * Hz1G(g1, mm);
66
67    // correct Ey field along left edge
68    mm = firstX;
69    for (nn = firstY; nn < lastY; nn++)
70      Ey(mm, nn) += Ceyh(mm, nn) * Hz1G(g1, mm - 1);
71
72    // correct Ey field along right edge
73    mm = lastX;
74    for (nn = firstY; nn < lastY; nn++)
75      Ey(mm, nn) -= Ceyh(mm, nn) * Hz1G(g1, mm);
76
77    // no need to correct Ex along top and bottom since
78    // incident Ex is zero
79
80    return;
81  }
```

The function to initialize the 1D auxiliary grid is shown in Program 8.23. As was the case for the TM^z case, the grid is terminated on the right with a lossy layer that is 20 cells wide. The rest of the grid corresponds to free space. (The first node in the grid is the hard-wired source node and hence the left side of the grid does not need to be terminated.)

Program 8.23 `grid1dhz.c`: Initialization function used for the 1D auxiliary grid for the TE^z TFSF boundary.

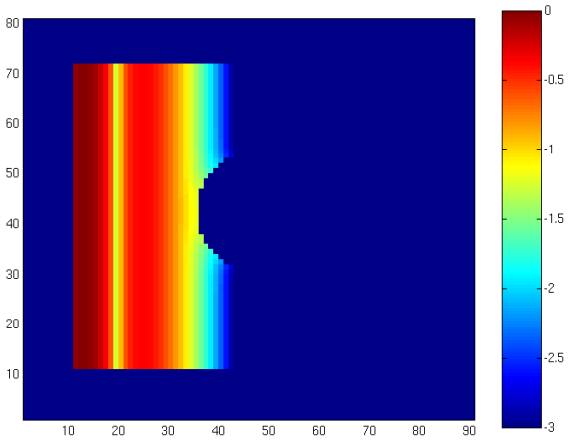
```
1  /* Create a 1D grid suitable for an auxilliary grid used as part of
2   * the implementation of a TFSF boundary in a TEz simulations. */
```

```
3
4  #include <math.h>
5  #include "fdtd-macro-tez.h"
6  #include "fdtd-alloc1.h"
7
8  #define NLOSS     20   // number of lossy cells at end of 1D grid
9  #define MAX_LOSS  0.35 // maximum loss factor in lossy layer
10
11 void gridInit1d(Grid *g) {
12   double imp0 = 377.0, depthInLayer = 0.0, lossFactor;
13   int mm;
14
15   SizeX += NLOSS;    // size of domain
16   Type = oneDGrid;   // set grid type
17
18   ALLOC_1D(g->hz,   SizeX - 1, double);
19   ALLOC_1D(g->chzh, SizeX - 1, double);
20   ALLOC_1D(g->chze, SizeX - 1, double);
21   ALLOC_1D(g->ey,   SizeX, double);
22   ALLOC_1D(g->ceye, SizeX, double);
23   ALLOC_1D(g->ceyh, SizeX, double);
24
25   /* set electric-field update coefficients */
26   for (mm = 0; mm < SizeX - 1; mm++) {
27     if (mm < SizeX - 1 - NLOSS) {
28       Ceye1(mm) = 1.0;
29       Ceyh1(mm) = Cdtds * imp0;
30       Chzh1(mm) = 1.0;
31       Chze1(mm) = Cdtds / imp0;
32     } else {
33       depthInLayer += 0.5;
34       lossFactor = MAX_LOSS * pow(depthInLayer / NLOSS, 2);
35       Ceye1(mm) = (1.0 - lossFactor) / (1.0 + lossFactor);
36       Ceyh1(mm) = Cdtds * imp0 / (1.0 + lossFactor);
37       depthInLayer += 0.5;
38       lossFactor = MAX_LOSS * pow(depthInLayer / NLOSS, 2);
39       Chzh1(mm) = (1.0 - lossFactor) / (1.0 + lossFactor);
40       Chze1(mm) = Cdtds / imp0 / (1.0 + lossFactor);
41     }
42   }
43
44   return;
45 }
```

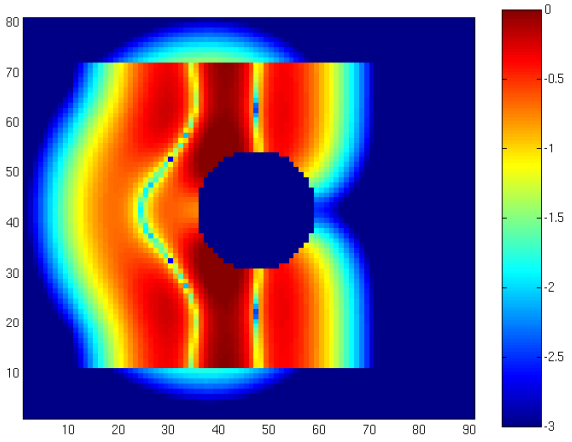
Figure 8.14 shows snapshots of the magnetic field throughout the computational domain at three different times. The snapshot in Fig. 8.14(a) was taken after 60 time steps. The leading edge of the incident pulse has just started to interact with the scatterer. No scattered fields are evident

in the SF region. The snapshot in Fig. 8.14(b) was taken after 100 time steps. The entire scatterer is now visible and scattered fields have just started to enter the SF region. The final snapshot was taken after 140 time steps.

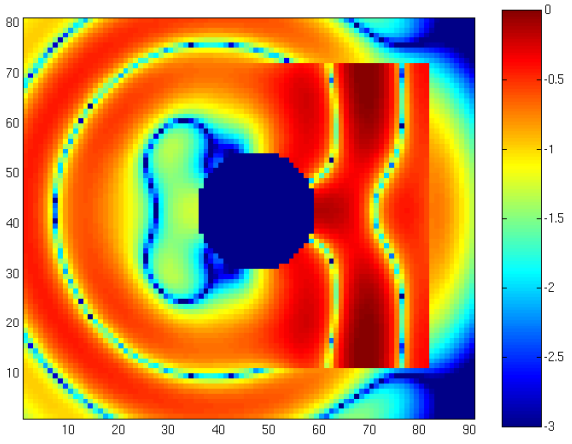
To obtain these snapshots, the snapshot code of Program 8.11 has to be slightly modified. Since we are now interested in obtaining H_z instead of E_z, the limits of the for-loops starting in line 68 of Program 8.11 would have to be changed to that which pertain to the H_z array. Furthermore, one would have to change `Ez(mm, nn)` in line 70 to `Hz(mm, nn)`. Because these changes are minor, the modified version of the program is not shown.



(a)



(b)



(c)

Figure 8.14: Pulsed TE^z illumination of a circular scatter. Display of the H_z field at time-steps (a) 60, (b) 100 and (c) 140. The field has been normalized by $1/377$ (i.e., the characteristic impedance of free space) and is shown with three decades of logarithmic scaling. The incident field is a Ricker wavelet discretized such that there are 30 points per wavelength at the most energetic frequency.

Chapter 9

Three-Dimensional FDTD

9.1 Introduction

With an understanding of the FDTD implementation of TE^z and TM^z grids, the additional steps needed to implement a three-dimensional (3D) grid are almost trivial. A 3D grid can be viewed as stacked layers of TE^z and TM^z grids which are offset a half spatial step in the z direction. The update equations for the H_z and E_z nodes are nearly identical to those which have been given already—the only difference is an additional index to specify the z location. The update equations for the other field components require slight changes to account for variations in the z direction (i.e., in the governing equations the partial derivative with respect to z is no longer zero).

We begin this chapter by discussing the implementation of 3D arrays in C. This is followed by details concerning the arrangement of nodes in 3D and the associated update equations. The chapter concludes with the code for an incremental dipole in a homogeneous space.

9.2 3D Arrays in C

For fields in a 3D space, it is, of course, natural to specify the location of a node using three indices representing the displacement in the x, y, and z directions. However, as was done for 2D grids, we will use a macro to translate the given indices into an offset into a 1D array. The memory associated with the 1D array will be allocated dynamically and the amount of memory will be precisely what is needed to store all the elements of the 3D "array." (We will refer to the macro as a 3D array since, other than the cleaner specification of the indices, its use in the code is indistinguishable from a traditional 3D array.)

For 3D arrays, incrementing the third index by one changes the variable being specified to the next consecutive variable in memory. Thinking of the third index as corresponding to the z direction, this implies that nodes that are adjacent to each other in the z direction are also adjacent to each other in memory. On the other hand, when the first or second index is incremented by one, that will *not* correspond to the next variable in memory. When the second index is incremented, one must move forward in memory an amount corresponding to the number of variables in the third dimension. For example, if the array size in the third dimension was 32 elements, then

Lecture notes by John Schneider. `fdtd-3d.tex`

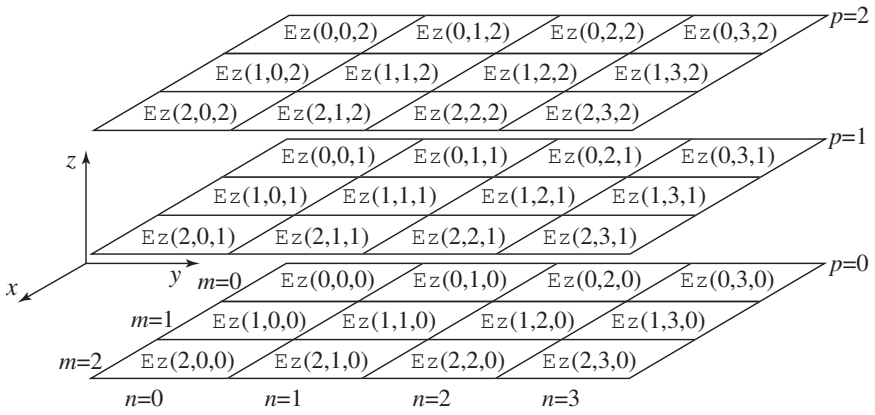


Figure 9.1: Depiction of elements of an array with dimensions $3 \times 4 \times 3$ in the x, y, and z directions, respectively. The indices m, n, and p, are used to specify the x, y, and z locations, respectively. The element at the "origin" has indices $(0, 0, 0)$ and is shown in the upper left corner of the bottom plane.

incrementing the second index by one would require that the offset in memory be advanced by 32. This is the same as in the 2D case where we can think of the size of the third dimension as corresponding to the number of columns (or, said another way, the number of elements in a row).

When the first index is incremented by one, the offset in memory must account for the array size in both the second and third dimension. To illustrate this, consider Fig. 9.1 which shows the elements of the 3D array `Ez`. The array is $3 \times 4 \times 3$, corresponding to the dimensions in the x, y and z directions. In reality, these elements will map to elements of a 1D array called `ez` which is shown in 9.2. Since `ez` is a 1D array, it takes a single index (or offset). Note that if one holds the m and n indices fixed (corresponding to the x and y directions) but increments the p index (corresponding to a movement in the z direction), the index of `ez` changes by one. However, if m and p are held fixed and n is incremented by one, the index of `ez` changed by 3 which correspond to the number of elements in the z directions. Finally, if n and p are held fixed but m is incremented by one, the index of `ez` changed by 12 which is the product of the dimensions in the y and z directions. Three-dimensional arrays can be thought of as a collection of 2D arrays. For the way in which we perform the indexing, the 2D arrays correspond to constant-x planes. Each of these 2D arrays must be large enough to hold the product of the number of elements along the y and z directions.

The construct we use for 3D arrays largely parallels that which was used for 2D arrays. The allocation macro `ALLOC_3D()` is shown in Fragment 9.1. The only difference between this and the allocation macros shown previously is the addition of another argument to specify the size of the array in the third dimension (this is the argument `NUMZ`). This dimension is multiplied by the other two dimensions and used as the first argument of `calloc()`.

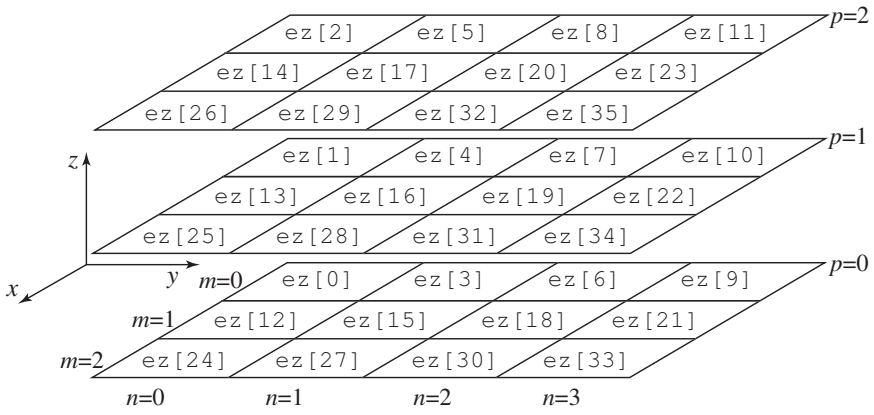


Figure 9.2: The 1D array `ez` is used to store the elements of `Ez`. The three indices for each elements of `Ez` shown in Fig. 9.1 map to the single index shown here.

Fragment 9.1 Macro for allocating memory for a 3D array.

```
1 #define ALLOC_3D(PNTR, NUMX, NUMY, NUMZ, TYPE)                          \
2     PNTR = (TYPE *)calloc((NUMX) * (NUMY) * (NUMZ), sizeof(TYPE)); \
3     if (!PNTR) {                                                      \
4       perror("ALLOC_3D");                                             \
5       fprintf(stderr,                                                 \
6           "Allocation failed for " #PNTR ".  Terminating...\n");    \
7       exit(-1);                                                       \
8     }
```

To illustrate the construction and use of a 3D array, the code in Fragment 9.2 shows how one could create a $6 \times 7 \times 8$ array. In this example the array dimensions are set in `#define`-statements in lines 1–3. Line 5 provides the macro `Ez()` which takes three (dummy) arguments. The preprocessor will replace all occurrences of `Ez()` with the expression involving `ez[]` shown at the right. The pointer `ez` is defined in line 6 and initially at run-time does not have any memory associated with it. However, after line 9 has executed `ez` will point to a block of memory that is sufficient to hold all the elements of the array and, at this point, `ez` can be treated as a 1D array (but we never use `ez` directly in the code—instead, we use the macro `Ez()` to access array elements). The nested for-loops starting at line 11 merely set each element equal to the product of the indices for that element. Note that this order of nesting is the one that should be used in practice: the inner-most loop should be over the z index and the outer-most loop should be over the x index. (This order helps minimize page faults and hence maximize performance.)

Fragment 9.2 Demonstration of the construction and manipulation of a $6 \times 7 \times 8$ array.

```
#define num_rows    8
#define num_columns 7
#define num_planes  6

#define Ez(M, N, P) ez[((M) * num_columns + (N)) * num_rows + (P)]

      ⋮

  double *ez;
  int m, n, p;

  ALLOC_3D(ez, num_planes, num_columns, num_rows, double);

  for (m = 0; m < num_planes; m++)
    for (n = 0; n < num_columns; n++)
      for (p = 0; p < num_rows; p++)
        Ez(m, n, p) = m * n * p;
```

9.3 Governing Equations and the 3D Grid

As has been the case previously, Ampere's and Faraday's laws are the relevant governing equations in constructing the FDTD algorithm. These equations are

$$-\sigma_m \mathbf{H} - \mu \frac{\partial \mathbf{H}}{\partial t} = \nabla \times \mathbf{E} = \begin{vmatrix} \hat{\mathbf{a}}_x & \hat{\mathbf{a}}_y & \hat{\mathbf{a}}_z \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ E_x & E_y & E_z \end{vmatrix}, \tag{9.1}$$

$$\sigma \mathbf{E} + \epsilon \frac{\partial \mathbf{E}}{\partial t} = \nabla \times \mathbf{H} = \begin{vmatrix} \hat{\mathbf{a}}_x & \hat{\mathbf{a}}_y & \hat{\mathbf{a}}_z \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial x} & \frac{\partial}{\partial z} \\ H_x & H_y & H_z \end{vmatrix}. \tag{9.2}$$

The components of these equations, when approximated by finite-differences at the appropriate points in space-time, yield the discretized update equations.

The necessary arrangement of nodes is show in Fig. 9.3. This grouping of six nodes can be considered the fundamental building block of a 3D grid. The following notation is used:

$$H_x(x,y,z,t) = H_x(m\Delta_x, n\Delta_y, p\Delta_z, q\Delta_t) = H_x^q[m,n,p]\,, \tag{9.3}$$
$$H_y(x,y,z,t) = H_y(m\Delta_x, n\Delta_y, p\Delta_z, q\Delta_t) = H_y^q[m,n,p]\,, \tag{9.4}$$
$$H_z(x,y,z,t) = H_z(m\Delta_x, n\Delta_y, p\Delta_z, q\Delta_t) = H_z^q[m,n,p]\,, \tag{9.5}$$
$$E_x(x,y,z,t) = E_x(m\Delta_x, n\Delta_y, p\Delta_z, q\Delta_t) = E_x^q[m,n,p]\,, \tag{9.6}$$
$$E_y(x,y,z,t) = E_y(m\Delta_x, n\Delta_y, p\Delta_z, q\Delta_t) = E_y^q[m,n,p]\,, \tag{9.7}$$
$$E_z(x,y,z,t) = E_z(m\Delta_x, n\Delta_y, p\Delta_z, q\Delta_t) = E_z^q[m,n,p]\,. \tag{9.8}$$

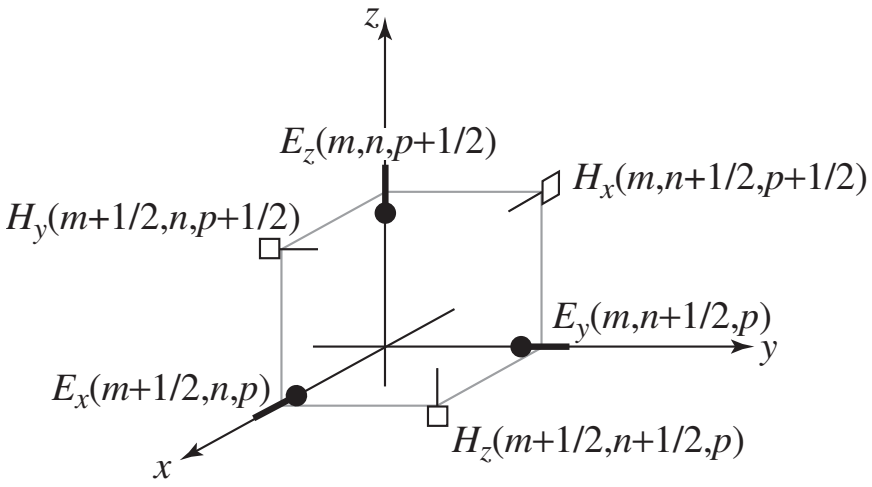


Figure 9.3: Arrangement of nodes in three dimensions. In a computer program all these nodes would have the same m, n, and p indices (the one-halves would be discarded from the equations—the offset would be understood). Electric-field nodes are displaced a half step in the direction in which they point while magnetic-field nodes are displaced a half step in the two directions they do not point. It is also implicitly understood that the electric- and magnetic-field nodes are offset from each other a half step in time.

In Fig. 9.3 the temporal location of the nodes is not specified. It is assumed the electric-field nodes exist at integer multiples of the time step and the magnetic-field nodes exists one-half of a temporal step away from the electric field nodes. As we will see when we implement the 3D algorithm in a computer program, the halves are suppressed and these six nodes will all have the same indices. Note that, for any given set of indices the electric-field nodes are displaced a half step in the direction in which they point while magnetic-field nodes are displaced a half step in the two directions they do not point.

Another view of a portion of the 3D grid is shown in Fig. 9.4. This type of depiction is typically call the Yee cube or Yee cell. This cube consists of electric-field nodes on the edges of the cube (hence four nodes of each electric-field component) and magnetic-field nodes on the faces (two nodes of each magnetic-field component). In a 3D grid one can shift the origin of this cube so that magnetic-field nodes are along the edges and electric-field nodes are on the faces. Although this is done by some authors, we will use the arrangement shown in Fig. 9.4.

With the arrangement of nodes shown in Figs. 9.3 and 9.4, the components of (9.1) and (9.2)

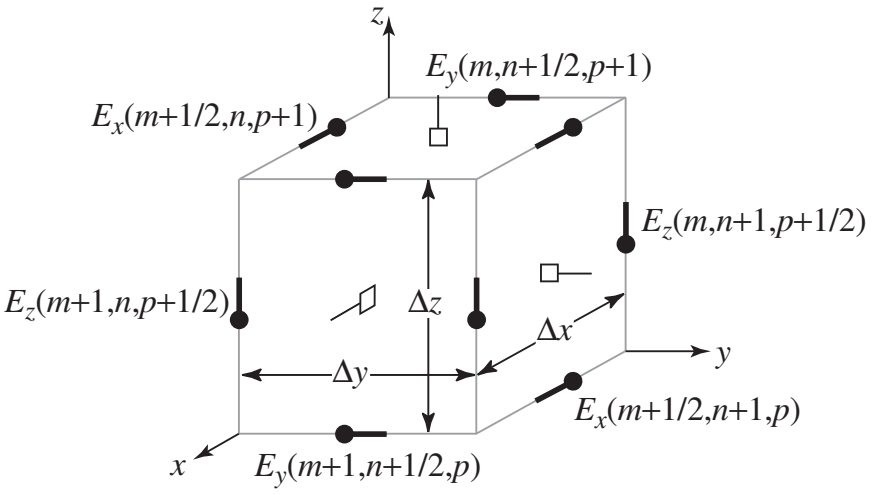


Figure 9.4: The nodes in a 3D FDTD grid are often drawn in the form of a Yee cube or Yee cell. In this depiction the nodes do not all have the same indices. As drawn here the cube would consist of four E_x nodes, four E_y nodes, and four E_z nodes, i.e., the electric fields are along the cube edges. Magnetic fields are on the cube faces and hence there would be two H_x nodes, two H_y nodes, and two H_z nodes.

expressed at the appropriate evaluation points are

$$-\sigma_m H_x - \mu\frac{\partial H_x}{\partial t} = \frac{\partial E_z}{\partial y} - \frac{\partial E_y}{\partial z}\bigg|_{x=m\Delta_x,y=(n+1/2)\Delta_y,z=(p+1/2)\Delta_z,t=q\Delta_t}, \tag{9.9}$$

$$-\sigma_m H_y - \mu\frac{\partial H_y}{\partial t} = \frac{\partial E_x}{\partial z} - \frac{\partial E_z}{\partial x}\bigg|_{x=(m+1/2)\Delta_x,y=n\Delta_y,z=(p+1/2)\Delta_z,t=q\Delta_t}, \tag{9.10}$$

$$-\sigma_m H_z - \mu\frac{\partial H_z}{\partial t} = \frac{\partial E_y}{\partial x} - \frac{\partial E_x}{\partial y}\bigg|_{x=(m+1/2)\Delta_x,y=(n+1/2)\Delta_y,z=p\Delta_z,t=q\Delta_t}, \tag{9.11}$$

$$\sigma E_x + \epsilon\frac{\partial E_x}{\partial t} = \frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z}\bigg|_{x=(m+1/2)\Delta_x,y=n\Delta_y,z=p\Delta_z,t=(q+1/2)\Delta_t}, \tag{9.12}$$

$$\sigma E_y + \epsilon\frac{\partial E_y}{\partial t} = \frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x}\bigg|_{x=m\Delta_x,y=(n+1/2)\Delta_y,z=p\Delta_z,t=(q+1/2)\Delta_t}, \tag{9.13}$$

$$\sigma E_z + \epsilon\frac{\partial E_z}{\partial t} = \frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y}\bigg|_{x=m\Delta_x,y=n\Delta_y,z=(p+1/2)\Delta_z,t=(q+1/2)\Delta_t}. \tag{9.14}$$

In these equations, ignoring loss for a moment, the temporal derivative of each field-component is always given by the spatial derivative of two components of the "other field." Also, the components of one field are related to the two orthogonal components of the other field. As has been done previously, the loss term can be approximated by the average of the field at two times steps.

Given our experience with 1- and 2D grids, the 3D update equations can be written simply by

inspection of the governing equations in the continuous world. The update equations are

$$H_x^{q+\frac{1}{2}}\left[m, n+\frac{1}{2}, p+\frac{1}{2}\right] = \frac{1-\frac{\sigma_m \Delta_t}{2\mu}}{1+\frac{\sigma_m \Delta_t}{2\mu}} H_x^{q-\frac{1}{2}}\left[m, n+\frac{1}{2}, p+\frac{1}{2}\right] + \frac{1}{1+\frac{\sigma_m \Delta_t}{2\mu}}\left(\frac{\Delta_t}{\mu \Delta_z}\left\{E_y^q\left[m, n+\frac{1}{2}, p+1\right] - E_y^q\left[m, n+\frac{1}{2}, p\right]\right\} - \frac{\Delta_t}{\mu \Delta_y}\left\{E_z^q\left[m, n+1, p+\frac{1}{2}\right] - E_z^q\left[m, n, p+\frac{1}{2}\right]\right\}\right), \quad (9.15)$$

$$H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}, n, p+\frac{1}{2}\right] = \frac{1-\frac{\sigma_m \Delta_t}{2\mu}}{1+\frac{\sigma_m \Delta_t}{2\mu}} H_y^{q-\frac{1}{2}}\left[m+\frac{1}{2}, n, p+\frac{1}{2}\right] + \frac{1}{1+\frac{\sigma_m \Delta_t}{2\mu}}\left(\frac{\Delta_t}{\mu \Delta_x}\left\{E_z^q\left[m+1, n, p+\frac{1}{2}\right] - E_z^q\left[m, n, p+\frac{1}{2}\right]\right\} - \frac{\Delta_t}{\mu \Delta_z}\left\{E_x^q\left[m+\frac{1}{2}, n, p+1\right] - E_x^q\left[m+\frac{1}{2}, n, p\right]\right\}\right), \quad (9.16)$$

$$H_z^{q+\frac{1}{2}}\left[m+\frac{1}{2}, n+\frac{1}{2}, p\right] = \frac{1-\frac{\sigma_m \Delta_t}{2\mu}}{1+\frac{\sigma_m \Delta_t}{2\mu}} H_z^{q-\frac{1}{2}}\left[m+\frac{1}{2}, n+\frac{1}{2}, p\right] + \frac{1}{1+\frac{\sigma_m \Delta_t}{2\epsilon}}\left(\frac{\Delta_t}{\mu \Delta_y}\left\{E_x^q\left[m+\frac{1}{2}, n+1, p\right] - E_x^q\left[m+\frac{1}{2}, n, p\right]\right\} - \frac{\Delta_t}{\epsilon \Delta_x}\left\{E_y^q\left[m+1, n+\frac{1}{2}, p\right] - E_y^q\left[m, n+\frac{1}{2}, p\right]\right\}\right). \quad (9.17)$$

$$E_x^{q+1}\left[m+\frac{1}{2}, n, p\right] = \frac{1-\frac{\sigma \Delta_t}{2\epsilon}}{1+\frac{\sigma \Delta_t}{2\epsilon}} E_x^q\left[m+\frac{1}{2}, n, p\right] + \frac{1}{1+\frac{\sigma \Delta_t}{2\epsilon}}\left(\frac{\Delta_t}{\epsilon \Delta_y}\left\{H_z^{q+\frac{1}{2}}\left[m+\frac{1}{2}, n+\frac{1}{2}, p\right] - H_z^{q+\frac{1}{2}}\left[m+\frac{1}{2}, n-\frac{1}{2}, p\right]\right\} - \frac{\Delta_t}{\epsilon \Delta_z}\left\{H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}, n, p+\frac{1}{2}\right] - H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}, n, p-\frac{1}{2}\right]\right\}\right), \quad (9.18)$$

$$E_y^{q+1}\left[m, n+\frac{1}{2}, p\right] = \frac{1-\frac{\sigma \Delta_t}{2\epsilon}}{1+\frac{\sigma \Delta_t}{2\epsilon}} E_y^q\left[m, n+\frac{1}{2}, p\right] + \frac{1}{1+\frac{\sigma \Delta_t}{2\epsilon}}\left(\frac{\Delta_t}{\epsilon \Delta_z}\left\{H_x^{q+\frac{1}{2}}\left[m, n+\frac{1}{2}, p+\frac{1}{2}\right] - H_x^{q+\frac{1}{2}}\left[m, n+\frac{1}{2}, p-\frac{1}{2}\right]\right\} - \frac{\Delta_t}{\epsilon \Delta_x}\left\{H_z^{q+\frac{1}{2}}\left[m+\frac{1}{2}, n+\frac{1}{2}, p\right] - H_z^{q+\frac{1}{2}}\left[m-\frac{1}{2}, n+\frac{1}{2}, p\right]\right\}\right), \quad (9.19)$$

$$E_z^{q+1}\left[m,n,p+\frac{1}{2}\right] = \frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}} E_z^q\left[m,n,p+\frac{1}{2}\right] + \frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\left(\frac{\Delta_t}{\epsilon\Delta_x}\left\{H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n,p+\frac{1}{2}\right] - H_y^{q+\frac{1}{2}}\left[m-\frac{1}{2},n,p+\frac{1}{2}\right]\right\} - \frac{\Delta_t}{\epsilon\Delta_y}\left\{H_x^{q+\frac{1}{2}}\left[m,n+\frac{1}{2},p+\frac{1}{2}\right] - H_x^{q+\frac{1}{2}}\left[m,n-\frac{1}{2},p+\frac{1}{2}\right]\right\}\right). \quad (9.20)$$

The coefficients in the update equations are assumed constant (in time) but may be functions of position. Consistent with the notation adopted previously and assuming a uniform grid in which $\Delta_x = \Delta_y = \Delta_z = \delta$, the magnetic-field update coefficients can be expressed as

$$C_{hxh}(m,n+1/2,p+1/2) = \left.\frac{1-\frac{\sigma_m\Delta_t}{2\mu}}{1+\frac{\sigma_m\Delta_t}{2\mu}}\right|_{m\delta,(n+1/2)\delta,(p+1/2)\delta}, \quad (9.21)$$

$$C_{hxe}(m,n+1/2,p+1/2) = \left.\frac{1}{1+\frac{\sigma_m\Delta_t}{2\mu}}\frac{\Delta_t}{\mu\delta}\right|_{m\delta,(n+1/2)\delta,(p+1/2)\delta}, \quad (9.22)$$

$$C_{hyh}(m+1/2,n,p+1/2) = \left.\frac{1-\frac{\sigma_m\Delta_t}{2\mu}}{1+\frac{\sigma_m\Delta_t}{2\mu}}\right|_{(m+1/2)\delta,n\delta,(p+1/2)\delta}, \quad (9.23)$$

$$C_{hye}(m+1/2,n,p+1/2) = \left.\frac{1}{1+\frac{\sigma_m\Delta_t}{2\mu}}\frac{\Delta_t}{\mu\delta}\right|_{(m+1/2)\delta,n\delta,(p+1/2)\delta}, \quad (9.24)$$

$$C_{hxh}(m+1/2,n+1/2,p) = \left.\frac{1-\frac{\sigma_m\Delta_t}{2\mu}}{1+\frac{\sigma_m\Delta_t}{2\mu}}\right|_{(m+1/2)\delta,(n+1/2)\delta,p\delta}, \quad (9.25)$$

$$C_{hxe}(m+1/2,n+1/2,p) = \left.\frac{1}{1+\frac{\sigma_m\Delta_t}{2\mu}}\frac{\Delta_t}{\mu\delta}\right|_{(m+1/2)\delta,(n+1/2)\delta,p\delta}. \quad (9.26)$$

For the electric-field update equations the coefficients are

$$C_{exe}(m+1/2,n,p) = \left.\frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}}\right|_{(m+1/2)\delta,n\delta,p\delta}, \tag{9.27}$$

$$C_{exh}(m+1/2,n,p) = \left.\frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\frac{\Delta_t}{\epsilon\delta}\right|_{(m+1/2)\delta,n\delta,p\delta}, \tag{9.28}$$

$$C_{eye}(m,n+1/2,p) = \left.\frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}}\right|_{m\delta,(n+1/2)\delta,p\delta}, \tag{9.29}$$

$$C_{eyh}(m,n+1/2,p) = \left.\frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\frac{\Delta_t}{\epsilon\delta}\right|_{m\delta,(n+1/2)\delta,p\delta}, \tag{9.30}$$

$$C_{eze}(m,n) = \left.\frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}}\right|_{m\delta,n\delta,(p+1/2)\delta}, \tag{9.31}$$

$$C_{ezh}(m,n,p+1/2) = \left.\frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\frac{\Delta_t}{\epsilon\delta}\right|_{m\delta,n\delta,(p+1/2)\delta}. \tag{9.32}$$

These coefficients can be related to the Courant number $c\Delta_t/\delta$. For a uniform grid in three dimensions the Courant limit is $1/\sqrt{3}$. There are rigorous derivations of this limit but there is also a simple empirical argument. It takes three time-steps to communicate information across the diagonal of a cube in the grid. The distance traveled across this diagonal is $\sqrt{3}\delta$. To ensure stability we must have that the distance traveled in the continuous world over these three time steps is less than the distance over which the grid can communicate information. Thus, we must have $c3\Delta_t \leq \sqrt{3}\delta$ or, rearranging, $S_c \leq 1/\sqrt{3}$.

As has been done previously, the explicit reference to time is dropped. Additionally, so that the indexing can be easily handled within a computer program, the spatial offsets of one-half are dropped explicitly but left implicitly understood. Thus, all one-halves are discarded from the left side of the update equations. Nodes on the right side of the equation will also have the one-halves dropped if the node is within the same group of nodes as the node being updated (where a group of nodes is as shown in Fig. 9.3). However, if the node on the right side is contained within a group that is a neighbor to the group that contains the node being updated, the one-half is replaced with a one. To illustrate further the grouping of nodes in three dimensions, Fig. 9.5 shows six groups of nodes and the corresponding set of indices for each group. The update-equation coefficients are evaluated at a point that is collocated with the node being updated. Thus, the 3D update equations can be written (assuming a suitable collection of macros which will be considered later):

```
Hx(m, n, p) = Chxh(m, n, p) * Hx(m, n, p) +
   Chxe(m, n, p) * ((Ey(m, n, p + 1) - Ey(m, n, p)) -
                    (Ez(m, n + 1, p) - Ez(m, n, p)));
Hy(m, n, p) = Chyh(m, n, p) * Hy(m, n, p) +
   Chye(m, n, p) * ((Ez(m + 1, n, p) - Ez(m, n, p)) -
                    (Ex(m, n, p + 1) - Ex(m, n, p)));
Hz(m, n, p) = Chzh(m, n, p) * Hz(m, n, p) +
```

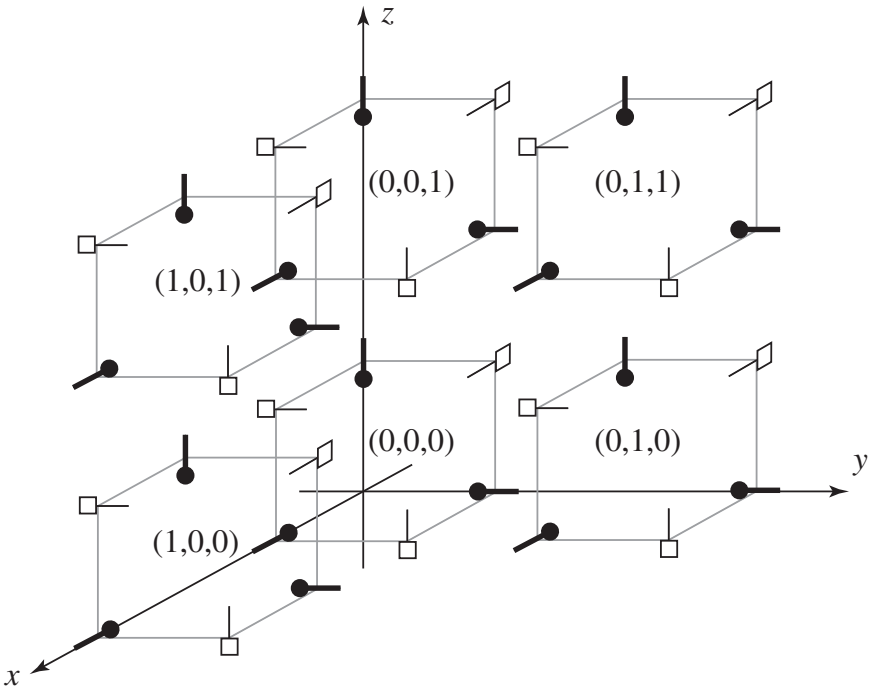


Figure 9.5: Arrangement of six groups of nodes where all of the nodes within the group have the same set of indices. The nodes in a group are joined by gray lines and their indices are shown as an ordered triplet in the center of the group.

```
   Chze(m, n, p) * ((Ex(m, n + 1, p) - Ex(m, n, p)) -
                    (Ey(m + 1, n, p) - Ey(m, n, p)));
Ex(m, n, p) = Cexe(m, n, p) * Ex(m, n, p) +
   Cexh(m, n, p) * ((Hz(m, n, p) - Hz(m, n - 1, p)) -
                    (Hy(m, n, p) - Hy(m, n, p - 1)));
Ey(m, n, p) = Ceye(m, n, p) * Ey(m, n, p) +
   Ceyh(m, n, p) * ((Hx(m, n, p) - Hx(m, n, p - 1)) -
                    (Hz(m, n, p) - Hz(m - 1, n, p)));
Ez(m, n, p) = Ceze(m, n, p) * Ez(m, n, p) +
   Cezh(m, n, p) * ((Hy(m, n, p) - Hy(m - 1, n, p)) -
                    (Hx(m, n, p) - Hx(m, n - 1, p)));
```

In our construction of 3D grids, the faces of the grid will always be terminated such that there are two electric-field components tangential to the face and one magnetic field normal to it. This is illustrated in Fig. 9.6. The computational domain shown in this figure is one which we describe as having dimensions of $5 \times 9 \times 7$ in the x, y, and z directions, respectively. Even though we call this a $5 \times 9 \times 7$ grid, none of the arrays associated with this computational domain actually have these

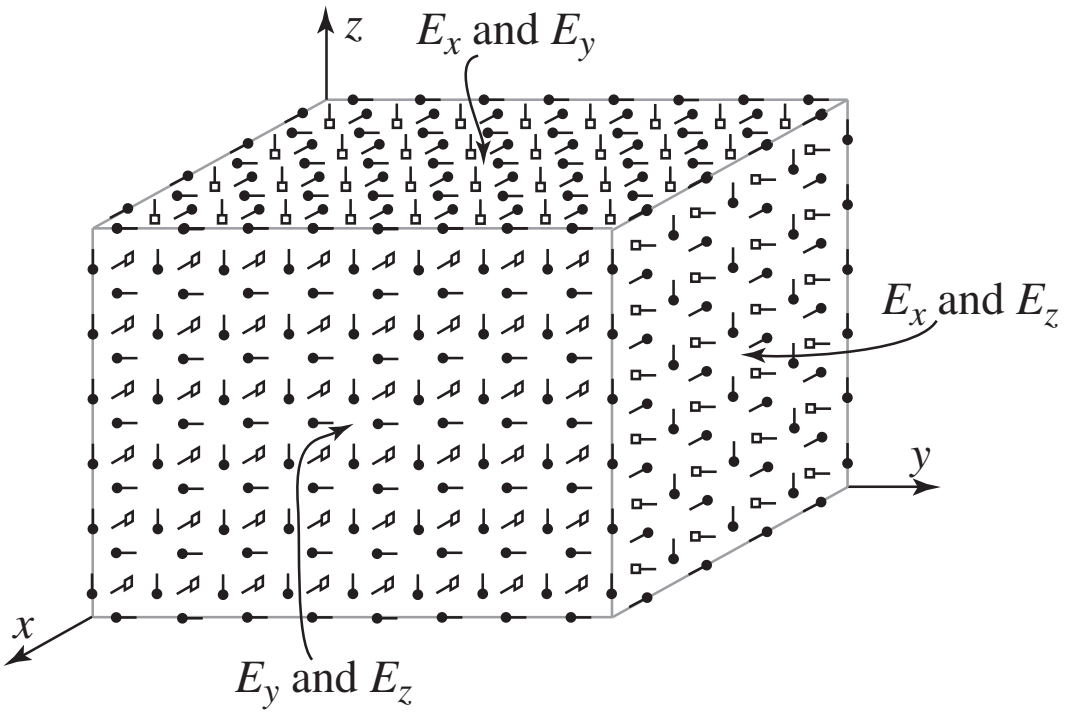


Figure 9.6: Faces of a computational domain which is $5 \times 9 \times 7$ in the x, y, and z directions, respectively. On the constant-x face the tangential fields are E_y and E_z, on the constant-y face they are E_x and E_z, and on the constant-z face they are E_x and E_y. There are also magnetic-field nodes which exist on these faces but their orientation is normal to the face.

dimensions! The fields of a computational domain that is $M \times N \times P$ would have dimensions of

$$E_x: \quad (M-1) \times N \times P \tag{9.33}$$
$$E_y: \quad M \times (N-1) \times P \tag{9.34}$$
$$E_z: \quad M \times N \times (P-1) \tag{9.35}$$
$$H_x: \quad M \times (N-1) \times (P-1) \tag{9.36}$$
$$H_y: \quad (M-1) \times N \times (P-1) \tag{9.37}$$
$$H_z: \quad (M-1) \times (N-1) \times P \tag{9.38}$$

Note that the electric fields have one less element in the direction in which they point than the nominal size of this grid. This is because of the inherent displacement of electric-field nodes in the direction in which they point. Rather than having an additional node essentially sticking beyond the rest of the grid, the array is truncated in this direction. Recall that the displacement of the magnetic-field nodes is in the two directions in which they do not point. Thus the magnetic-field arrays are truncated in the two directions they do not point. In terms of Yee cubes, an $M \times N \times P$ grid would consists of $(M-1) \times (N-1) \times (P-1)$ complete cubes.

9.4 3D Example

Here we provide the code to implement a simple 3D simulation in which a short dipole source is embedded in a homogeneous domain. The dipole is merely an additive source applied to an E_x node in the center of the grid. First-order ABC's are used to terminate the grid. Since there are two tangential electric fields on each face of the computational domain, the ABC must be applied to two fields per face.

The `main()` function is shown in Program 9.3. The overall structure is little changed from previous simulations. The ABC, the grid, the source function, and the snapshot code are initialized by calling initialization functions outside of the time-stepping loop. Within the time-stepping loop the magnetic fields are updated, the electric fields are updated, the source function is applied to the E_x node at the center of the grid, the ABC is applied, and then, assuming it is the appropriate time step, a snapshot is taken. Actually, as we will see, two different snapshots are taken. There are many ways one might choose to display these 3D vector fields. We will merely record one field component over a 2D plane (or perhaps multiple planes).

Program 9.3 `3ddemo.c` 3D simulation of an electric dipole realized with an additive source applied to an E_x node.

```
/* 3D simulation with dipole source at center of grid. */

#include "fdtd-alloc.h"
#include "fdtd-macro.h"
#include "fdtd-proto.h"
#include "ezinc.h"

int main()
{
  Grid *g;

  ALLOC_1D(g, 1, Grid); // allocate memory for grid structure
  gridInit(g);          // initialize 3D grid

  abcInit(g);           // initialize ABC
  ezIncInit(g);
  snapshot3dInit(g);    // initialize snapshots

  /* do time stepping */
  for (Time = 0; Time < MaxTime; Time++) {
    updateH(g);         // update magnetic fields
    updateE(g);         // update electric fields
    Ex((SizeX - 1) / 2, SizeY / 2, SizeZ / 2) += ezInc(Time, 0.0);
    abc(g);             // apply ABC
    snapshot3d(g);      // take a snapshot (if appropriate)
  } // end of time-stepping

```

```
28   return 0;
29 }
```

The code used to realize the source function, i.e., the Ricker wavelet, is unchanged from before and hence not shown (ref. Program 8.10). The header `fdtd-alloc.h` merely provides the three allocation macros `ALLOC_1D()`, `ALLOC_2D()`, and `ALLOC_3D()` and hence is not shown here. Similarly, the header `fdtd-grid1.h`, which defines the elements of the `Grid` structure, is unchanged from before and thus not shown (ref. Program 8.3). The header `fdtd-proto.h` provides the prototypes for the various functions. Since these prototypes simply show that each function takes a single argument (i.e., a pointer to a `Grid` structure), that header file is also not shown.

The header `fdtd-macro.h` shown in Program 9.4 provides macros for all the types of grids we have considered so far. In this particular program we only need the macros for the 3D arrays, but having created this collection of macros we are well prepared to use it, unchanged, to tackle a wide variety of FDTD problems. As was done in the previous chapter, there are macros which assume that the `Grid` structure is named `g` while there is another set of macros that allows the name of the `Grid` to be specified explicitly.

Program 9.4 `fdtd-macro.h` Header that provides the macros to access the elements of any of the arrays that have been considered thus far. One set of macros assumes the name of the `Grid` is `g`. Another set allows the name of the `Grid` to be specified as an additional argument.

```
1  #ifndef _FDTD_MACRO_H
2  #define _FDTD_MACRO_H
3
4  #include "fdtd-grid1.h"
5
6  /* macros that permit the "Grid" to be specified */
7  /* one-dimensional grid */
8  #define Hy1G(G, M)      G->hy[M]
9  #define Chyh1G(G, M)    G->chyh[M]
10 #define Chye1G(G, M)    G->chye[M]
11
12 #define Ez1G(G, M)      G->ez[M]
13 #define Ceze1G(G, M)    G->ceze[M]
14 #define Cezh1G(G, M)    G->cezh[M]
15
16 /* TMz grid */
17 #define Hx2G(G, M, N)    G->hx[(M) * (SizeYG(G) - 1) + N]
18 #define Chxh2G(G, M, N) G->chxh[(M) * (SizeYG(G) - 1) + N]
19 #define Chxe2G(G, M, N) G->chxe[(M) * (SizeYG(G) - 1) + N]
20
21 #define Hy2G(G, M, N)    G->hy[(M) * SizeYG(G) + N]
22 #define Chyh2G(G, M, N) G->chyh[(M) * SizeYG(G) + N]
```

```
23 #define Chye2G(G, M, N) G->chye[(M) * SizeYG(G) + N]
24
25 #define Ez2G(G, M, N)   G->ez[(M) * SizeYG(G) + N]
26 #define Ceze2G(G, M, N) G->ceze[(M) * SizeYG(G) + N]
27 #define Cezh2G(G, M, N) G->cezh[(M) * SizeYG(G) + N]
28
29 /* TEz grid */
30 #define Ex2G(G, M, N)   G->ex[(M) * SizeYG(G) + N]
31 #define Cexe2G(G, M, N) G->cexe[(M) * SizeYG(G) + N]
32 #define Cexh2G(G, M, N) G->cexh[(M) * SizeYG(G) + N]
33
34 #define Ey2G(G, M, N)   G->ey[(M) * (SizeYG(G) - 1) + N]
35 #define Ceye2G(G, M, N) G->ceye[(M) * (SizeYG(G) - 1) + N]
36 #define Ceyh2G(G, M, N) G->ceyh[(M) * (SizeYG(G) - 1) + N]
37
38 #define Hz2G(G, M, N)   G->hz[(M) * (SizeYG(G) - 1) + N]
39 #define Chzh2G(G, M, N) G->chzh[(M) * (SizeYG(G) - 1) + N]
40 #define Chze2G(G, M, N) G->chze[(M) * (SizeYG(G) - 1) + N]
41
42 /* 3D grid */
43 #define HxG(G, M, N, P)   G->hx[((M) * (SizeYG(G) - 1) + N) * (SizeZG(G) - 1) + P]
44 #define ChxhG(G, M, N, P) G->chxh[((M) * (SizeYG(G) - 1) + N) * (SizeZG(G) - 1) + P
45 #define ChxeG(G, M, N, P) G->chxe[((M) * (SizeYG(G) - 1) + N) * (SizeZG(G) - 1) + P
46
47 #define HyG(G, M, N, P)   G->hy[((M) * SizeYG(G) + N) * (SizeZG(G) - 1) + P]
48 #define ChyhG(G, M, N, P) G->chyh[((M) * SizeYG(G) + N) * (SizeZG(G) - 1) + P]
49 #define ChyeG(G, M, N, P) G->chye[((M) * SizeYG(G) + N) * (SizeZG(G) - 1) + P]
50
51 #define HzG(G, M, N, P)   G->hz[((M) * (SizeYG(G) - 1) + N) * SizeZG(G) + P]
52 #define ChzhG(G, M, N, P) G->chzh[((M) * (SizeYG(G) - 1) + N) * SizeZG(G) + P]
53 #define ChzeG(G, M, N, P) G->chze[((M) * (SizeYG(G) - 1) + N) * SizeZG(G) + P]
54
55 #define ExG(G, M, N, P)   G->ex[((M) * SizeYG(G) + N) * SizeZG(G) + P]
56 #define CexeG(G, M, N, P) G->cexe[((M) * SizeYG(G) + N) * SizeZG(G) + P]
57 #define CexhG(G, M, N, P) G->cexh[((M) * SizeYG(G) + N) * SizeZG(G) + P]
58
59 #define EyG(G, M, N, P)   G->ey[((M) * (SizeYG(G) - 1) + N) * SizeZG(G) + P]
60 #define CeyeG(G, M, N, P) G->ceye[((M) * (SizeYG(G) - 1) + N) * SizeZG(G) + P]
61 #define CeyhG(G, M, N, P) G->ceyh[((M) * (SizeYG(G) - 1) + N) * SizeZG(G) + P]
62
63 #define EzG(G, M, N, P)   G->ez[((M) * SizeYG(G) + N) * (SizeZG(G) - 1) + P]
64 #define CezeG(G, M, N, P) G->ceze[((M) * SizeYG(G) + N) * (SizeZG(G) - 1) + P]
65 #define CezhG(G, M, N, P) G->cezh[((M) * SizeYG(G) + N) * (SizeZG(G) - 1) + P]
66
67 #define SizeXG(G)      G->sizeX
68 #define SizeYG(G)      G->sizeY
69 #define SizeZG(G)      G->sizeZ
```

```
70  #define TimeG(G)        G->time
71  #define MaxTimeG(G)     G->maxTime
72  #define CdtdsG(G)       G->cdtds
73  #define TypeG(G)        G->type
74
75  /* macros that assume the "Grid" is "g" */
76  /* one-dimensional grid */
77  #define Hy1(M)      Hy1G(g, M)
78  #define Chyh1(M)    Chyh1G(g, M)
79  #define Chye1(M)    Chye1G(g, M)
80
81  #define Ez1(M)      Ez1G(g, M)
82  #define Ceze1(M)    Ceze1G(g, M)
83  #define Cezh1(M)    Cezh1G(g, M)
84
85  /* TMz grid */
86  #define Hx2(M, N)    Hx2G(g, M, N)
87  #define Chxh2(M, N) Chxh2G(g, M, N)
88  #define Chxe2(M, N) Chxe2G(g, M, N)
89
90  #define Hy2(M, N)    Hy2G(g, M, N)
91  #define Chyh2(M, N) Chyh2G(g, M, N)
92  #define Chye2(M, N) Chye2G(g, M, N)
93
94  #define Ez2(M, N)    Ez2G(g, M, N)
95  #define Ceze2(M, N) Ceze2G(g, M, N)
96  #define Cezh2(M, N) Cezh2G(g, M, N)
97
98  /* TEz grid */
99  #define Hz2(M, N)    Hz2G(g, M, N)
100 #define Chzh2(M, N) Chzh2G(g, M, N)
101 #define Chze2(M, N) Chze2G(g, M, N)
102
103 #define Ex2(M, N)    Ex2G(g, M, N)
104 #define Cexe2(M, N) Cexe2G(g, M, N)
105 #define Cexh2(M, N) Cexh2G(g, M, N)
106
107 #define Ey2(M, N)    Ey2G(g, M, N)
108 #define Ceye2(M, N) Ceye2G(g, M, N)
109 #define Ceyh2(M, N) Ceyh2G(g, M, N)
110
111 /* 3D grid */
112 #define Hx(M, N, P)    HxG(g, M, N, P)
113 #define Chxh(M, N, P) ChxhG(g, M, N, P)
114 #define Chxe(M, N, P) ChxeG(g, M, N, P)
115
116 #define Hy(M, N, P)    HyG(g, M, N, P)
```

```
#define Chyh(M, N, P) ChyhG(g, M, N, P)
#define Chye(M, N, P) ChyeG(g, M, N, P)

#define Hz(M, N, P)   HzG(g, M, N, P)
#define Chzh(M, N, P) ChzhG(g, M, N, P)
#define Chze(M, N, P) ChzeG(g, M, N, P)

#define Ex(M, N, P)   ExG(g, M, N, P)
#define Cexe(M, N, P) CexeG(g, M, N, P)
#define Cexh(M, N, P) CexhG(g, M, N, P)

#define Ey(M, N, P)   EyG(g, M, N, P)
#define Ceye(M, N, P) CeyeG(g, M, N, P)
#define Ceyh(M, N, P) CeyhG(g, M, N, P)

#define Ez(M, N, P)   EzG(g, M, N, P)
#define Ceze(M, N, P) CezeG(g, M, N, P)
#define Cezh(M, N, P) CezhG(g, M, N, P)

#define SizeX       SizeXG(g)
#define SizeY       SizeYG(g)
#define SizeZ       SizeZG(g)
#define Time        TimeG(g)
#define MaxTime     MaxTimeG(g)
#define Cdtds       CdtdsG(g)
#define Type        TypeG(g)

#endif
```

The file `update3d.c` is shown in Program 9.5. When `updateE()` or `updateH()` are called they begin by checking the `Type` of the grid. These same functions can be called whether updating a 1D, 2D, or 3D grid. However, for the 1D grid there is the assumption that one is dealing with a z-polarized wave and for 2D propagation one has either TM^z- or TE^z-polarization. (A rotation of coordinate systems can be used to map any 1D simulation to one that is z-polarized or any 2D simulation to one that is either TE^z- or TM^z-polarized.)

Program 9.5 `update3d.c` Function that can be used to update any of the grids.

```
#include "fdtd-macro.h"
#include <stdio.h>

/* update magnetic field */
void updateH(Grid *g) {
  int mm, nn, pp;

```

```
8   if (Type == oneDGrid) {
9
10    for (mm = 0; mm < SizeX - 1; mm++)
11      Hy1(mm) = Chyh1(mm) * Hy1(mm)
12        + Chye1(mm) * (Ez1(mm + 1) - Ez1(mm));
13
14  } else if (Type == tmZGrid) {
15
16    for (mm = 0; mm < SizeX; mm++)
17      for (nn = 0; nn < SizeY - 1; nn++)
18        Hx2(mm, nn) = Chxh2(mm, nn) * Hx2(mm, nn)
19          - Chxe2(mm, nn) * (Ez2(mm, nn + 1) - Ez2(mm, nn));
20
21    for (mm = 0; mm < SizeX - 1; mm++)
22      for (nn = 0; nn < SizeY; nn++)
23        Hy2(mm, nn) = Chyh2(mm, nn) * Hy2(mm, nn)
24          + Chye2(mm, nn) * (Ez2(mm + 1, nn) - Ez2(mm, nn));
25
26  } else if (Type == teZGrid) {
27
28    for(mm = 0; mm < SizeX - 1; mm++)
29      for(nn = 0; nn < SizeY - 1; nn++)
30        Hz2(mm, nn) = Chzh2(mm, nn) * Hz2(mm, nn) -
31          Chze2(mm, nn) * ((Ey2(mm + 1, nn) - Ey2(mm, nn)) -
32                           (Ex2(mm, nn + 1) - Ex2(mm, nn)));
33
34  } else if (Type == threeDGrid) {
35
36    for (mm = 0; mm < SizeX; mm++)
37      for (nn = 0; nn < SizeY - 1; nn++)
38        for (pp = 0; pp < SizeZ - 1; pp++)
39          Hx(mm, nn, pp) = Chxh(mm, nn, pp) * Hx(mm, nn, pp) +
40            Chxe(mm, nn, pp) * ((Ey(mm, nn, pp + 1) - Ey(mm, nn, pp)) -
41                                (Ez(mm, nn + 1, pp) - Ez(mm, nn, pp)));
42
43    for (mm = 0; mm < SizeX - 1; mm++)
44      for (nn = 0; nn < SizeY; nn++)
45        for (pp = 0; pp < SizeZ - 1; pp++)
46          Hy(mm, nn, pp) = Chyh(mm, nn, pp) * Hy(mm, nn, pp) +
47            Chye(mm, nn, pp) * ((Ez(mm + 1, nn, pp) - Ez(mm, nn, pp)) -
48                                (Ex(mm, nn, pp + 1) - Ex(mm, nn, pp)));
49
50    for (mm = 0; mm < SizeX - 1; mm++)
51      for (nn = 0; nn < SizeY - 1; nn++)
52        for (pp = 0; pp < SizeZ; pp++)
53          Hz(mm, nn, pp) = Chzh(mm, nn, pp) * Hz(mm, nn, pp) +
54            Chze(mm, nn, pp) * ((Ex(mm, nn + 1, pp) - Ex(mm, nn, pp)) -
```

```
55                                    (Ey(mm + 1, nn, pp) - Ey(mm, nn, pp)));
56
57    } else {
58      fprintf(stderr, "updateH: Unknown grid type.  Terminating...\n");
59    }
60
61    return;
62  }  /* end updateH() */
63
64
65  /* update electric field */
66  void updateE(Grid *g) {
67    int mm, nn, pp;
68
69    if (Type == oneDGrid) {
70
71      for (mm = 1; mm < SizeX - 1; mm++)
72        Ez1(mm) = Ceze1(mm) * Ez1(mm)
73          + Cezh1(mm) * (Hy1(mm) - Hy1(mm - 1));
74
75    } else if (Type == tmZGrid) {
76
77      for (mm = 1; mm < SizeX - 1; mm++)
78        for (nn = 1; nn < SizeY - 1; nn++)
79          Ez2(mm, nn) = Ceze2(mm, nn) * Ez2(mm, nn) +
80            Cezh2(mm, nn) * ((Hy2(mm, nn) - Hy2(mm - 1, nn)) -
81                             (Hx2(mm, nn) - Hx2(mm, nn - 1)));
82
83    } else if (Type == teZGrid) {
84
85      for(mm = 1; mm < SizeX - 1; mm++)
86        for(nn = 1; nn < SizeY - 1; nn++)
87          Ex2(mm, nn) = Cexe2(mm, nn) * Ex2(mm, nn) +
88            Cexh2(mm, nn) * (Hz2(mm, nn) - Hz2(mm, nn - 1));
89
90      for(mm = 1; mm < SizeX - 1; mm++)
91        for(nn = 1; nn < SizeY - 1; nn++)
92          Ey2(mm, nn) = Ceye2(mm, nn) * Ey2(mm, nn) -
93            Ceyh2(mm, nn) * (Hz2(mm, nn) - Hz2(mm - 1, nn));
94
95    } else if (Type == threeDGrid) {
96
97      for (mm = 0; mm < SizeX - 1; mm++)
98        for (nn = 1; nn < SizeY - 1; nn++)
99          for (pp = 1; pp < SizeZ - 1; pp++)
100           Ex(mm, nn, pp) = Cexe(mm, nn, pp) * Ex(mm, nn, pp) +
101             Cexh(mm, nn, pp) * ((Hz(mm, nn, pp) - Hz(mm, nn - 1, pp)) -
```

```
                                (Hy(mm, nn, pp) - Hy(mm, nn, pp - 1)));

    for (mm = 1; mm < SizeX - 1; mm++)
      for (nn = 0; nn < SizeY - 1; nn++)
        for (pp = 1; pp < SizeZ - 1; pp++)
          Ey(mm, nn, pp) = Ceye(mm, nn, pp) * Ey(mm, nn, pp) +
            Ceyh(mm, nn, pp) * ((Hx(mm, nn, pp) - Hx(mm, nn, pp - 1)) -
                                (Hz(mm, nn, pp) - Hz(mm - 1, nn, pp)));

    for (mm = 1; mm < SizeX - 1; mm++)
      for (nn = 1; nn < SizeY - 1; nn++)
        for (pp = 0; pp < SizeZ - 1; pp++)
          Ez(mm, nn, pp) = Ceze(mm, nn, pp) * Ez(mm, nn, pp) +
            Cezh(mm, nn, pp) * ((Hy(mm, nn, pp) - Hy(mm - 1, nn, pp)) -
                                (Hx(mm, nn, pp) - Hx(mm, nn - 1, pp)));

  } else {
    fprintf(stderr, "updateE: Unknown grid type.  Terminating...\n");
  }

  return;
}  /* end updateE() */
```

The code to realize the first-order ABC is shown in Program 9.6. A first-order ABC requires that a single "old" value be recorded for each electric field that is tangential to a face of the grid. There are two tangential components per face. For example, at the "$x = 0$" face, E_y and E_z are the tangential components. These fields are stored in arrays named `Eyx0(n, p)` and `Ezx0(n, p)`. The "`x0`" part of the name specifies that these values are at the start of the grid in the x-direction. Since these old fields are recorded over a constant-x face, only the indices corresponding to the y and z directions are specified (hence these arrays only take two indices). The array `Eyx1(n, p)` and `Ezx1(n, p)` correspond to the tangential field at the end of the grid in the x-direction. There are similarly named arrays for the other two directions.

Program 9.6 `abc3dfirst.c` The code used to implement a first-order ABC on each face of the 3D domain.

```
#include "fdtd-alloc.h"
#include "fdtd-macro.h"

/* Macros to access stored "old" value */
#define Eyx0(N, P) eyx0[(N) * (SizeZ) + (P)]
#define Ezx0(N, P) ezx0[(N) * (SizeZ - 1) + (P)]
#define Eyx1(N, P) eyx1[(N) * (SizeZ) + (P)]
#define Ezx1(N, P) ezx1[(N) * (SizeZ - 1) + (P)]

```

```
10 #define Exy0(M, P) exy0[(M) * (SizeZ) + (P)]
11 #define Ezy0(M, P) ezy0[(M) * (SizeZ - 1) + (P)]
12 #define Exy1(M, P) exy1[(M) * (SizeZ) + (P)]
13 #define Ezy1(M, P) ezy1[(M) * (SizeZ - 1) + (P)]
14
15 #define Exz0(M, N) exz0[(M) * (SizeY) + (N)]
16 #define Eyz0(M, N) eyz0[(M) * (SizeY - 1) + (N)]
17 #define Exz1(M, N) exz1[(M) * (SizeY) + (N)]
18 #define Eyz1(M, N) eyz1[(M) * (SizeY - 1) + (N)]
19
20 /* global variables not visible outside of this package */
21 static double abccoef = 0.0;
22 static double *exy0, *exy1, *exz0, *exz1,
23   *eyx0, *eyx1, *eyz0, *eyz1,
24   *ezx0, *ezx1, *ezy0, *ezy1;
25
26 /* initialization function */
27 void abcInit(Grid *g)
28 {
29
30   abccoef = (Cdtds - 1.0) / (Cdtds + 1.0);
31
32   /* allocate memory for ABC arrays */
33   ALLOC_2D(eyx0, SizeY - 1, SizeZ, double);
34   ALLOC_2D(ezx0, SizeY, SizeZ - 1, double);
35   ALLOC_2D(eyx1, SizeY - 1, SizeZ, double);
36   ALLOC_2D(ezx1, SizeY, SizeZ - 1, double);
37
38   ALLOC_2D(exy0, SizeX - 1, SizeZ, double);
39   ALLOC_2D(ezy0, SizeX, SizeZ - 1, double);
40   ALLOC_2D(exy1, SizeX - 1, SizeZ, double);
41   ALLOC_2D(ezy1, SizeX, SizeZ - 1, double);
42
43   ALLOC_2D(exz0, SizeX - 1, SizeY, double);
44   ALLOC_2D(eyz0, SizeX, SizeY - 1, double);
45   ALLOC_2D(exz1, SizeX - 1, SizeY, double);
46   ALLOC_2D(eyz1, SizeX, SizeY - 1, double);
47
48   return;
49 }  /* end abcInit() */
50
51 /* function that applies ABC -- called once per time step */
52 void abc(Grid *g)
53 {
54   int mm, nn, pp;
55
56   if (abccoef == 0.0) {
```

```
57      fprintf(stderr,
58              "abc: abcInit must be called before abc.  Terminating...\n");
59      exit(-1);
60    }
61
62    /* ABC at "x0" */
63    mm = 0;
64    for (nn = 0; nn < SizeY - 1; nn++)
65      for (pp = 0; pp < SizeZ; pp++) {
66        Ey(mm, nn, pp) = Eyx0(nn, pp) +
67          abccoef * (Ey(mm + 1, nn, pp) - Ey(mm, nn, pp));
68        Eyx0(nn, pp) = Ey(mm + 1, nn, pp);
69      }
70    for (nn = 0; nn < SizeY; nn++)
71      for (pp = 0; pp < SizeZ - 1; pp++) {
72        Ez(mm, nn, pp) = Ezx0(nn, pp) +
73          abccoef * (Ez(mm + 1, nn, pp) - Ez(mm, nn, pp));
74        Ezx0(nn, pp) = Ez(mm + 1, nn, pp);
75      }
76
77    /* ABC at "x1" */
78    mm = SizeX - 1;
79    for (nn = 0; nn < SizeY - 1; nn++)
80      for (pp = 0; pp < SizeZ; pp++) {
81        Ey(mm, nn, pp) = Eyx1(nn, pp) +
82          abccoef * (Ey(mm - 1, nn, pp) - Ey(mm, nn, pp));
83        Eyx1(nn, pp) = Ey(mm - 1, nn, pp);
84      }
85    for (nn = 0; nn < SizeY; nn++)
86      for (pp = 0; pp < SizeZ - 1; pp++) {
87        Ez(mm, nn, pp) = Ezx1(nn, pp) +
88          abccoef * (Ez(mm - 1, nn, pp) - Ez(mm, nn, pp));
89        Ezx1(nn, pp) = Ez(mm - 1, nn, pp);
90      }
91
92    /* ABC at "y0" */
93    nn = 0;
94    for (mm = 0; mm < SizeX - 1; mm++)
95      for (pp = 0; pp < SizeZ; pp++) {
96        Ex(mm, nn, pp) = Exy0(mm, pp) +
97          abccoef * (Ex(mm, nn + 1, pp) - Ex(mm, nn, pp));
98        Exy0(mm, pp) = Ex(mm, nn + 1, pp);
99      }
100   for (mm = 0; mm < SizeX; mm++)
101     for (pp = 0; pp < SizeZ - 1; pp++) {
102       Ez(mm, nn, pp) = Ezy0(mm, pp) +
103         abccoef * (Ez(mm, nn + 1, pp) - Ez(mm, nn, pp));
```

```
104       Ezy0(mm, pp) = Ez(mm, nn + 1, pp);
105     }
106
107   /* ABC at "y1" */
108   nn = SizeY - 1;
109   for (mm = 0; mm < SizeX - 1; mm++)
110     for (pp = 0; pp < SizeZ; pp++) {
111       Ex(mm, nn, pp) = Exy1(mm, pp) +
112         abccoef * (Ex(mm, nn - 1, pp) - Ex(mm, nn, pp));
113       Exy1(mm, pp) = Ex(mm, nn - 1, pp);
114     }
115   for (mm = 0; mm < SizeX; mm++)
116     for (pp = 0; pp < SizeZ - 1; pp++) {
117       Ez(mm, nn, pp) = Ezy1(mm, pp) +
118         abccoef * (Ez(mm, nn - 1, pp) - Ez(mm, nn, pp));
119       Ezy1(mm, pp) = Ez(mm, nn - 1, pp);
120     }
121
122   /* ABC at "z0" (bottom) */
123   pp = 0;
124   for (mm = 0; mm < SizeX - 1; mm++)
125     for (nn = 0; nn < SizeY; nn++) {
126       Ex(mm, nn, pp) = Exz0(mm, nn) +
127         abccoef * (Ex(mm, nn, pp + 1) - Ex(mm, nn, pp));
128       Exz0(mm, nn) = Ex(mm, nn, pp + 1);
129     }
130   for (mm = 0; mm < SizeX; mm++)
131     for (nn = 0; nn < SizeY - 1; nn++) {
132       Ey(mm, nn, pp) = Eyz0(mm, nn) +
133         abccoef * (Ey(mm, nn, pp + 1) - Ey(mm, nn, pp));
134       Eyz0(mm, nn) = Ey(mm, nn, pp + 1);
135     }
136
137   /* ABC at "z1" (top) */
138   pp = SizeZ - 1;
139   for (mm = 0; mm < SizeX - 1; mm++)
140     for (nn = 0; nn < SizeY; nn++) {
141       Ex(mm, nn, pp) = Exz1(mm, nn) +
142         abccoef * (Ex(mm, nn, pp - 1) - Ex(mm, nn, pp));
143       Exz1(mm, nn) = Ex(mm, nn, pp - 1);
144     }
145   for (mm = 0; mm < SizeX; mm++)
146     for (nn = 0; nn < SizeY - 1; nn++) {
147       Ey(mm, nn, pp) = Eyz1(mm, nn) +
148         abccoef * (Ey(mm, nn, pp - 1) - Ey(mm, nn, pp));
149       Eyz1(mm, nn) = Ey(mm, nn, pp - 1);
150     }
```

```
151
152   return;
153 }  /* end abc() */
```

The function to initialize the 3D grid is shown in Program 9.7. Here the grid is simply homogeneous free space. The function starts by setting various parameters of the grid, such as the size and the number of time steps, in lines 9–14. The function then allocates space for the various arrays (lines 17–35). Finally, the function initializes the values of the coefficient arrays to correspond to free space (lines 38–79).

Program 9.7 `grid3dhomo.c` Function to initialize a homogeneous 3D grid.

```
1  #include "fdtd-macro.h"
2  #include "fdtd-alloc.h"
3  #include <math.h>
4
5  void gridInit(Grid *g) {
6    double imp0 = 377.0;
7    int mm, nn, pp;
8
9    Type = threeDGrid;
10   SizeX = 32; // size of domain
11   SizeY = 31;
12   SizeZ = 31;
13   MaxTime = 300; // duration of simulation
14   Cdtds = 1.0 / sqrt(3.0); // Courant number
15
16   /* memory allocation */
17   ALLOC_3D(g->hx,   SizeX, SizeY - 1, SizeZ - 1, double);
18   ALLOC_3D(g->chxh, SizeX, SizeY - 1, SizeZ - 1, double);
19   ALLOC_3D(g->chxe, SizeX, SizeY - 1, SizeZ - 1, double);
20   ALLOC_3D(g->hy,   SizeX - 1, SizeY, SizeZ - 1, double);
21   ALLOC_3D(g->chyh, SizeX - 1, SizeY, SizeZ - 1, double);
22   ALLOC_3D(g->chye, SizeX - 1, SizeY, SizeZ - 1, double);
23   ALLOC_3D(g->hz,   SizeX - 1, SizeY - 1, SizeZ, double);
24   ALLOC_3D(g->chzh, SizeX - 1, SizeY - 1, SizeZ, double);
25   ALLOC_3D(g->chze, SizeX - 1, SizeY - 1, SizeZ, double);
26
27   ALLOC_3D(g->ex,   SizeX - 1, SizeY, SizeZ, double);
28   ALLOC_3D(g->cexe, SizeX - 1, SizeY, SizeZ, double);
29   ALLOC_3D(g->cexh, SizeX - 1, SizeY, SizeZ, double);
30   ALLOC_3D(g->ey,   SizeX, SizeY - 1, SizeZ, double);
31   ALLOC_3D(g->ceye, SizeX, SizeY - 1, SizeZ, double);
32   ALLOC_3D(g->ceyh, SizeX, SizeY - 1, SizeZ, double);
33   ALLOC_3D(g->ez,   SizeX, SizeY, SizeZ - 1, double);
```

```
34   ALLOC_3D(g->ceze, SizeX, SizeY, SizeZ - 1, double);
35   ALLOC_3D(g->cezh, SizeX, SizeY, SizeZ - 1, double);
36
37   /* set electric-field update coefficients */
38   for (mm = 0; mm < SizeX - 1; mm++)
39     for (nn = 0; nn < SizeY; nn++)
40       for (pp = 0; pp < SizeZ; pp++) {
41         Cexe(mm, nn, pp) = 1.0;
42         Cexh(mm, nn, pp) = Cdtds * imp0;
43       }
44
45   for (mm = 0; mm < SizeX; mm++)
46     for (nn = 0; nn < SizeY - 1; nn++)
47       for (pp = 0; pp < SizeZ; pp++) {
48         Ceye(mm, nn, pp) = 1.0;
49         Ceyh(mm, nn, pp) = Cdtds * imp0;
50       }
51
52   for (mm = 0; mm < SizeX; mm++)
53     for (nn = 0; nn < SizeY; nn++)
54       for (pp = 0; pp < SizeZ - 1; pp++) {
55         Ceze(mm, nn, pp) = 1.0;
56         Cezh(mm, nn, pp) = Cdtds * imp0;
57       }
58
59   /* set magnetic-field update coefficients */
60   for (mm = 0; mm < SizeX; mm++)
61     for (nn = 0; nn < SizeY - 1; nn++)
62       for (pp = 0; pp < SizeZ - 1; pp++) {
63         Chxh(mm, nn, pp) = 1.0;
64         Chxe(mm, nn, pp) = Cdtds / imp0;
65       }
66
67   for (mm = 0; mm < SizeX - 1; mm++)
68     for (nn = 0; nn < SizeY; nn++)
69       for (pp = 0; pp < SizeZ - 1; pp++) {
70         Chyh(mm, nn, pp) = 1.0;
71         Chye(mm, nn, pp) = Cdtds / imp0;
72       }
73
74   for (mm = 0; mm < SizeX - 1; mm++)
75     for (nn = 0; nn < SizeY - 1; nn++)
76       for (pp = 0; pp < SizeZ; pp++) {
77         Chzh(mm, nn, pp) = 1.0;
78         Chze(mm, nn, pp) = Cdtds / imp0;
79       }
80
```

```
81   return;
82 }  /* end gridInit() */
```

As mentioned, there are many ways one might display 3D vector data. Here we merely record the E_x field over a constant-x and a constant-y plane. In this way, the core of the snapshot code is quite similar to that which was used in the 2D simulations. The snapshot code to accomplish this is shown in Program 9.8.

Program 9.8 `snapshot3d.c` Functions used to record 2D snapshots of the E_x field. At the appropriate time steps, two snapshots are taken: one over a constant-x plane and another over a constant-y plane. These snapshots are written to separate files.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "fdtd-macro.h"
4
5  static int temporalStride = -2, frameX = 0, frameY = 0, startTime;
6  static char basename[80];
7
8  void snapshot3dInit(Grid *g) {
9
10   int choice;
11
12   printf("Do you want 2D snapshots of the 3D grid? (1=yes, 0=no) ");
13   scanf("%d", &choice);
14   if (choice == 0) {
15     temporalStride = -1;
16     return;
17   }
18
19   printf("Duration of simulation is %d steps.\n", MaxTime);
20   printf("Enter start time and temporal stride: ");
21   scanf(" %d %d", &startTime, &temporalStride);
22   printf("Enter the base name: ");
23   scanf(" %s", basename);
24
25   return;
26 }  /* end snapshot3dInit() */
27
28
29 void snapshot3d(Grid *g) {
30   int mm, nn, pp;
31   float dim1, dim2, temp;
32   char filename[100];
33   FILE *out;
```

```
34
35    /* ensure temporal stride set to a reasonable value */
36    if (temporalStride == -1) {
37      return;
38    } if (temporalStride < -1) {
39      fprintf(stderr,
40        "snapshot2d: snapshotInit2d must be called before snapshot.\n"
41        "            Temporal stride must be set to positive value.\n");
42      exit(-1);
43    }
44
45    /* get snapshot if temporal conditions met */
46    if (Time >= startTime &&
47        (Time - startTime) % temporalStride == 0) {
48
49      /************ write the constant-x slice ************/
50      sprintf(filename, "%s-x.%d", basename, frameX++);
51      out = fopen(filename, "wb");
52
53      /* write dimensions to output file */
54      dim1 = SizeY; // express dimensions as floats
55      dim2 = SizeZ; // express dimensions as floats
56      fwrite(&dim1, sizeof(float), 1, out);
57      fwrite(&dim2, sizeof(float), 1, out);
58
59      /* write remaining data */
60      mm = (SizeX - 1) / 2;
61      for (pp = SizeZ - 1; pp >= 0; pp--)
62        for (nn = 0; nn < SizeY; nn++) {
63          temp = (float)Ex(mm, nn, pp); // store data as a float
64          fwrite(&temp, sizeof(float), 1, out); // write the float
65        }
66
67      fclose(out);  // close file
68
69      /************ write the constant-y slice ************/
70      sprintf(filename, "%s-y.%d", basename, frameY++);
71      out = fopen(filename, "wb");
72
73      /* write dimensions to output file */
74      dim1 = SizeX - 1; // express dimensions as floats
75      dim2 = SizeZ; // express dimensions as floats
76      fwrite(&dim1, sizeof(float), 1, out);
77      fwrite(&dim2, sizeof(float), 1, out);
78
79      /* write remaining data */
80      nn = SizeY / 2;
```

```
81      for (pp = SizeZ - 1; pp >= 0; pp--)
82        for (mm = 0; mm < SizeX - 1; mm++) {
83          temp = (float)Ex(mm, nn, pp); // store data as a float
84          fwrite(&temp, sizeof(float), 1, out); // write the float
85        }
86
87      fclose(out);  // close file
88    }
89
90    return;
91  }  /* end snapshot3d() */
```

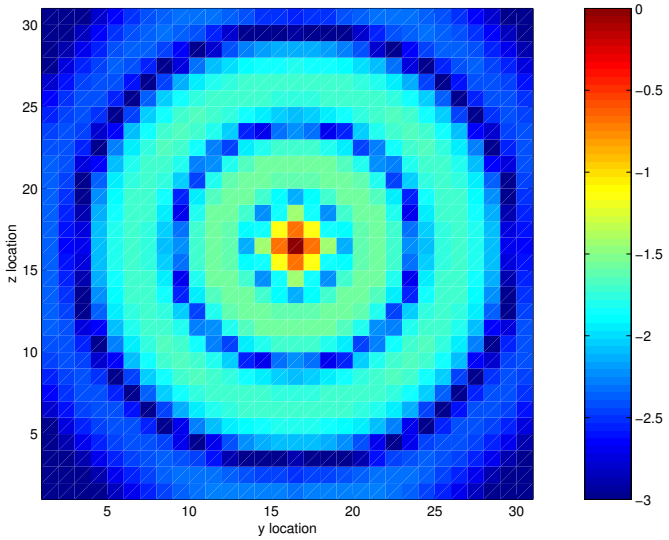
Figure 9.7 shows snapshots of E_x taken over two different planes at time step 40. The Ricker wavelet is such that there are 15 points per wavelength at the most energetic frequency. The field has been normalized by 0.3 and three decades of scaling are used. In Fig. 9.7(a), taken over a constant-x plane, the field is seen to radiate isotropically away from the dipole source—the dipole is normal to the plane. In Fig. 9.7(b), taken over a constant-y plane, the source is contained in the plane and oriented horizontally. Therefore the radiated field is stronger above and below the dipole than along the line of the dipole.

9.5 TFSF Boundary

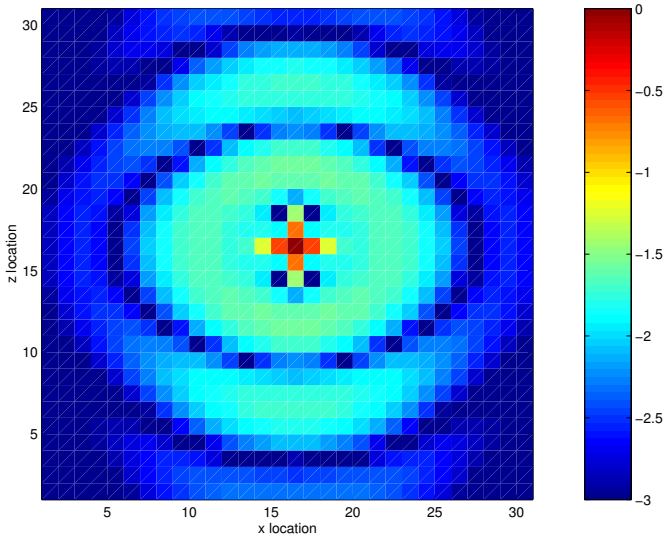
A total-field scattered-field boundary can be used in 3D grids to introduce the incident field. Although we have only considered using the TFSF boundary to introduce plane waves, it is worth noting that in principle any field could be introduced over the boundary. So, for example, if the incident field were due to a dipole source which was located physically outside of the grid, the field due to that source could be introduced over the TFSF boundary. But, whatever the type of incident field, one should be careful to ensure that the description of the incident field over the boundary matches the way the field actually behaves in the grid. Simply using the expression for the incident field in the continuous world will invariably cause some leakage across the boundary (the amount of leakage can always be reduced by using a finer discretization and may be acceptably small for various applications). Here we will restrict consideration to an incident plane wave and a one-dimensional auxiliary grid will be used to determine the incident field. Furthermore, we will assume the direction of wave propagation is along one of the grid axes.

Figure 9.8 shows a 3D computational domain in which a TFSF boundary exists. The TFSF boundary can be any shape, but we will restrict consideration to the cuboid shape shown in the figure. The boundary has six faces. Each face has two electric-field components and two magnetic-field components tangential to the boundary. These are the fields that must have their values corrected to account for the presence of the TFSF boundary since they will have at least one neighbor on the opposite side of the boundary.

To illustrate the dependence of nodes across the boundary, 2D slices are taken through a computational domain with nominal dimensions $9 \times 7 \times 8$. Figure 9.9 shows the orientation of two constant-x slices. These slices are separated by a half spatial-step in the x direction. The fields contained in these slices are shown in Fig. 9.10. The TFSF boundary is shown as a dashed line and



(a)



(b)

Figure 9.7: Color map of the E_x field at time-step 40. (a) Field over a constant-x plane that passes through the source node. (b) Field over a constant-y plane that passes through the source node. The field has been normalize by 0.3 and three decades of scaling are used. These images were generated using the Matlab commands in Appendix C. The image could be interpolated to smooth the obvious pixelization but that is not done here in order to emphasizes the inherent discrete nature of the simulation.

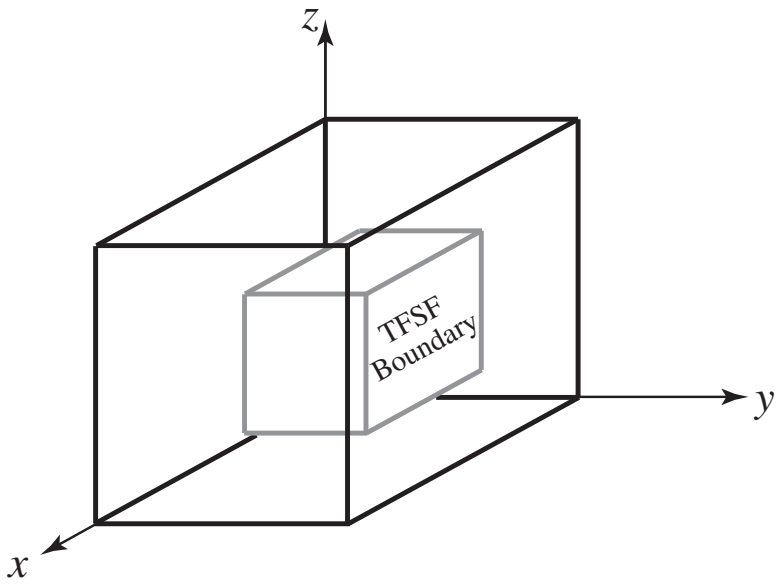


Figure 9.8: Three-dimensional computational domain which contains a TFSF boundary.

nodes that must be corrected owing to the existence of a neighbor on the other side of the boundary are enclosed in a "box" with rounded edges. Figure 9.10(a) shows the plane that contains E_y, E_z, and H_x while Fig. 9.10(b) contains E_x, H_y, and H_z. These two slices can be overlain to show the view seen looking along the x axis. The TFSF boundary would be specified by the first and last node associated with the boundary, i.e., two ordered triplets. The correspondence of these nodes to the boundary dimensions are indicated by the nodes enclosed by a dashed line and labeled "First" and "Last." In the example shown here, the indices of the first node would be $(2, 2, 2)$ and the indices of the last node would be $(6, 4, 5)$. (However, from just these two slices we cannot determine the extent of the TF region in the x direction. Instead, that will be shown in subsequent figures.)

Note that along the "bottom" and "top" of the TFSF boundary in Fig. 9.10(a) there are two pairs of nodes that must be corrected while in Fig. 9.10(b) there are three pairs of nodes that must be corrected. Similarly, there are different numbers of pairs along the two sides. The construction of the TFSF boundary used here is such that corrections are only applied to electric fields within the total-field region and are only applied to magnetic fields in the scattered-field region.

Figure 9.11 shows the orientation of two constant-y slices and Fig. 9.12 shows the fields over these two slices. The slices are separated by a half spatial-step in the y direction. As before, these slices can be overlain to see the view looking along the axis (in this case the y axis). From this view one can see that the first and last indices in the x direction are 2 and 6, respectively.

Figure 9.13 shows the orientation of two constant-z slices and Fig. 9.14 shows the fields over these two slices. The slices are separated by a half spatial-step in the z direction. These slices correspond to the TE^z and TM^z grids which were studied in Chap. 8.

Figures 9.10, 9.12, and 9.14 show all the possible dependencies across the TFSF boundary. However, in some applications not all these dependencies may be relevant. For example, consider the case when the incident electric field is polarized in the z direction and propagating in the x direction. In this case there are only two non-zero components of the incident field: E_z and H_y.

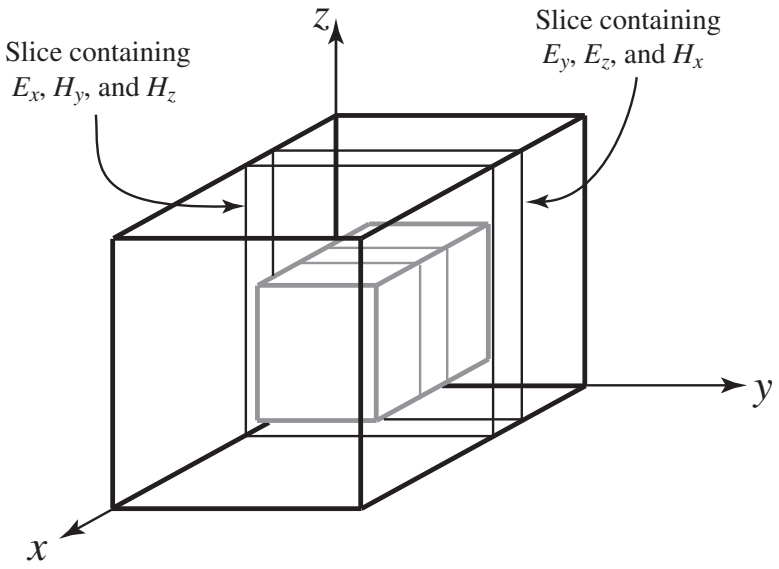


Figure 9.9: Constant-x slices of the computational domain used to illustration the relationship of nodes across the TFSF boundary. The slices are separated by a half spatial-step in the x direction.

Thus, not all of the fields tangential to the TFSF boundary would have to be corrected. Only those fields that depend on an E_z or H_y node on the other side of the boundary would need to be corrected.

Continuing with the assumption of an incident field whose only non-zero components are E_z and H_y, one sees from Fig. 9.10(a) that H_x nodes on the constant-y faces would have to be corrected since they depend on E_z nodes on the other side of the boundary (these are the nodes along the left and right side of the TFSF boundary in Fig. 9.10(a)). However, the corresponding E_z nodes would not have to be corrected because there is no incident H_x field. From Fig. 9.10(b) it is seen that E_x nodes on the constant-z faces would have to be be corrected since they depend on H_y on the other side of the boundary. But, the corresponding H_y nodes do not have to be corrected because there is no incident E_x field.

Figure 9.12(a) shows that E_x must be corrected over constant-z faces (the top and bottom of the TFSF boundary in the figure). (This agrees with the conclusion drawn from inspection of Fig. 9.10(a). There is redundant information in these figures.) Figure 9.12(a) also shows that both E_z and H_y must be corrected on constant-x faces. Inspection of Fig. 9.12(b) shows that there is no need to correct E_y, H_x, or H_z over constant-x or constant-z faces since there is no incident field at the nodes that neighbor these components.

Figure 9.14(a) indicates, for the assumed incident field, there is no need to correct E_x, E_y, or H_z over constant-x and constant-y faces. Figure 9.14(b) shows that H_x must be corrected over constant-y faces. This is the same conclusion one draws from inspection of Fig. 9.10(a). It also shows, as was seen in Fig. 9.12(a), that both E_z and H_y must be corrected over constant-x faces.

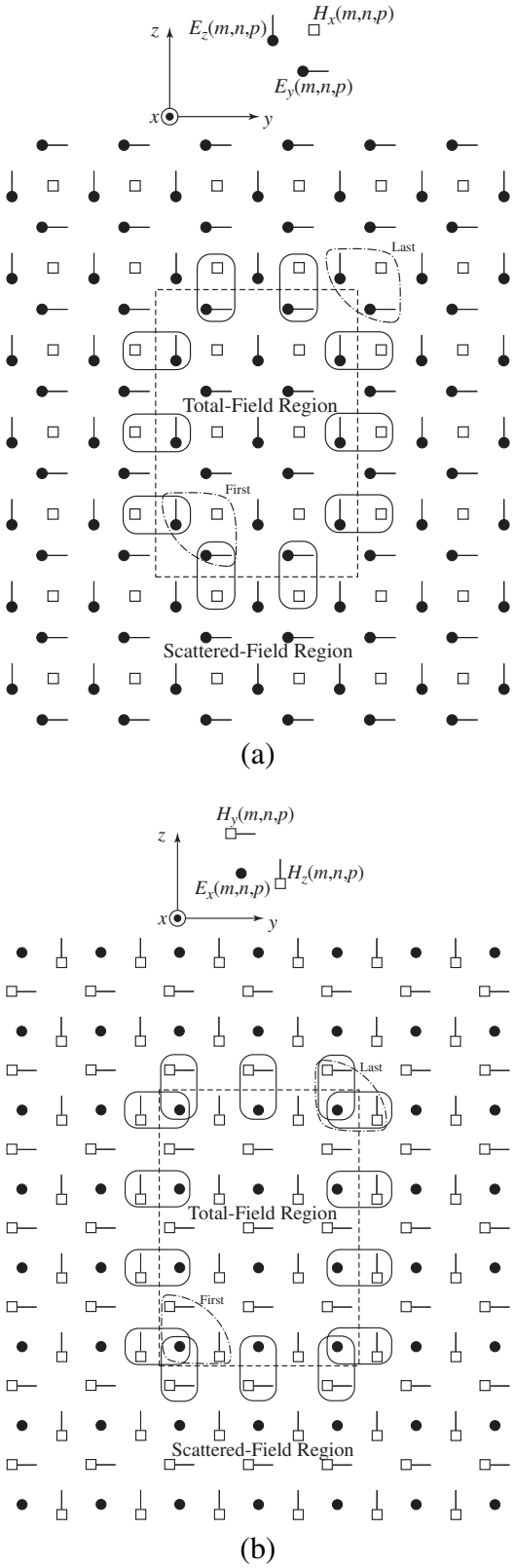


Figure 9.10: Fields over the constant-x slices depicted in Fig. 9.9. (a) Slice containing E_y, E_z, and H_x. (b) Slice containing E_x, H_y, and H_z.

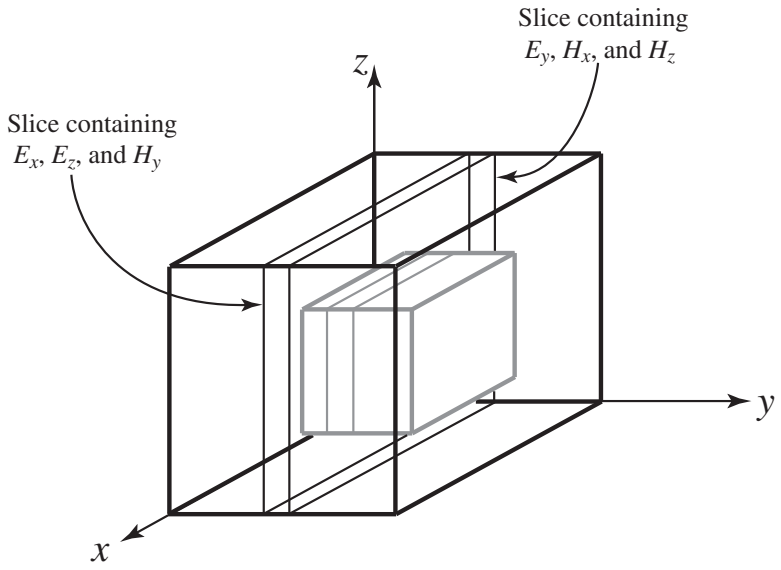


Figure 9.11: Constant-y slices of the computational domain used to illustration the relationship of nodes across the TFSF boundary. The slices are separated by a half spatial-step in the y direction.

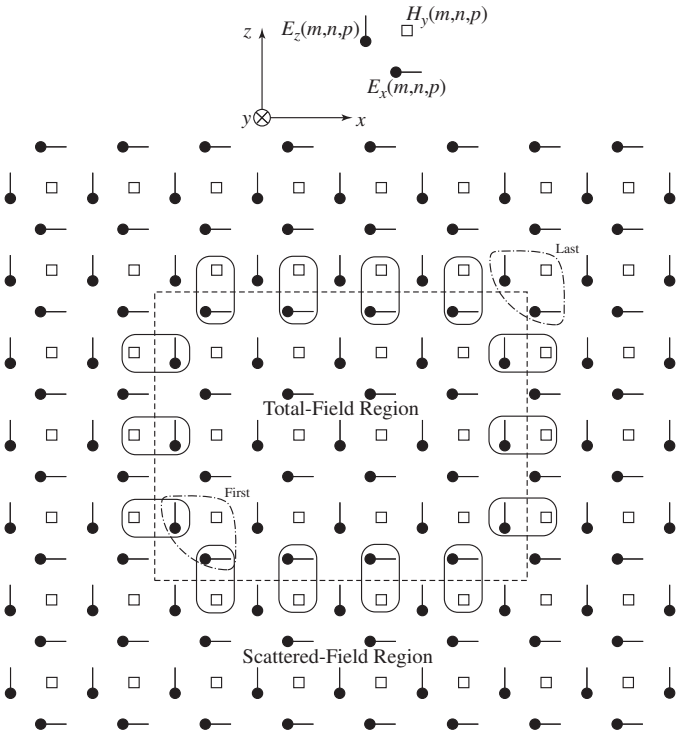
9.6 TFSF Demonstration

To demonstrate the implementation of a 3D TFSF boundary, we will model an incident field that is polarized in the z direction and propagating in the x direction. This corresponds to the scenario described in the previous section. Because of the given polarization and direction of propagation, many of the dependencies shown in Figs. 9.10–9.14 will not require any coding.

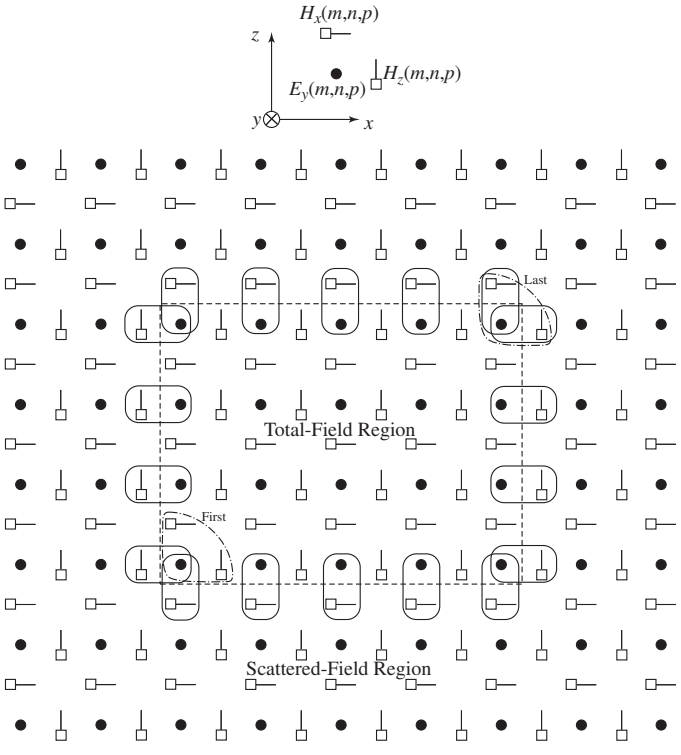
The grid will be $35 \times 35 \times 35$. A Courant number equal to the limit of $1/\sqrt{3}$ will be used. Two simulations will be performed. In one there will be no scatterer and in the other a spherical PEC scatterer will be present. In 3D, the simplest way to model a solid PEC is to test if the center of a given Yee cube is within the PEC. If it is, as will be shown below, the 12 electric-field nodes on the edges of the cube are set to zero.

Program 9.9 shows the main body of the program. This program is essentially the same as the 2D program that contained a TFSF boundary (ref. Program 8.12). As shown in line 14, the TFSF code is initialized by calling an initialization function outside of the time-stepping loop. The corrections to the TFSF boundary are applied by the function `tfsf()` which, as shown in line 21, is called once per time-step. To work properly, this function must be called after the magnetic-field update, but before the electric-field update. After the magnetic fields are updated, the function `tfsf()` applies the necessary correction to the fields tangential to the TFSF boundary. Immediately after returning from this function, the electric fields are not in a consistent state in that the correction has been applied to the electric fields in anticipation of the impending update. This is the same as the case for the 2D implementation of a TFSF boundary discussed in Sec. 8.6.

Program 9.9 `3d-tfsf-demo.c` Main body of a program to implement a TFSF boundary in 3D.



(a)



(b)

Figure 9.12: Fields over the constant-y slices depicted in Fig. 9.11. (a) Slice containing E_x, E_z, and H_y. (b) Slice containing E_y, H_x, and H_z.

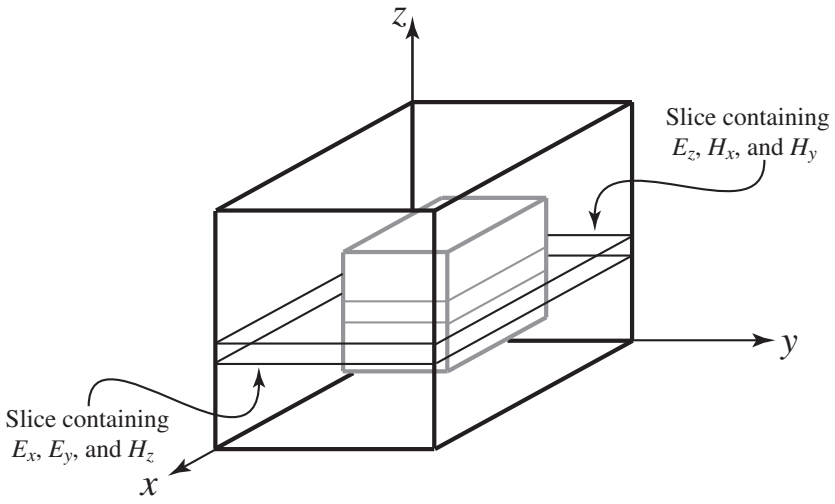


Figure 9.13: Constant-z slices of the computational domain used to illustration the relationship of nodes across the TFSF boundary. The slices are separated by a half spatial-step in the z direction.

```
/* 3D simulation with a TFSF boundary. */

#include "fdtd-alloc.h"
#include "fdtd-macro.h"
#include "fdtd-proto.h"

int main()
{
  Grid *g;

  ALLOC_1D(g, 1, Grid); // allocate memory for grid structure
  gridInit(g);          // initialize 3D grid

  tfsfInit(g);          // initialize TFSF boundary
  abcInit(g);           // initialize ABC
  snapshot3dInit(g);    // initialize snapshots

  /* do time stepping */
  for (Time = 0; Time < MaxTime; Time++) {
    updateH(g);      // update magnetic fields
    tfsf(g);         // apply correction to TFSF boundary
    updateE(g);      // update electric fields
    abc(g);          // apply ABC
    snapshot3d(g);   // take a snapshot (if appropriate)
  } // end of time-stepping

```

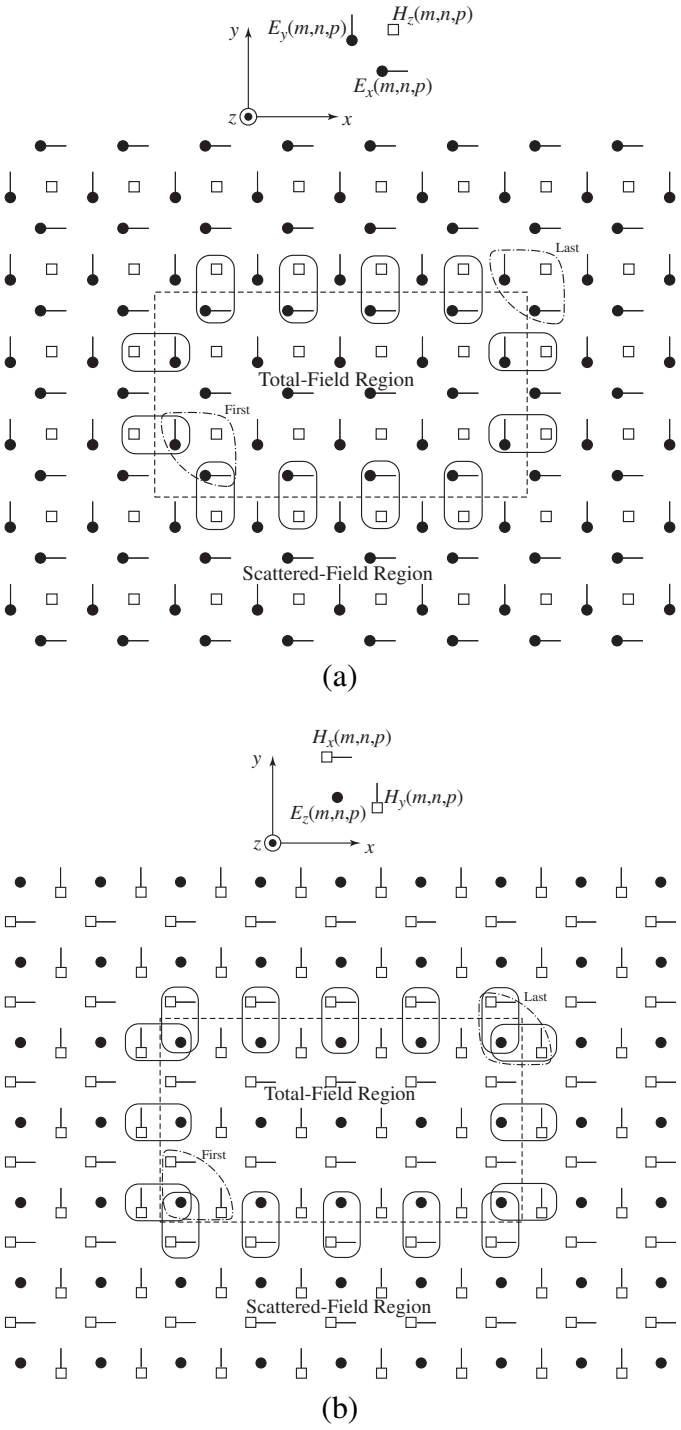


Figure 9.14: Fields over the constant-z slices depicted in Fig. 9.13. (a) Slice containing E_x, E_y, and H_z. (b) Slice containing E_z, H_x, and H_y.

```
27   return 0;
28 }
```

Program 9.10 provides the code for the TFSF functions. The function `tfsfInit()` take a single `Grid` argument, i.e., the 3D grid. There are seven static local variables in this program. Six of those specify the first and last points in the total-field region. The seventh is the `Grid` pointer `g1` that is used for the 1D auxiliary grid that represents the incident field. The initialization function `tfsfInit()` starts by allocating memory for `g1` and then copying the values from the 3D grid to the 1D grid. As discussed in connection with the 2D TFSF boundary, this is done to ensure the duration and Courant number are the same in both grids. Then, in line 21, the function `gridInit1D()` is called to complete the initialization of the 1D grid. This function is unchanged from that shown in Program 8.15. Starting in line 23, `tfsfInit()` prompts the user for indices for the first and last points in the total-field region. Finally, the source function initialization is called (line 29). In the results to be shown, the source function was a Ricker wavelet discretized such that there were 20 points per wavelength at the most energetic frequency.

Program 9.10 `tfsf-3d-ez.c` Three-dimensional TFSF implementation that assumes the electric field is polarized in the z direction and propagation is in the x direction.

```
1  /* TFSF boundary for a 3D grid.  A 1D auxiliary grid is used to
2   * calculate the incident field which is assumed to be propagating in
3   * the x direction and polarized in the z direction. */
4
5  #include <string.h>  // for memcpy
6  #include "fdtd-macro.h"
7  #include "fdtd-proto.h"
8  #include "fdtd-alloc.h"
9  #include "ezinc.h"
10
11 static int
12     firstX = 0, firstY, firstZ, // indices for first point in TF region
13     lastX, lastY, lastZ;        // indices for last point in TF region
14
15 static Grid *g1;  // 1D auxilliary grid
16
17 void tfsfInit(Grid *g) {
18
19   ALLOC_1D(g1, 1, Grid);       // allocate memory for 1D Grid
20   memcpy(g1, g, sizeof(Grid)); // copy information from 3D array
21   gridInit1d(g1);              // initialize 1d grid
22
23   printf("Grid is %d by %d by %d.\n", SizeX, SizeY, SizeZ);
24   printf("Enter indices for first point in TF region: ");
25   scanf(" %d %d %d", &firstX, &firstY, &firstZ);
```

```
26    printf("Enter indices for last point in TF region: ");
27    scanf(" %d %d %d", &lastX, &lastY, &lastZ);
28
29    ezIncInit(g); // initialize source function
30
31    return;
32  }  /* end tfsfInit() */
33
34
35  void tfsf(Grid *g) {
36    int mm, nn, pp;
37
38    // check if tfsfInit() has been called
39    if (firstX <= 0) {
40      fprintf(stderr,
41        "tfsf: tfsfInit must be called before tfsfUpdate.\n"
42        "      Boundary location must be set to positive value.\n");
43      exit(-1);
44    }
45
46    /****** constant x faces -- scattered-field nodes ******/
47
48    // correct Hy at firstX-1/2 by subtracting Ez_inc
49    mm = firstX;
50    for (nn = firstY; nn <= lastY; nn++)
51      for (pp = firstZ; pp < lastZ; pp++)
52        Hy(mm - 1, nn, pp) -= Chye(mm, nn, pp) * Ez1G(g1, mm);
53
54    // correct Hy at lastX + 1/2 by adding Ez_inc
55    mm = lastX;
56    for (nn = firstY; nn <= lastY; nn++)
57      for (pp = firstZ; pp < lastZ; pp++)
58        Hy(mm, nn, pp) += Chye(mm, nn, pp) * Ez1G(g1, mm);
59
60    /**** constant y faces -- scattered-field nodes ****/
61
62    // correct Hx at firstY-1/2 by adding Ez_inc
63    nn = firstY;
64    for (mm = firstX; mm <= lastX; mm++)
65      for (pp = firstZ; pp < lastZ; pp++)
66        Hx(mm, nn - 1, pp) += Chxe(mm, nn - 1, pp) * Ez1G(g1, mm);
67
68    // correct Hx at lastY+1/2 by subtracting Ez_inc
69    nn = lastY;
70    for (mm = firstX; mm <= lastX; mm++)
71      for (pp = firstZ; pp < lastZ; pp++)
72        Hx(mm, nn, pp) -= Chxe(mm, nn, pp) * Ez1G(g1, mm);
```

```
73
74    /**** constant z faces -- scattered-field nodes ****/
75
76    // nothing to correct on this face
77
78    /**** update the fields in the auxiliary 1D grid ****/
79    updateH(g1);     // update 1D magnetic field
80    updateE(g1);     // update 1D electric field
81    Ez1G(g1, 0) = ezInc(TimeG(g1), 0.0); // set source node
82    TimeG(g1)++;     // increment time in 1D grid
83
84    /**** constant x faces -- total-field nodes ****/
85
86    // correct Ez at firstX face by subtracting Hy_inc
87    mm = firstX;
88    for (nn = firstY; nn <= lastY; nn++)
89      for (pp = firstZ; pp < lastZ; pp++)
90        Ez(mm, nn, pp) -= Cezh(mm, nn, pp) * Hy1G(g1, mm - 1);
91
92    // correct Ez at lastX face by adding Hy_inc
93    mm = lastX;
94    for (nn = firstY; nn <= lastY; nn++)
95      for (pp = firstZ; pp < lastZ; pp++)
96        Ez(mm, nn, pp) += Cezh(mm, nn, pp) * Hy1G(g1, mm);
97
98    /**** constant y faces -- total-field nodes ****/
99
100   // nothing to correct on this face
101
102   /**** constant z faces -- total-field nodes ****/
103
104   // correct Ex at firstZ face by adding Hy_inc
105   pp = firstZ;
106   for (mm = firstX; mm < lastX; mm++)
107     for (nn = firstY; nn <= lastY; nn++)
108       Ex(mm, nn, pp) += Cexh(mm, nn, pp) * Hy1G(g1, mm);
109
110   // correct Ex at lastZ face by subtracting Hy_inc
111   pp = lastZ;
112   for (mm = firstX; mm < lastX; mm++)
113     for (nn = firstY; nn <= lastY; nn++)
114       Ex(mm, nn, pp) -= Cexh(mm, nn, pp) * Hy1G(g1, mm);
115
116   return;
117 }  /* end tfsf() */
```

The function `tfsf()` begins on line 35. This function, which is called once per time-step,

applies the corrections to the various faces of the TFSF boundary as described in the previous section.

The code to implement the 3D grid is shown in Program 9.11. This code is almost identical to the code for the homogeneous grid that was given in Program 9.7. The the only significant difference is the possible inclusion of the PEC sphere. The variables associated with the sphere are listed in lines 10 and 11. The users is queried if the sphere is present. If it is, the variable `isSpherePresent` is set to one. Otherwise it is set to zero. The radius of the sphere, which is stored in `radius`, is set to 8 cells while the indices for the center of the sphere are set to $(17, 17, 17)$. The grid is initially set to uniform free space. However, if the sphere is present, as shown starting at line 67, the center of each Yee cube is checked. If it is within a distance of `radius` cells from the center of the sphere, all 12 electric-field nodes on the edges of the cube are set to zero. (In the for-loops associated with this check, the squared values of the distances are used so as to avoid having to calculate square roots.) The update coefficients for the magnetic field are unaffected by the presence of the PEC.

Program 9.11 `grid3dsphere.c` Function to initialize a `Grid` structure. The user is prompted to determine if a PEC sphere of radius 8-cells should be present. If it is not, the grid is homogeneous free space.

```
1  #include "fdtd-macro.h"
2  #include "fdtd-alloc.h"
3  #include <math.h>
4
5  void gridInit(Grid *g) {
6    double imp0 = 377.0;
7    int mm, nn, pp;
8
9    // sphere parameters
10   int m_c = 17, n_c = 17, p_c = 17, isSpherePresent;
11   double m2, n2, p2, r2, radius = 8.0;
12
13   Type    = threeDGrid;
14   SizeX   = 35; // size of domain
15   SizeY   = 35;
16   SizeZ   = 35;
17   MaxTime = 300; // duration of simulation
18   Cdtds   = 1.0 / sqrt(3.0); // Courant number
19
20   printf("If the sphere present: (1=yes, 0=no) ");
21   scanf(" %d", &isSpherePresent);
22
23   /* memory allocation */
24   ALLOC_3D(g->hx,   SizeX, SizeY - 1, SizeZ - 1, double);
25   ALLOC_3D(g->chxh, SizeX, SizeY - 1, SizeZ - 1, double);
26   ALLOC_3D(g->chxe, SizeX, SizeY - 1, SizeZ - 1, double);
27   ALLOC_3D(g->hy,   SizeX - 1, SizeY, SizeZ - 1, double);
```

```
28  ALLOC_3D(g->chyh, SizeX - 1, SizeY, SizeZ - 1, double);
29  ALLOC_3D(g->chye, SizeX - 1, SizeY, SizeZ - 1, double);
30  ALLOC_3D(g->hz,   SizeX - 1, SizeY - 1, SizeZ, double);
31  ALLOC_3D(g->chzh, SizeX - 1, SizeY - 1, SizeZ, double);
32  ALLOC_3D(g->chze, SizeX - 1, SizeY - 1, SizeZ, double);
33
34  ALLOC_3D(g->ex,   SizeX - 1, SizeY, SizeZ, double);
35  ALLOC_3D(g->cexe, SizeX - 1, SizeY, SizeZ, double);
36  ALLOC_3D(g->cexh, SizeX - 1, SizeY, SizeZ, double);
37  ALLOC_3D(g->ey,   SizeX, SizeY - 1, SizeZ, double);
38  ALLOC_3D(g->ceye, SizeX, SizeY - 1, SizeZ, double);
39  ALLOC_3D(g->ceyh, SizeX, SizeY - 1, SizeZ, double);
40  ALLOC_3D(g->ez,   SizeX, SizeY, SizeZ - 1, double);
41  ALLOC_3D(g->ceze, SizeX, SizeY, SizeZ - 1, double);
42  ALLOC_3D(g->cezh, SizeX, SizeY, SizeZ - 1, double);
43
44  /* set electric-field update coefficients */
45  for (mm = 0; mm < SizeX - 1; mm++)
46    for (nn = 0; nn < SizeY; nn++)
47      for (pp = 0; pp < SizeZ; pp++) {
48        Cexe(mm, nn, pp) = 1.0;
49        Cexh(mm, nn, pp) = Cdtds * imp0;
50      }
51
52  for (mm = 0; mm < SizeX; mm++)
53    for (nn = 0; nn < SizeY - 1; nn++)
54      for (pp = 0; pp < SizeZ; pp++) {
55        Ceye(mm, nn, pp) = 1.0;
56        Ceyh(mm, nn, pp) = Cdtds * imp0;
57      }
58
59  for (mm = 0; mm < SizeX; mm++)
60    for (nn = 0; nn < SizeY; nn++)
61      for (pp = 0; pp < SizeZ - 1; pp++) {
62        Ceze(mm, nn, pp) = 1.0;
63        Cezh(mm, nn, pp) = Cdtds * imp0;
64      }
65
66  // zero the nodes associated with the PEC sphere
67  if (isSpherePresent) {
68    r2 = radius * radius;
69    for (mm = 2; mm < SizeX - 2; mm++) {
70      m2 = (mm + 0.5 - m_c) * (mm + 0.5 - m_c);
71      for (nn = 2; nn < SizeY - 2; nn++) {
72        n2 = (nn + 0.5 - n_c) * (nn + 0.5 - n_c);
73        for (pp = 2; pp < SizeZ - 2; pp++) {
74          p2 = (pp + 0.5 - p_c) * (pp + 0.5 - p_c);
```

```
75            // if distance to center of a cube is less than radius
76            // of the sphere, zero all the surrounding electric
77            // field nodes
78            if (m2 + n2 + p2 < r2) {
79              // zero surrounding Ex nodes
80              Cexe(mm, nn,  pp) = 0.0;
81              Cexe(mm, nn + 1, pp) = 0.0;
82              Cexe(mm, nn,  pp + 1) = 0.0;
83              Cexe(mm, nn + 1, pp + 1) = 0.0;
84              Cexh(mm, nn,  pp) = 0.0;
85              Cexh(mm, nn + 1, pp) = 0.0;
86              Cexh(mm, nn,  pp + 1) = 0.0;
87              Cexh(mm, nn + 1, pp + 1) = 0.0;
88              // zero surrounding Ey nodes
89              Ceye(mm, nn, pp) = 0.0;
90              Ceye(mm + 1, nn, pp) = 0.0;
91              Ceye(mm, nn, pp + 1) = 0.0;
92              Ceye(mm + 1, nn, pp + 1) = 0.0;
93              Ceyh(mm, nn, pp) = 0.0;
94              Ceyh(mm + 1, nn, pp) = 0.0;
95              Ceyh(mm, nn, pp + 1) = 0.0;
96              Ceyh(mm + 1, nn, pp + 1) = 0.0;
97              // zero surrounding Ez nodes
98              Ceze(mm, nn, pp) = 0.0;
99              Ceze(mm + 1, nn, pp) = 0.0;
100             Ceze(mm, nn + 1, pp) = 0.0;
101             Ceze(mm + 1, nn + 1, pp) = 0.0;
102             Cezh(mm, nn,  pp) = 0.0;
103             Cezh(mm + 1, nn,  pp) = 0.0;
104             Cezh(mm, nn + 1, pp) = 0.0;
105             Cezh(mm + 1, nn + 1, pp) = 0.0;
106           }
107         }
108       }
109     }
110   }
111
112
113   /* set magnetic-field update coefficients */
114   for (mm = 0; mm < SizeX; mm++)
115     for (nn = 0; nn < SizeY - 1; nn++)
116       for (pp = 0; pp < SizeZ - 1; pp++) {
117         Chxh(mm, nn, pp) = 1.0;
118         Chxe(mm, nn, pp) = Cdtds / imp0;
119       }
120
121   for (mm = 0; mm < SizeX - 1; mm++)
```

```
122     for (nn = 0; nn < SizeY; nn++)
123       for (pp = 0; pp < SizeZ - 1; pp++) {
124         Chyh(mm, nn, pp) = 1.0;
125         Chye(mm, nn, pp) = Cdtds / imp0;
126       }
127
128   for (mm = 0; mm < SizeX - 1; mm++)
129     for (nn = 0; nn < SizeY - 1; nn++)
130       for (pp = 0; pp < SizeZ; pp++) {
131         Chzh(mm, nn, pp) = 1.0;
132         Chze(mm, nn, pp) = Cdtds / imp0;
133       }
134
135   return;
136 }  /* end gridInit() */
```

Figure 9.15 show the E_z field take over a constant y slice along the center of the computational domain. The figures on the left show the field when there is no scatterer while the figures on the right show the field at the same time-step but when the spherical scatterer is present. In the absence of a scatterer, one can see that there are no fields in the scattered-field region. The first point in the total-field region has indices of $(5, 5, 5)$ while the last point had indices of $(30, 30, 30)$.

In these simulations a first-order ABC is used and the code is unchanged from that presented in Program 9.6. The snapshot code used to generate the data for Fig. 9.15 is slightly different from Program 9.8 in that here the E_z field is being recorded while in Program 9.8 the E_x field was being recorded. However, this represents a minor change and hence the modified snapshot code is not shown. Another minor change that is not explicitly shown is that it would be necessary for the header file `fdtd-proto.h` to include the prototypes for the TFSF functions `tfsfInit()` and `tfsf()`. As with nearly all the other functions, these prototypes would merely show that these functions have a pointer to a `Grid` structure as their single argument.

9.7 Unequal Spatial Steps

In the previous discussion we have always assumed that $\Delta_x = \Delta_y = \Delta_z = \delta$. But how do things change if $\Delta_x \neq \Delta_y \neq \Delta_z$? We will consider that question in this section. First, let us introduce the following notation

$$\begin{aligned} \delta = \Delta_x \quad &\Rightarrow \quad \Delta_x = \delta, \\ r_y = \frac{\Delta_y}{\Delta_x} \quad &\Rightarrow \quad \Delta_y = r_y \delta, \\ r_z = \frac{\Delta_z}{\Delta_x} \quad &\Rightarrow \quad \Delta_z = r_z \delta. \end{aligned}$$

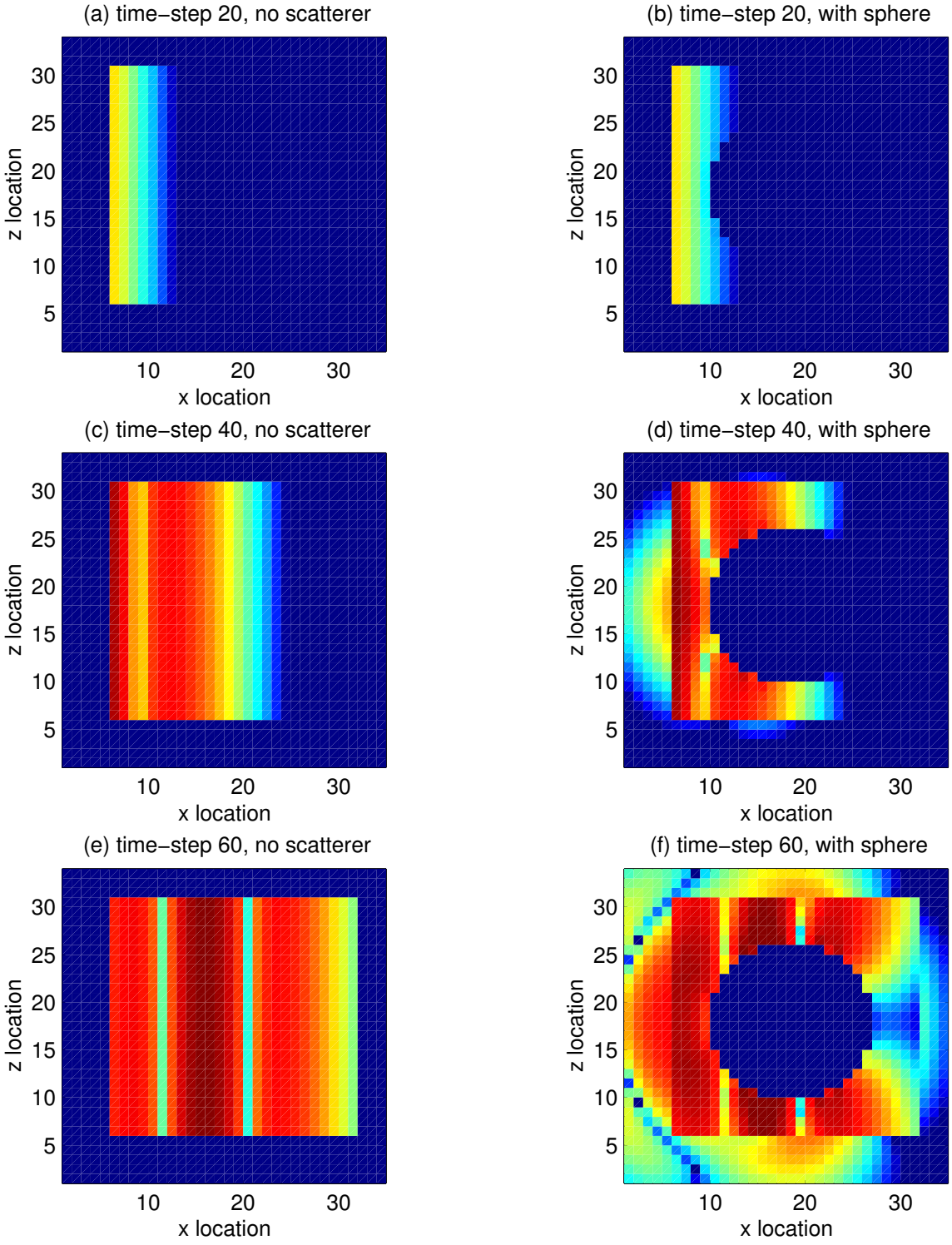


Figure 9.15: The E_z field over a constant-y plane taken from a 3D simulation in which the incident electric field is z-polarized and propagation is in the x direction. In the figures on the left no scatterer is present. Figures (a), (c), and (e) are taken at time-steps, 20, 40, and 60, respectively. The figures on the right show the field at the same time-steps, but a spherical scatterer is present.

Furthermore, we will defined scaled field quantities such that

$$\begin{aligned} e_x &= \Delta_x E_x, & h_x &= \Delta_x H_x, \\ e_y &= \Delta_y E_y, & h_y &= \Delta_y H_y, \\ e_z &= \Delta_z E_z, & h_z &= \Delta_z H_z. \end{aligned}$$

With these definitions in place, let us consider a rewritten form of (9.18)

$$\begin{aligned} \frac{1}{\Delta_x}\Delta_x E_x^{q+1}\left[m+\frac{1}{2},n,p\right] &= \frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}}\frac{1}{\Delta_x}\Delta_x E_x^{q}\left[m+\frac{1}{2},n,p\right] \\ &+ \frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\left(\frac{\Delta_t}{\epsilon\Delta_y\Delta_z}\left\{\Delta_z H_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n+\frac{1}{2},p\right]-\Delta_z H_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n-\frac{1}{2},p\right]\right\}\right. \\ &\left.- \frac{\Delta_t}{\epsilon\Delta_y\Delta_z}\left\{\Delta_y H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n,p+\frac{1}{2}\right]-\Delta_y H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n,p-\frac{1}{2}\right]\right\}\right). \quad (9.39) \end{aligned}$$

Multiplying through by Δ_x and employing the definitions given above, this update equation becomes

$$\begin{aligned} e_x^{q+1}\left[m+\frac{1}{2},n,p\right] &= \frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}}e_x^{q}\left[m+\frac{1}{2},n,p\right] \\ &+ \frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\frac{\Delta_t}{\epsilon\delta}\frac{1}{r_y r_z}\left(\left\{h_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n+\frac{1}{2},p\right]-h_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n-\frac{1}{2},p\right]\right\}\right. \\ &\left.- \left\{h_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n,p+\frac{1}{2}\right]-h_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n,p-\frac{1}{2}\right]\right\}\right). \quad (9.40) \end{aligned}$$

Importantly, all the (scaled) magnetic fields are multiplied by the same coefficient.

From inspection of (9.18), it might have appeared that when the spatial step sizes are not equal, another set of coefficients would have to be introduced (i.e., one for the term involving $1/\Delta_y$ and one for the term involving $1/\Delta_z$). This is true if the field components are not scaled by their respective lengths. However, by scaling the fields, it is still only necessary to have two coefficients per update equation.

The complete set up update equations for the scaled fields are:

$$\begin{aligned} h_x^{q+\frac{1}{2}}\left[m,n+\frac{1}{2},p+\frac{1}{2}\right] &= \frac{1-\frac{\sigma_m\Delta_t}{2\mu}}{1+\frac{\sigma_m\Delta_t}{2\mu}}h_x^{q-\frac{1}{2}}\left[m,n+\frac{1}{2},p+\frac{1}{2}\right] \\ &+ \frac{1}{1+\frac{\sigma_m\Delta_t}{2\mu}}\frac{\Delta_t}{\mu\delta}\frac{1}{r_y r_z}\left(\left\{e_y^{q}\left[m,n+\frac{1}{2},p+1\right]-e_y^{q}\left[m,n+\frac{1}{2},p\right]\right\}\right. \\ &\left.- \left\{e_z^{q}\left[m,n+1,p+\frac{1}{2}\right]-e_z^{q}\left[m,n,p+\frac{1}{2}\right]\right\}\right), \quad (9.41) \end{aligned}$$

$$h_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n,p+\frac{1}{2}\right]=\frac{1-\frac{\sigma_m\Delta_t}{2\mu}}{1+\frac{\sigma_m\Delta_t}{2\mu}}h_y^{q-\frac{1}{2}}\left[m+\frac{1}{2},n,p+\frac{1}{2}\right]$$
$$+\frac{1}{1+\frac{\sigma_m\Delta_t}{2\mu}}\frac{\Delta_t}{\mu\delta}\frac{r_y}{r_z}\left(\left\{e_z^q\left[m+1,n,p+\frac{1}{2}\right]-e_z^q\left[m,n,p+\frac{1}{2}\right]\right\}\right.$$
$$\left.-\left\{e_x^q\left[m+\frac{1}{2},n,p+1\right]-e_x^q\left[m+\frac{1}{2},n,p\right]\right\}\right),\quad(9.42)$$

$$h_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n+\frac{1}{2},p\right]=\frac{1-\frac{\sigma_m\Delta_t}{2\mu}}{1+\frac{\sigma_m\Delta_t}{2\mu}}h_z^{q-\frac{1}{2}}\left[m+\frac{1}{2},n+\frac{1}{2},p\right]$$
$$+\frac{1}{1+\frac{\sigma_m\Delta_t}{2\epsilon}}\frac{\Delta_t}{\mu\delta}\frac{r_z}{r_y}\left(\left\{e_x^q\left[m+\frac{1}{2},n+1,p\right]-e_x^q\left[m+\frac{1}{2},n,p\right]\right\}\right.$$
$$\left.-\left\{e_y^q\left[m+1,n+\frac{1}{2},p\right]-e_y^q\left[m,n+\frac{1}{2},p\right]\right\}\right).\quad(9.43)$$

$$e_x^{q+1}\left[m+\frac{1}{2},n,p\right]=\frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}}e_x^q\left[m+\frac{1}{2},n,p\right]$$
$$+\frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\frac{\Delta_t}{\epsilon\delta}\frac{1}{r_yr_z}\left(\left\{h_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n+\frac{1}{2},p\right]-h_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n-\frac{1}{2},p\right]\right\}\right.$$
$$\left.-\left\{h_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n,p+\frac{1}{2}\right]-h_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n,p-\frac{1}{2}\right]\right\}\right).\quad(9.44)$$

$$e_y^{q+1}\left[m,n+\frac{1}{2},p\right]=\frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}}e_y^q\left[m,n+\frac{1}{2},p\right]$$
$$+\frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\frac{\Delta_t}{\epsilon\delta}\frac{r_y}{r_z}\left(\left\{h_x^{q+\frac{1}{2}}\left[m,n+\frac{1}{2},p-\frac{1}{2}\right]-h_x^{q+\frac{1}{2}}\left[m,n+\frac{1}{2},p-\frac{1}{2}\right]\right\}\right.$$
$$\left.-\left\{h_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n+\frac{1}{2},p\right]-h_z^{q+\frac{1}{2}}\left[m-\frac{1}{2},n+\frac{1}{2},p\right]\right\}\right),\quad(9.45)$$

$$e_z^{q+1}\left[m,n,p+\frac{1}{2}\right]=\frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}}e_z^q\left[m,n,p+\frac{1}{2}\right]$$
$$+\frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\frac{\Delta_t}{\epsilon\delta}\frac{r_z}{r_y}\left(\left\{h_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n,p+\frac{1}{2}\right]-h_y^{q+\frac{1}{2}}\left[m-\frac{1}{2},n,p+\frac{1}{2}\right]\right\}\right.$$
$$\left.-\left\{h_x^{q+\frac{1}{2}}\left[m,n+\frac{1}{2},p+\frac{1}{2}\right]-h_x^{q+\frac{1}{2}}\left[m,n-\frac{1}{2},p+\frac{1}{2}\right]\right\}\right).\quad(9.46)$$

Thinking now in terms of coefficients, note that all the "self-term" coefficients are virtually unchanged from those given previously, i.e., C_{hxh}, C_{hyh}, C_{hxh}, C_{exe}, C_{eye}, and C_{eze} are as given

in (9.21), (9.23), (9.25), (9.27), (9.29), and (9.27), respectively. (There is a slight difference in that those expressions listed the evaluation points merely in terms of a uniform spatial step size of δ—one would now have to think in terms of Δ_x, Δ_y, and Δ_z, for displacements in the x, y, and z directions, respectively.)

The "cross" coefficients, such as C_{hxe} and C_{eyh}, are nearly the same as before where the only differences are that δ specifically represents the spatial step Δ_x, the scale factors r_y and r_z now appear, and the locations are specifically in terms of Δ_x, Δ_y, and Δ_z. These scaled coefficients are now

$$C_{hxe}(m, n+1/2, p+1/2) = \left. \frac{1}{1+\frac{\sigma_m \Delta_t}{2\mu}} \frac{\Delta_t}{\mu\delta} \frac{1}{r_y r_z} \right|_{m\Delta_x,(n+1/2)\Delta_y,(p+1/2)\Delta_z}, \tag{9.47}$$

$$C_{hye}(m+1/2, n, p+1/2) = \left. \frac{1}{1+\frac{\sigma_m \Delta_t}{2\mu}} \frac{\Delta_t}{\mu\delta} \frac{r_y}{r_z} \right|_{(m+1/2)\Delta_x,n\Delta_y,(p+1/2)\Delta_z}, \tag{9.48}$$

$$C_{hxe}(m+1/2, n+1/2, p) = \left. \frac{1}{1+\frac{\sigma_m \Delta_t}{2\mu}} \frac{\Delta_t}{\mu\delta} \frac{r_z}{r_y} \right|_{(m+1/2)\Delta_x,(n+1/2)\Delta_y,p\Delta_z}, \tag{9.49}$$

$$C_{exh}(m+1/2, n, p) = \left. \frac{1}{1+\frac{\sigma \Delta_t}{2\epsilon}} \frac{\Delta_t}{\epsilon\delta} \frac{1}{r_y r_z} \right|_{(m+1/2)\Delta_x,n\Delta_y,p\Delta_z}, \tag{9.50}$$

$$C_{eyh}(m, n+1/2, p) = \left. \frac{1}{1+\frac{\sigma \Delta_t}{2\epsilon}} \frac{\Delta_t}{\epsilon\delta} \frac{r_y}{r_z} \right|_{m\Delta_x,(n+1/2)\Delta_y,p\Delta_z}, \tag{9.51}$$

$$C_{ezh}(m, n, p+1/2) = \left. \frac{1}{1+\frac{\sigma \Delta_t}{2\epsilon}} \frac{\Delta_t}{\epsilon\delta} \frac{r_z}{r_y} \right|_{m\Delta_x,n\Delta_y,(p+1/2)\Delta_z}. \tag{9.52}$$

Finally, stability dictates that

$$\Delta_t \leq \frac{1}{c\sqrt{\frac{1}{\Delta_x^2} + \frac{1}{\Delta_y^2} + \frac{1}{\Delta_z^2}}} \tag{9.53}$$

which, after employing the definitions given above and rearranging, can be written as

$$\frac{c\Delta_t}{\delta} \leq \frac{1}{\sqrt{1 + \frac{1}{r_y^2} + \frac{1}{r_z^2}}} \tag{9.54}$$

Note that when $r_y = r_z = 1$ all the update equations and coefficients are identical to what was previously given for a uniform grid. This may seem rather odd because now we are discussing scaled fields instead of the fields themselves, e.g., we are dealing with $e_x = \Delta_x E_x$ instead of E_x. (The scaled "electric" and "magnetic" fields have units of volts and amperes, respectively, instead of volts per meter and ampere per meter, and hence these fields are really voltages and currents.) However, one must keep in mind that for these scaled fields the source terms would corresponding have to be scaled. For example, if there were an additive source current in the update equation for E_x, in the scaled version of the update equation this source term would also be scaled by Δ_x. Thus,

if one were to compare the values in a simulation involving the scaled and unscaled x-component of the electric field, the scaled values would be larger by a factor of Δ_x. When the scaled field is divided by Δ_x one obtains the same electric field that would be obtained directly from a simulation with unscaled fields.

Returning to the question raised at the beginning of the section: But how do things change if $\Delta_x \neq \Delta_y \neq \Delta_z$? The answer is that very little changes. The same code can be used for simulations with either equal or unequal spatial steps. The only differences will be in the Courant number, as given by (9.54), in the coefficients of the "cross" coefficients, as given by (9.47)–(9.52), and the fact that one is now modeling the scaled fields rather than the fields themselves. Source terms should also be appropriately scaled but that will not be considered further here.

Chapter 10

Dispersive Material

10.1 Introduction

For many problems one can obtain acceptably accurate results by assuming material parameters are constants. However, constant material parameters are inherently an approximation. For example, it is impossible to have a lossless dielectric with constant permittivity (except, of course, for free space). If such a material did exist it would violate causality. (For a material to behave causally, the Kramers-Kronig relations show that for any deviation from free-space behavior the imaginary part of the permittivity or permeability, i.e., the loss, cannot vanish for all frequencies. Nevertheless, as far as causality is concerned, the loss can be arbitrarily small.)

A non-unity, scalar, constant relative permittivity is equivalent to assuming the polarization of charge within a material is instantaneous and in perfect proportion to the applied electric field. Furthermore, the reaction is the same at all frequencies, is the same in all directions, is the same for all times, and the same proportionality constant holds for all field strengths. In reality, essentially none of the these assumptions are absolutely correct. The relationship between the electric flux density $\mathbf{D}$ and the electric field $\mathbf{E}$ can reflect all the complexity of the real world. Instead of simply having $\mathbf{D} = \epsilon\mathbf{E}$ where ϵ is a scalar constant, one can make ϵ a tensor to describe different behaviors in different directions (off diagonal terms would indicate the amount of coupling from one direction to another). The permittivity can also be written as a nonlinear function of the applied electric field to account for nonlinear media. The material parameters can be functions of time (such as might pertain to a material which is being heated). Finally, one should not forget that the permittivity can be a function of position to account for spatial inhomogeneities.

When the speed of light in a material is a function of frequency, the material is said to be dispersive. The fact that the FDTD grid is dispersive has been discussed in Chap. 7. That dispersion is a numerical artifact and is distinct from the subject of this chapter. We have also considered lossy materials. Even when the conductivity of a material is assumed to be constant, the material is dispersive (ref. (5.69) which shows that the phase constant is not linearly proportional to the frequency which must be the case for non-dispersive propagation).

When the permittivity or permeability of a material are functions of frequency, the material is dispersive. In time-harmonic form one can account for the frequency dependence of permittivity by writing $\hat{\mathbf{D}}(\omega) = \hat{\epsilon}(\omega)\hat{\mathbf{E}}(\omega)$, where a caret is used to indicate a quantity in the frequency domain.

[†]Lecture notes by John Schneider. `fdtd-dispersive-material.tex`

This expression is simple in the frequency domain, but the FDTD method is a time-domain technique. The multiplication of harmonic functions is equivalent to convolution in the time domain. Therefore it requires some additional effort to model these types of dispersive materials. We start with a brief review of dispersive materials and then consider two ways in which to model such materials in the FDTD method.

10.2 Constitutive Relations and Dispersive Media

The electric flux density and magnetic field are related to the electric field and the magnetic flux density via

$$\mathbf{D} = \epsilon_0 \mathbf{E} + \mathbf{P}, \tag{10.1}$$

$$\mathbf{H} = \frac{\mathbf{B}}{\mu_0} - \mathbf{M}, \tag{10.2}$$

where $\mathbf{P}$ and $\mathbf{M}$ account for the electric and magnetic dipoles, respectively, induced in the media. Keep in mind that force on a charge is a function of $\mathbf{E}$ and $\mathbf{B}$ so in some sense $\mathbf{E}$ and $\mathbf{B}$ are the "real" fields. The polarization vector $\mathbf{P}$ accounts for the local displacement of bound charge in a material. Because of the way in which $\mathbf{P}$ is constructed, by adding it to $\epsilon_0\mathbf{E}$ the resulting electric flux density $\mathbf{D}$ has the local effect of bound charge removed. In this way, the integral of $\mathbf{D}$ over a closed surface yields the free charge (thus Gauss's law, as expressed using the $\mathbf{D}$ field, is true whether material is present or not).

The magnetic field $\mathbf{H}$ ignores the local effect of bound charge in motion. Thus, the integration of $\mathbf{H}$ over a closed loop yields the current flowing through the surface enclosed by that loop where the current is due to either the flow of free charge or displacement current (i.e., the integral form of Ampere's law). Rearranging the terms in (10.2) and multiplying by μ_0 yields an expression for the magnetic flux density*, i.e.,

$$\mathbf{B} = \mu_0(\mathbf{H} + \mathbf{M}). \tag{10.3}$$

At a given frequency, for a linear, isotropic medium, the polarization vectors can be related to the electric and magnetic fields via an electric or magnetic susceptibility

$$\hat{\mathbf{P}}(\omega) = \epsilon_0 \hat{\chi}_e(\omega)\hat{\mathbf{E}}(\omega), \tag{10.4}$$

$$\hat{\mathbf{M}}(\omega) = \hat{\chi}_m(\omega)\hat{\mathbf{H}}(\omega), \tag{10.5}$$

where $\hat{\chi}_e(\omega)$ and $\hat{\chi}_m(\omega)$ are the electric and magnetic susceptibility, respectively.† Thus we can write

$$\hat{\mathbf{D}}(\omega) = \epsilon_0 \hat{\mathbf{E}}(\omega) + \epsilon_0 \hat{\chi}_e(\omega)\hat{\mathbf{E}}(\omega), \tag{10.6}$$

$$\hat{\mathbf{B}}(\omega) = \mu_0 \hat{\mathbf{H}}(\omega) + \mu_0 \hat{\chi}_m(\omega)\hat{\mathbf{H}}(\omega). \tag{10.7}$$

*Note that there are those (e.g., Feynman) who advocate that one should avoid using $\mathbf{D}$ and $\mathbf{H}$. Others (e.g., Sommerfeld) have discussed the unfortunate naming of the magnetic field and the magnetic flux density. However, those issues are peripheral to the main subject of interest here and we will employ the notation and usage as is commonly found in engineering electromagnetics.

†Here we will assume that $\hat{\chi}_e(\omega)$ and $\hat{\chi}_m(\omega)$ are not functions of time. Thus frequency response of the material "today" is the same as it will be "tomorrow."

For the time being we restrict discussion to the electric fields where the permittivity $\hat{\epsilon}(\omega)$ is defined as

$$\hat{\epsilon}(\omega) = \epsilon_0 \hat{\epsilon}_r(\omega) = \epsilon_0(\epsilon_\infty + \hat{\chi}_e(\omega)). \tag{10.8}$$

where $\hat{\epsilon}_r$ is the relative permittivity and, as will be seen, the constant ϵ_∞ accounts for the effect of the charged material at high frequencies where the susceptibility function goes to zero.

The time-domain electric flux density can be obtain by inverse transforming (10.6). The product of $\hat{\chi}_e$ and $\hat{\mathbf{E}}$ in the frequency domain yields a convolution in the time domain. The fields are assumed to be zero prior to $t = 0$, so this yields

$$\mathbf{D}(t) = \epsilon_0 \mathbf{E}(t) + \int_{\tau=0}^{t} \chi_e(\tau)\mathbf{E}(t-\tau)d\tau \tag{10.9}$$

where $\chi_e(t)$ is the inverse transform of $\hat{\chi}_e(\omega)$:

$$\chi_e(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{\chi}_e(\omega) e^{j\omega t} d\omega. \tag{10.10}$$

The time-domain function $\epsilon_0 \chi_e(t)$ corresponds to the polarization vector $\mathbf{P}(t)$ for an impulsive electric field—effectively the impulse response of the medium.

We now consider three common susceptibility functions. As will be shown these are based on either simple mechanical or electrical models.

10.2.1 Drude Materials

In the Drude model, which often provides a good model of the behavior of conductors, charges are assumed to move under the influence of the electric field but they experience a damping force as well. This can be described by the following simple mechanical model

$$M\frac{d^2\mathbf{x}}{dt^2} = Q\mathbf{E}(t) - Mg\frac{d\mathbf{x}}{dt} \tag{10.11}$$

where M is the mass of the charge, g is the damping coefficient, Q is the amount of charge, and $\mathbf{x}$ is the displacement of the charge (the displacement is assumed to be in an arbitrary direction and not restricted to the x axis despite the use of the symbol $\mathbf{x}$). The left side of this equation is mass times acceleration and the right side is the sum of the forces on the charge, i.e., a driving force and a damping force. Rearranging this and converting to the frequency domain yields

$$M(j\omega)^2 \hat{\mathbf{x}}(\omega) + Mg(j\omega)\hat{\mathbf{x}}(\omega) = Q\hat{\mathbf{E}}(\omega). \tag{10.12}$$

Thus the displacement can be expressed as

$$\hat{\mathbf{x}}(\omega) = -\frac{Q}{M\,(\omega^2 - jg\omega)}\hat{\mathbf{E}}(\omega). \tag{10.13}$$

The polarization vector $\mathbf{P}$ is related to the dipole moment of individual charges. If N is the number of dipoles per unit volume, the polarization vector is given by

$$\hat{\mathbf{P}} = NQ\hat{\mathbf{x}}. \tag{10.14}$$

Note that $\mathbf{P}$ has units of charge per units area (C/m^2) and thus, as would be expected from (10.1), has the same units as electric flux. Combining (10.13) and (10.14) yields

$$\hat{\mathbf{P}}(\omega) = -\epsilon_0 \frac{\frac{NQ^2}{M\epsilon_0}}{\omega^2 - jg\omega} \hat{\mathbf{E}}(\omega). \tag{10.15}$$

The electric susceptibility for Drude materials is thus given by

$$\hat{\chi}_e(\omega) = -\frac{\omega_p^2}{\omega^2 - jg\omega} \tag{10.16}$$

where $\omega_p^2 = NQ^2/(M\epsilon_0)$. However, we are not overly concerned with the specifics behind any one constant. For example, some authors may elect to combine the mass and damping coefficient which were kept as separate quantities in (10.11) (the product of the two dictates the damping force). Regardless of how the constants are defined, ultimately the Drude susceptibility will take the form shown in (10.16).

The relative permittivity for a Drude material can thus be written

$$\hat{\epsilon}_r(\omega) = \epsilon_\infty - \frac{\omega_p^2}{\omega^2 - jg\omega}. \tag{10.17}$$

Note that as ω goes to infinity the relative permittivity reduces to ϵ_∞. Consider a rather special case in which $\epsilon_\infty = 1$ and $g = 0$. When $\omega = \omega_p/\sqrt{2}$ the relative permittivity is -1. It is possible, at least to some extent, to construct a material which has not only this kind of behavior for permittivity but also for the behavior for the permeability, i.e., $\mu_r = \epsilon_r = -1$. This kind of material, which is known by various names including meta material, double-negative material, backward-wave material, and left-handed material, possesses many interesting properties. Some properties of these "meta materials" are also rather controversial as some people have made claims that others dispute (such as the ability to construct a "perfect" lens using a planar slab of this material).

The impulse response for the medium is the inverse Fourier transform of (10.16):

$$\chi_e(t) = \frac{\omega_p^2}{g}\left(1 - e^{-gt}\right) u(t) \tag{10.18}$$

where $u(t)$ is the unit step function. The factor g is seen to determine the rate at which the response goes to zero, i.e., its inverse is the relaxation time. The factor ω_p is known as the plasma frequency.

10.2.2 Lorentz Material

Lorentz material is based on a second-order mechanical model of charge motion. In this case, in addition to a damping force, there is a restoring force (effectively a spring force which wants to bring the charge back to its initial position). The sum of the forces can be expressed as

$$M\frac{d^2\mathbf{x}}{dt^2} = Q\mathbf{E}(t) - Mg\frac{d\mathbf{x}}{dt} - MK\mathbf{x}. \tag{10.19}$$

The terms in commons with those in (10.11) as they had in the case of the Drude model. The additional term represents the restoring force (which is proportional to the displacement) and is scaled by the spring constant K.

Converting to the frequency domain and rearranging yields

$$-M\omega^2\hat{\mathbf{x}}(\omega) + jMg\omega\hat{\mathbf{x}}(\omega) + MK\hat{\mathbf{x}}(\omega) = Q\hat{\mathbf{E}}(\omega). \tag{10.20}$$

Thus the displacement is given by

$$\hat{\mathbf{x}}(\omega) = \frac{Q}{M\left(K + jg\omega - \omega^2\right)}\hat{\mathbf{E}}(\omega). \tag{10.21}$$

Relating displacement to the polarization vector via $\hat{\mathbf{P}} = NQ\hat{\mathbf{x}}$ yields

$$\hat{\mathbf{P}}(\omega) = \epsilon_0\frac{NQ^2}{M\epsilon_0\left(K + jg\omega - \omega^2\right)}\hat{\mathbf{E}}(\omega). \tag{10.22}$$

From this the susceptibility is identified as

$$\hat{\chi}_e(\omega) = \frac{NQ^2}{M\epsilon_0\left(K + jg\omega - \omega^2\right)} \tag{10.23}$$

However, as before, the important thing is the form of the function, not the individual constants. We thus write this as

$$\hat{\chi}_e(\omega) = \frac{\epsilon_\ell\omega_\ell^2}{\omega_\ell^2 + 2jg_\ell\omega - \omega^2} \tag{10.24}$$

where ω_ℓ is the undamped resonant frequency, g_ℓ is the damping coefficient, and ϵ_ℓ (together with ϵ_∞) accounts for the relatively permittivity at zero frequency. The corresponding relative permittivity is given by

$$\hat{\epsilon}_r(\omega) = \epsilon_\infty + \frac{\epsilon_\ell\omega_\ell^2}{\omega_\ell^2 + 2jg_\ell\omega - \omega^2}. \tag{10.25}$$

Note that when the frequency goes to zero the relative permittivity becomes $\epsilon_\infty + \epsilon_\ell$ whereas when the frequency goes to infinity the relative permittivity is simply ϵ_∞.

The time-domain form of the susceptibility function is given by the inverse transform of (10.24). This yields

$$\chi_e(t) = \frac{\epsilon_\ell\omega_\ell^2}{\sqrt{\omega_\ell^2 - g_\ell^2}}e^{-g_\ell t}\sin\left(t\sqrt{\omega_\ell^2 - g_\ell^2}\right)u(t). \tag{10.26}$$

10.2.3 Debye Material

Debye materials can be thought of as a simple RC circuit where the amount of polarization is related to the voltage across the capacitor. The "source" driving the circuit is the electric field. For a step in the source, there may be a constant polarization. As the frequency goes to infinity, the polarization goes to zero (leaving just the constant residual high-frequency term). The susceptibility is thus written

$$\hat{\chi}_e(\omega) = \frac{\epsilon_d}{1 + j\omega\tau_d} \tag{10.27}$$

where τ_d is the time constant and ϵ_d (together with ϵ_∞) accounts for the relative permittivity when the frequency is zero. The relative permittivity is given by

$$\hat{\epsilon}_r(\omega) = \epsilon_\infty + \frac{\epsilon_d}{1 + j\omega\tau_d}. \tag{10.28}$$

The time-domain form of the susceptibility function is

$$\chi_e(t) = \frac{\epsilon_d}{\tau_d} e^{-t/\tau_d} u(t). \tag{10.29}$$

10.3 Debye Materials Using the ADE Method

The finite-difference approximation of the differential equation that relates the polarization and the electric field can be used to obtain the polarization at future times in terms of its past value and an expression involving the electric field. By doing so, one can obtain a consistent FDTD model that requires that a quantity related to the polarization be stored as an additional variable. This approach is known as the auxiliary differential equation (ADE) method.

In the frequency domain Ampere's law can be written

$$\epsilon_0\epsilon_\infty j\omega\hat{\mathbf{E}} + \sigma\hat{\mathbf{E}} + \hat{\mathbf{J}}_p = \nabla \times \hat{\mathbf{H}} \tag{10.30}$$

where the polarization current $\hat{\mathbf{J}}_p$ is given by

$$\hat{\mathbf{J}}_p = j\omega\hat{\mathbf{P}} = j\omega\epsilon_0\hat{\chi}_e\hat{\mathbf{E}}. \tag{10.31}$$

For a Debye material this becomes

$$\hat{\mathbf{J}}_p = j\omega\epsilon_0\frac{\epsilon_d}{1 + j\omega\tau}\hat{\mathbf{E}}. \tag{10.32}$$

Multiplying through by $1 + j\omega\tau$ yields

$$\hat{\mathbf{J}}_p + j\omega\tau\hat{\mathbf{J}}_p = j\omega\epsilon_0\epsilon_d\hat{\mathbf{E}}. \tag{10.33}$$

Converting to the time domain produces

$$\mathbf{J}_p + \tau\frac{\partial\mathbf{J}_p}{\partial t} = \epsilon_0\epsilon_d\frac{\partial\mathbf{E}}{\partial t}. \tag{10.34}$$

Discretizing this about the time-step $q + 1/2$ yields

$$\frac{\mathbf{J}_p^{q+1} + \mathbf{J}_p^q}{2} + \tau\frac{\mathbf{J}_p^{q+1} - \mathbf{J}_p^q}{\Delta_t} = \epsilon_0\epsilon_d\frac{\mathbf{E}^{q+1} - \mathbf{E}^q}{\Delta_t}. \tag{10.35}$$

Since we are assuming an isotropic medium, the polarization current is aligned with the electric field. Hence in (10.35) the x component of $\mathbf{J}_p$ depends only on the x component of $\mathbf{E}$ (and similarly for the y and z components).

Solving (10.35) for $\mathbf{J}_p^{q+1}$ yields

$$\mathbf{J}_p^{q+1} = \frac{1-\frac{\Delta_t}{2\tau}}{1+\frac{\Delta_t}{2\tau}}\mathbf{J}_p^q + \frac{\frac{\Delta_t}{\tau}}{1+\frac{\Delta_t}{2\tau}}\frac{\epsilon_0\epsilon_d}{\Delta_t}\left(\mathbf{E}^{q+1}-\mathbf{E}^q\right). \tag{10.36}$$

Whatever the time-constant τ is, it can be expressed in terms of some multiple of the time-step, i.e., $\tau = N_\tau \Delta_t$ where N_τ does not need to be an integer. As will be shown, it is convenient to multiply both sides of (10.36) by the spatial step size. We will assume a uniform grid in which $\Delta_x = \Delta_y = \Delta_z = \delta$. Thus (10.36) can be written

$$\delta\mathbf{J}_p^{q+1} = \frac{1-\frac{1}{2N_\tau}}{1+\frac{1}{2N_\tau}}\delta\mathbf{J}_p^q + \frac{\frac{1}{N_\tau}}{1+\frac{1}{2N_\tau}}\frac{\epsilon_0\epsilon_d\delta}{\Delta_t}\left(\mathbf{E}^{q+1}-\mathbf{E}^q\right). \tag{10.37}$$

Consider the factor $\epsilon_0\epsilon_d\delta/\Delta_t$:

$$\frac{\epsilon_0\epsilon_d\delta}{\Delta_t} = \frac{\sqrt{\epsilon_0\mu_0}\epsilon_d\delta}{\sqrt{\frac{\mu_0}{\epsilon_0}}\Delta_t} = \frac{\epsilon_d\delta}{\eta_0 c\Delta_t} = \frac{\epsilon_d}{\eta_0 S_c}. \tag{10.38}$$

Therefore (10.37) can be written

$$\delta\mathbf{J}_p^{q+1} = C_{jj}\delta\mathbf{J}_p^q + C_{je}\left(\mathbf{E}^{q+1}-\mathbf{E}^q\right), \tag{10.39}$$

where

$$C_{jj} = \frac{1-\frac{1}{2N_\tau}}{1+\frac{1}{2N_\tau}}, \tag{10.40}$$

$$C_{je} = \frac{\frac{1}{N_\tau}}{1+\frac{1}{2N_\tau}}\frac{\epsilon_d}{\eta_0 S_c}. \tag{10.41}$$

Recall that Ampere's law is discretized about the time-step $(q+1/2)\Delta_t$. Since the polarization current appears in Ampere's law, we thus need an expression for $\delta\mathbf{J}_p^{q+1/2}$. This is simply given by the average of $\delta\mathbf{J}_p^{q+1}$ (which is given by (10.39)) and $\delta\mathbf{J}_p^q$:

$$\delta\mathbf{J}_p^{q+1/2} = \frac{\delta\mathbf{J}_p^{q+1}+\delta\mathbf{J}_p^q}{2} = \frac{1}{2}\left([1+C_{jj}]\delta\mathbf{J}_p^q + C_{je}\left(\mathbf{E}^{q+1}-\mathbf{E}^q\right)\right). \tag{10.42}$$

The discrete time-domain form of Ampere's law expanded about the time-step $(q+1/2)$ is

$$\epsilon_0\epsilon_\infty\frac{\mathbf{E}^{q+1}-\mathbf{E}^q}{\Delta_t} + \sigma\frac{\mathbf{E}^{q+1}+\mathbf{E}^q}{2} + \mathbf{J}_p^{q+1/2} = \nabla\times\mathbf{H}^{q+1/2}. \tag{10.43}$$

Multiplying through by δ and using (10.42) in (10.43) yields

$$\frac{\epsilon_\infty\epsilon_0\delta}{\Delta_t}\left(\mathbf{E}^{q+1}-\mathbf{E}^q\right) + \frac{\sigma\delta}{2}\left(\mathbf{E}^{q+1}+\mathbf{E}^q\right) + \frac{1}{2}\left([1+C_{jj}]\delta\mathbf{J}_p^q + C_{je}\left(\mathbf{E}^{q+1}-\mathbf{E}^q\right)\right) = \delta\nabla\times\mathbf{H}^{q+1/2}. \tag{10.44}$$

The curl of the magnetic field, which involves spatial derivatives, will have a δ in the denominator of the finite differences. Thus $\delta\nabla \times \mathbf{H}^{q+1/2}$ will involve merely the difference of the various magnetic-field components.

Regrouping terms in (10.44) produces

$$\mathbf{E}^{q+1}\left(\frac{\epsilon_\infty\epsilon_0\delta}{\Delta_t}+\frac{\sigma\delta}{2}+\frac{1}{2}C_{je}\right)=\mathbf{E}^{q}\left(\frac{\epsilon_\infty\epsilon_0\delta}{\Delta_t}-\frac{\sigma\delta}{2}+\frac{1}{2}C_{je}\right)+\delta\nabla\times\mathbf{H}^{q+1/2}-\frac{1}{2}\left[1+C_{jj}\right]\delta\mathbf{J}_p^q. \tag{10.45}$$

The term multiplying the electric fields can be written as

$$\frac{\epsilon_\infty\epsilon_0\delta}{\Delta_t}\left(1\pm\frac{\sigma\Delta_t}{2\epsilon_\infty\epsilon_0}+\frac{C_{je}\Delta_t}{2\epsilon_\infty\epsilon_0\delta}\right)=\frac{\epsilon_\infty}{\eta_0 S_c}\left(1\pm\frac{\sigma\Delta_t}{2\epsilon_\infty\epsilon_0}+\frac{C_{je}\eta_0 S_c}{2\epsilon_\infty}\right). \tag{10.46}$$

Therefore (10.45) can be written

$$\mathbf{E}^{q+1}=\frac{1-\frac{\sigma\Delta_t}{2\epsilon_\infty\epsilon_0}+\frac{C_{je}\eta_0 S_c}{2\epsilon_\infty}}{1+\frac{\sigma\Delta_t}{2\epsilon_\infty\epsilon_0}+\frac{C_{je}\eta_0 S_c}{2\epsilon_\infty}}\mathbf{E}^{q}+\frac{\frac{\eta_0 S_c}{\epsilon_\infty}}{1+\frac{\sigma\Delta_t}{2\epsilon_\infty\epsilon_0}+\frac{C_{je}\eta_0 S_c}{2\epsilon_\infty}}\left(\delta\nabla\times\mathbf{H}^{q+1/2}-\frac{1}{2}\left[1+C_{jj}\right]\delta\mathbf{J}_p^q\right). \tag{10.47}$$

The way in which the term $\sigma\Delta_t/(2\epsilon_0)$ could be expressed in terms of the skin depth was discussed in Sec. 5.7.

The FDTD model of a Debye material would be implemented as follows:

1. Update the magnetic fields in the usual way. $\mathbf{H}^{q-1/2} \Rightarrow \mathbf{H}^{q+1/2}$.
2. For each electric-field component, do the following:
   (a) Copy the electric field to a temporary variable. $E_{tmp} = E^q$.
   (b) Update the electric field using (10.47). $E^q \Rightarrow E^{q+1}$.
   (c) Update the polarization current (actually δ times the polarization current) using (10.39). $J_p^q \Rightarrow J_p^{q+1}$. This update requires both E^{q+1} and E^q (which is stored in E_{tmp}).
3. Repeat.

The polarization current (actually the product of the spatial step-size and the polarization current, i.e., $\delta\mathbf{J}_p$) must be stored as a separate quantity. Of course it only needs to be stored for nodes at which it is non-zero.

If the polarization current is initially zero and C_{je} is zero, the polarization current will always be zero and the material behaves as a standard non-dispersive material (although, of course, dispersion is always present in the grid itself). Thus the update equations presented here can be used throughout a simulation which is a mix of dispersive and non-dispersive media. One merely has to set the constants to the appropriate values for a given location.

10.4 Drude Materials Using the ADE Method

The electric susceptibility for a Drude material is given in (10.16) and can also be written

$$\hat{\chi}_e(\omega)=\frac{\omega_p^2}{j\omega(j\omega+g)}. \tag{10.48}$$

The associated polarization current is

$$\hat{\mathbf{J}}_p = j\omega\hat{\mathbf{P}} = j\omega\epsilon_0\hat{\chi}_e\hat{\mathbf{E}} = j\omega\epsilon_0\frac{\omega_p^2}{j\omega(j\omega+g)}\hat{\mathbf{E}}. \tag{10.49}$$

Canceling $j\omega$ and cross multiplying by $g + j\omega$ yields

$$g\hat{\mathbf{J}}_p + j\omega\hat{\mathbf{J}}_p = \epsilon_0\omega_p^2\hat{\mathbf{E}}. \tag{10.50}$$

Expressed in the time-domain, this is

$$g\mathbf{J}_p + \frac{\partial\mathbf{J}_p}{\partial t} = \epsilon_0\omega_p^2\mathbf{E}. \tag{10.51}$$

Discretizing time and expanding this about the time-step $q + 1/2$ yields

$$g\frac{\mathbf{J}_p^{q+1}+\mathbf{J}_p^q}{2} + \frac{\mathbf{J}_p^{q+1}-\mathbf{J}_p^q}{\Delta_t} = \epsilon_0\omega_p^2\frac{\mathbf{E}^{q+1}+\mathbf{E}^q}{2}. \tag{10.52}$$

Solving for $\mathbf{J}_p^{q+1}$ we obtain

$$\mathbf{J}_p^{q+1} = \frac{1-\frac{g\Delta_t}{2}}{1+\frac{g\Delta_t}{2}}\mathbf{J}_p^q + \frac{1}{1+\frac{g\Delta_t}{2}}\frac{\epsilon_0\omega_p^2\Delta_t}{2}\left(\mathbf{E}^{q+1}+\mathbf{E}^q\right). \tag{10.53}$$

The damping or loss term g is the inverse of the relaxation time and hence can be expressed as a multiple of the number of time steps, i.e.,

$$g = \frac{1}{N_g\Delta_t} \tag{10.54}$$

where N_g does not need to be integer. Consider the term multiplying the electric field

$$\frac{\epsilon_0\omega_p^2\Delta_t}{2} = \frac{\sqrt{\epsilon_0\mu_0}4\pi^2 f_p^2\Delta_t}{\sqrt{\frac{\mu_0}{\epsilon_0}}2} = \frac{2\pi^2c^2\Delta_t}{\eta_0\lambda_p^2c} = \frac{2\pi^2c\Delta_t}{\eta_0(N_p\delta)^2} = \frac{2\pi^2S_c}{\eta_0N_p^2\delta} \tag{10.55}$$

where f_p is the plasma frequency in Hertz, λ_p is the free-space wavelength at this frequency, and N_p is the number of points per wavelength at the plasma frequency. Since this term contains δ in the denominator, we multiply (10.53) by δ to obtain

$$\delta\mathbf{J}_p^{q+1} = C_{jj}\delta\mathbf{J}_p^q + C_{je}\left(\mathbf{E}^{q+1}+\mathbf{E}^q\right) \tag{10.56}$$

where

$$C_{jj} = \frac{1-\frac{1}{2N_g}}{1+\frac{1}{2N_g}}, \tag{10.57}$$

$$C_{je} = \frac{1}{1+\frac{1}{2N_g}}\frac{2\pi^2S_c}{\eta_0N_p^2}. \tag{10.58}$$

Note the similarity between (10.39) and (10.56). These equations have nearly identical forms but the constants are different and there is a different sign associated with the "old" value of the electric field.

The general discretized form of Ampere's law expanded about the time-step $(q+1/2)$ is unchanged from (10.43) and is repeat below

$$\epsilon_0\epsilon_\infty \frac{\mathbf{E}^{q+1}-\mathbf{E}^q}{\Delta_t} + \sigma \frac{\mathbf{E}^{q+1}+\mathbf{E}^q}{2} + \mathbf{J}_p^{q+1/2} = \nabla \times \mathbf{H}^{q+1/2}. \tag{10.59}$$

As before, $\delta\mathbf{J}_p^{q+1/2}$ can be obtained by the average of $\delta\mathbf{J}_p^{q+1}$ and $\delta\mathbf{J}_p^q$:

$$\delta\mathbf{J}_p^{q+1/2} = \frac{\delta\mathbf{J}_p^{q+1}+\delta\mathbf{J}_p^q}{2} = \frac{1}{2}\left([1+C_{jj}]\delta\mathbf{J}_p^q + C_{je}\left(\mathbf{E}^{q+1}+\mathbf{E}^q\right)\right). \tag{10.60}$$

Multiplying through by δ and using (10.60) for the polarization current yields

$$\frac{\epsilon_\infty\epsilon_0\delta}{\Delta_t}\left(\mathbf{E}^{q+1}-\mathbf{E}^q\right) + \frac{\sigma\delta}{2}\left(\mathbf{E}^{q+1}+\mathbf{E}^q\right) + \frac{1}{2}\left([1+C_{jj}]\delta\mathbf{J}_p^q + C_{je}\left(\mathbf{E}^{q+1}+\mathbf{E}^q\right)\right) = \delta\nabla\times\mathbf{H}^{q+1/2}. \tag{10.61}$$

Note that, other than the constants being those for a Drude material, the only way in which this expression differs from (10.44) is the sign of the "old" electric field associated with the polarization current. Therefore (10.47) again yields the expression for the "future" electric field if we make the appropriate change of sign. The result is

$$\mathbf{E}^{q+1} = \frac{1-\frac{\sigma\Delta_t}{2\epsilon_\infty\epsilon_0}-\frac{C_{je}\eta_0 S_c}{2\epsilon_\infty}}{1+\frac{\sigma\Delta_t}{2\epsilon_\infty\epsilon_0}+\frac{C_{je}\eta_0 S_c}{2\epsilon_\infty}}\mathbf{E}^q + \frac{\frac{\eta_0 S_c}{\epsilon_\infty}}{1+\frac{\sigma\Delta_t}{2\epsilon_\infty\epsilon_0}+\frac{C_{je}\eta_0 S_c}{2\epsilon_\infty}}\left(\delta\nabla\times\mathbf{H}^{q+1/2} - \frac{1}{2}\left[1+C_{jj}\right]\delta\mathbf{J}_p^q\right). \tag{10.62}$$

The implementation of an FDTD algorithm for Drude material now parallels that for Debye material:

1. Update the magnetic fields in the usual way. $\mathbf{H}^{q-1/2} \Rightarrow \mathbf{H}^{q+1/2}$.
2. For each electric-field component, do the following
   (a) Copy the electric field to a temporary variable. $E_{tmp} = E^q$.
   (b) Update the electric field using (10.62). $E^q \Rightarrow E^{q+1}$.
   (c) Update the polarization current (actually δ times the polarization current) using (10.56). $J^q \Rightarrow J^{q+1}$. This update requires both E^{q+1} and E^q (which is stored in E_{tmp}).
3. Repeat.

10.5 Magnetically Dispersive Material

In the frequency domain Faraday's law is

$$\nabla\times\hat{\mathbf{E}} = -j\omega\hat{\mathbf{B}} - \sigma_m\hat{\mathbf{H}} \tag{10.63}$$

where σ_m is the magnetic conductivity. Generalizing (10.7) slightly, the permeability can be written as

$$\hat{\mu}(\omega) = \mu_0 \hat{\mu}_r(\omega) = \mu_0(\mu_\infty + \hat{\chi}_m(\omega)). \tag{10.64}$$

where the factor μ_∞ accounts for the permeability at high frequencies. Faraday's law can thus be written

$$\nabla \times \hat{\mathbf{E}} = -j\omega\mu_0\mu_\infty \hat{\mathbf{H}}(\omega) - \sigma_m \hat{\mathbf{H}} - \hat{\mathbf{J}}_m. \tag{10.65}$$

where the magnetic polarization current $\hat{\mathbf{J}}_m$ is given by

$$\hat{\mathbf{J}}_m = j\omega\mu_0 \hat{\mathbf{M}} = j\omega\mu_0 \hat{\chi}_m(\omega)\hat{\mathbf{H}}(\omega). \tag{10.66}$$

The derivation of the equations which govern an FDTD implementation of a magnetically dispersive material parallel that of electrically dispersive material. Here we consider the case of Drude dispersion where

$$\hat{\chi}_m(\omega) = -\frac{\omega_p^2}{\omega^2 - jg\omega} = \frac{\omega_p^2}{j\omega(j\omega + g)}. \tag{10.67}$$

(The plasma frequency ω_p and damping term g for magnetic susceptibility are distinct from those of electric susceptibility.) Thus the polarization current and magnetic field are related by

$$\hat{\mathbf{J}}_m = j\omega\mu_0 \frac{\omega_p^2}{j\omega(j\omega + g)}\hat{\mathbf{H}} = \frac{\mu_0\omega_p^2}{(j\omega + g)}\hat{\mathbf{H}}. \tag{10.68}$$

Multiplying by $g + j\omega$ yields

$$(g + j\omega)\hat{\mathbf{J}}_m = \mu_0\omega_p^2 \hat{\mathbf{H}}. \tag{10.69}$$

The time-domain equivalent of this is

$$g\mathbf{J}_m + \frac{\partial \mathbf{J}_m}{\partial t} = \mu_0\omega_p^2 \mathbf{H}. \tag{10.70}$$

Discretizing time and expanding this about the time-step q yields

$$g\frac{\mathbf{J}_m^{q+1/2} + \mathbf{J}_m^{q-1/2}}{2} + \frac{\mathbf{J}_m^{q+1/2} - \mathbf{J}_m^{q-1/2}}{\Delta_t} = \mu_0\omega_p^2 \frac{\mathbf{H}^{q+1/2} + \mathbf{H}^{q-1/2}}{2}. \tag{10.71}$$

Solving for $\mathbf{J}_m^{q+1/2}$ yields

$$\mathbf{J}_m^{q+1/2} = \frac{1 - \frac{g\Delta_t}{2}}{1 + \frac{g\Delta_t}{2}}\mathbf{J}_m^{q-1/2} + \frac{1}{1 + \frac{g\Delta_t}{2}}\frac{\mu_0\omega_p^2\Delta_t}{2}\left(\mathbf{H}^{q+1/2} + \mathbf{H}^{q-1/2}\right). \tag{10.72}$$

Consider the term multiplying the magnetic field

$$\frac{\mu_0\omega_p^2\Delta_t}{2} = \frac{\sqrt{\frac{\mu_0}{\epsilon_0}}\sqrt{\epsilon_0\mu_0}4\pi^2 f_p^2\Delta_t}{2} = \frac{2\pi^2\eta_0 c^2\Delta_t}{\lambda_p^2 c} = \frac{2\pi^2\eta_0 c\Delta_t}{(N_p\delta)^2} = \frac{2\pi^2\eta_0 S_c}{N_p^2\delta}. \tag{10.73}$$

where, as before, f_p is the plasma frequency in Hertz, λ_p is the free-space wavelength at this frequency, and N_p is the number of points per wavelength at the plasma frequency. Again expressing

the damping coefficient in terms of some multiple of the time step, i.e., $g = 1/(N_g\Delta_t)$, multiplying all terms in (10.72) by δ, and employing the final form of the term given in (10.73), the update equation for the polarization current can be written as

$$\delta\mathbf{J}_m^{q+1/2} = C_{jj}\delta\mathbf{J}_m^{q-1/2} + C_{jh}\left(\mathbf{H}^{q+1/2} + \mathbf{H}^{q-1/2}\right) \tag{10.74}$$

where

$$C_{jj} = \frac{1-\frac{1}{2N_g}}{1+\frac{1}{2N_g}}, \tag{10.75}$$

$$C_{jh} = \frac{1}{1+\frac{1}{2N_g}}\frac{2\pi^2\eta_0 S_c}{N_p^2}. \tag{10.76}$$

To obtain the magnetic polarization current at time-step q, the current at time-steps $q+1/2$ and $q-1/2$ are averaged:

$$\delta\mathbf{J}_m^q = \frac{\delta\mathbf{J}_m^{q+1/2} + \delta\mathbf{J}_m^{q-1/2}}{2} = \frac{1}{2}\left([1+C_{jj}]\delta\mathbf{J}_m^{q-1/2} + C_{jh}\left(\mathbf{H}^{q+1/2} + \mathbf{H}^{q-1/2}\right)\right). \tag{10.77}$$

The discretized form of Faraday's law expanded about time-step q is

$$-\mu_0\mu_\infty\frac{\mathbf{H}^{q+1/2} - \mathbf{H}^{q-1/2}}{\Delta_t} - \sigma_m\frac{\mathbf{H}^{q+1/2} - \mathbf{H}^{q-1/2}}{2} - \mathbf{J}_m^q = \nabla\times\mathbf{E}^q. \tag{10.78}$$

Multiplying through by $-\delta$ and using (10.77) for the polarization current yields

$$\begin{aligned}\frac{\mu_\infty\mu_0\delta}{\Delta_t}\left(\mathbf{H}^{q+1/2} - \mathbf{H}^{q-1/2}\right) + \frac{\sigma_m\delta}{2}\left(\mathbf{H}^{q+1/2} + \mathbf{H}^{q-1/2}\right)\\ + \frac{1}{2}\left([1+C_{jj}]\delta\mathbf{J}_m^{q-1/2} + C_{jh}\left(\mathbf{H}^{q+1/2} + \mathbf{H}^{q-1/2}\right)\right) = -\delta\nabla\times\mathbf{E}^q.\end{aligned} \tag{10.79}$$

After regrouping terms we obtain

$$\begin{aligned}\mathbf{H}^{q+1/2}\left(\frac{\mu_\infty\mu_0\delta}{\Delta_t} + \frac{\sigma_m\delta}{2} + \frac{1}{2}C_{jh}\right) =\\ \mathbf{H}^{q-1/2}\left(\frac{\mu_\infty\mu_0\delta}{\Delta_t} - \frac{\sigma_m\delta}{2} + \frac{1}{2}C_{jh}\right) - \delta\nabla\times\mathbf{E}^q - \frac{1}{2}\left[1+C_{jj}\right]\delta\mathbf{J}_m^{q-1/2}.\end{aligned} \tag{10.80}$$

The factor $\mu_0\delta/\Delta_t$ is equivalent to η_0/S_c so that the update equation for the magnetic field can be written

$$\mathbf{H}^{q+1/2} = \frac{1-\frac{\sigma_m\Delta_t}{2\mu_\infty\mu_0} - \frac{C_{jh}S_c}{2\eta_0\mu_\infty}}{1+\frac{\sigma_m\Delta_t}{2\mu_\infty\mu_0} + \frac{C_{jh}S_c}{2\eta_0\mu_\infty}}\mathbf{H}^{q-1/2} + \frac{\frac{S_c}{\eta_0\mu_\infty}}{1+\frac{\sigma_m\Delta_t}{2\mu_\infty\mu_0} + \frac{C_{jh}S_c}{2\eta_0\mu_\infty}}\left(-\delta\nabla\times\mathbf{E}^q - \frac{1}{2}\left[1+C_{jj}\right]\delta\mathbf{J}_m^{q-1/2}\right). \tag{10.81}$$

An FDTD model of a magnetically dispersive material would be implemented as follows:

1. For each magnetic-field component, do the following

(a) Copy the magnetic field to a temporary variable. $H_{tmp} = H^{q-1/2}$.

(b) Update the magnetic field using (10.81). $H^{q-1/2} \Rightarrow H^{q+1/2}$.

(c) Update the polarization current (actually δ times the polarization current) using (10.74). $J_m^{q-1/2} \Rightarrow J_m^{q+1/2}$. This update requires both $H^{q+1/2}$ and $H^{q-1/2}$ (which is stored in H_{tmp}).

2. Update the electric fields in whatever way is appropriate (which may include the dispersive implementations described previously) $\mathbf{E}^q \Rightarrow \mathbf{E}^{q+1}$.

3. Repeat.

10.6 Piecewise Linear Recursive Convolution

An alternative implementation of dispersive material is offered by the piecewise linear recursive convolution (PLRC) method. Recall that multiplication in the frequency domain is equivalent to convolution in the time domain. Thus, the frequency-domain relationship

$$\hat{\mathbf{D}}(\omega) = \epsilon_0 \epsilon_\infty \hat{\mathbf{E}}(\omega) + \epsilon_0 \hat{\chi}_e(\omega) \hat{\mathbf{E}}(\omega) \tag{10.82}$$

is equivalent to the time-domain relationship

$$\mathbf{D}(t) = \epsilon_0 \epsilon_\infty \mathbf{E}(t) + \epsilon_0 \int_{\zeta=0}^{t} \mathbf{E}(t-\zeta) \chi_e(\zeta) d\zeta \tag{10.83}$$

where ζ is a dummy variable of integration. In discrete form the electric flux density at time-step $q\Delta_t$ can be written

$$\mathbf{D}^q = \epsilon_0 \epsilon_\infty \mathbf{E}^q + \epsilon_0 \int_{\zeta=0}^{q\Delta_t} \mathbf{E}(q\Delta_t - \zeta) \chi_e(\zeta) d\zeta. \tag{10.84}$$

Although in an FDTD simulation the electric field $\mathbf{E}$ would only be available at discrete points in time, we wish to treat the field as if it varies continuously insofar as the integral is concerned. This is accomplished by assuming the field varies linearly between sample points. For example, assume the continuous variable t is between $i\Delta_t$ and $(i+1)\Delta_t$. Over this range the electric field is approximated by

$$\mathbf{E}(t) = \mathbf{E}^i + \frac{\mathbf{E}^{i+1} - \mathbf{E}^i}{\Delta_t}(t - i\Delta_t) \quad \text{for} \quad i\Delta_t \le t \le (i+1)\Delta_t. \tag{10.85}$$

When t is equal to $i\Delta_t$ we obtain $\mathbf{E}^i$ and when t is $(i+1)\Delta_t$ we obtain $\mathbf{E}^{i+1}$. The field varies linearly between these points.

To obtain a more general representation of the electric field, let us define the pulse function $p_i(t)$ which is given by

$$p_i(t) = \begin{cases} 1 & \text{if} \quad i\Delta_t \le t < (i+1)\Delta_t, \\ 0 & \text{otherwise.} \end{cases} \tag{10.86}$$

Using this pulse function the electric field can be written as

$$\mathbf{E}(t) = \sum_{i=0}^{M-1} \left[\mathbf{E}^i + \frac{\mathbf{E}^{i+1} - \mathbf{E}^i}{\Delta_t}(t - i\Delta_t) \right] p_i(t). \tag{10.87}$$

This provides a piecewise-linear approximation of the electric field over M segments. Despite the summation, the pulse function ensures that only one segment is turned on for any given value of t. Hence the summation can be thought of as serving more to collect together the various segments rather than as serving to add several terms.

In the integrand of (10.84) the argument of the electric field is $q\Delta_t - \zeta$ where $q\Delta_t$ is constant with respect to the variable of integration ζ. When ζ varies from $i\Delta_t$ to $(i+1)\Delta_t$ the electric field should vary from the discrete points $\mathbf{E}^{q-i}$ to $\mathbf{E}^{q-i-1}$. Thus, the electric field can be represented by

$$\mathbf{E}(q\Delta_t - \zeta) = \mathbf{E}^{q-i} + \frac{\mathbf{E}^{q-i-1} - \mathbf{E}^{q-i}}{\Delta_t}(\zeta - i\Delta_t) \quad \text{for} \quad i\Delta_t \le \zeta \le (i+1)\Delta_t. \tag{10.88}$$

Using the pulse function, the field over all values of ζ can be written

$$\mathbf{E}(q\Delta_t - \zeta) = \sum_{i=0}^{q-1} \left[\mathbf{E}^{q-i} + \frac{\mathbf{E}^{q-i-1} - \mathbf{E}^{q-i}}{\Delta_t}(\zeta - i\Delta_t) \right] p_i(\zeta). \tag{10.89}$$

Note the limits of the summation. The upper limit of integration is $q\Delta_t$ which corresponds to the end-point of the segment which varies from $(q-1)\Delta_t$ to $q\Delta_t$. This segment has an index of $q-1$ (segment 0 varies from 0 to Δ_t, segment 1 varies from Δ_t to $2\Delta_t$, and so on). The lower limit used here is not actually dictated by the electric field. Rather, when we combine the electric field with the susceptibility function the product is zero for ζ less than zero since the susceptibility is zero for ζ less than zero (due to the material impulse response being causal). Hence we start the lower limit of the summation at zero.

Substituting (10.89) into (10.84) yields

$$\mathbf{D}^q = \epsilon_0\epsilon_\infty \mathbf{E}^q + \epsilon_0 \int_{\zeta=0}^{q\Delta_t} \sum_{i=0}^{q-1} \left[\mathbf{E}^{q-i} + \frac{\mathbf{E}^{q-i-1} - \mathbf{E}^{q-i}}{\Delta_t}(\zeta - i\Delta_t) \right] p_i(\zeta)\chi_e(\zeta)d\zeta. \tag{10.90}$$

The summation and integration can be interchanged. However, the pulse function dictates that the integration only needs to be carried out over the range of values where the pulse function is unity. Thus we can write

$$\mathbf{D}^q = \epsilon_0\epsilon_\infty \mathbf{E}^q + \epsilon_0 \sum_{i=0}^{q-1} \int_{\zeta=i\Delta_t}^{(i+1)\Delta_t} \left[\mathbf{E}^{q-i} + \frac{\mathbf{E}^{q-i-1} - \mathbf{E}^{q-i}}{\Delta_t}(\zeta - i\Delta_t) \right] \chi_e(\zeta)d\zeta. \tag{10.91}$$

The samples of the electric field are constants with respect to the variable of integration and can be taken outside of the integral. This yields

$$\mathbf{D}^q = \epsilon_0\epsilon_\infty \mathbf{E}^q + \epsilon_0 \sum_{i=0}^{q-1} \left[\mathbf{E}^{q-i} \left(\int_{\zeta=i\Delta_t}^{(i+1)\Delta_t} \chi_e(\zeta)d\zeta \right) + \frac{\mathbf{E}^{q-i-1} - \mathbf{E}^{q-i}}{\Delta_t} \left(\int_{\zeta=i\Delta_t}^{(i+1)\Delta_t} (\zeta - i\Delta_t)\chi_e(\zeta)d\zeta \right) \right] \tag{10.92}$$

To simplify the notation, we define the following

$$\chi^i = \int_{\zeta=i\Delta_t}^{(i+1)\Delta_t} \chi_e(\zeta)d\zeta, \tag{10.93}$$

$$\xi^i = \frac{1}{\Delta_t}\int_{\zeta=i\Delta_t}^{(i+1)\Delta_t} (\zeta - i\Delta_t)\chi_e(\zeta)d\zeta. \tag{10.94}$$

This allows us to write

$$\mathbf{D}^q = \epsilon_0\epsilon_\infty\mathbf{E}^q + \epsilon_0\sum_{i=0}^{q-1}\left[\mathbf{E}^{q-i}\chi^i + \left(\mathbf{E}^{q-i-1} - \mathbf{E}^{q-i}\right)\xi^i\right]. \tag{10.95}$$

In discrete form and using the electric flux density, Ampere's law can be written

$$\nabla \times \mathbf{H}^{q+1/2} = \frac{\mathbf{D}^{q+1} - \mathbf{D}^q}{\Delta_t}. \tag{10.96}$$

Equation (10.95) gives $\mathbf{D}^q$ in terms of the electric field. This equation can also be used to express $\mathbf{D}^{q+1}$ in terms of the electric field: one merely replaces q with $q+1$. This yields

$$\mathbf{D}^{q+1} = \epsilon_0\epsilon_\infty\mathbf{E}^{q+1} + \epsilon_0\sum_{i=0}^{q}\left[\mathbf{E}^{q-i+1}\chi^i + \left(\mathbf{E}^{q-i} - \mathbf{E}^{q-i+1}\right)\xi^i\right]. \tag{10.97}$$

In order to obtain an update equation for the electric field, we must express $\mathbf{E}^{q+1}$ in terms of other known (or past) quantities. As things stand now, there is an $\mathbf{E}^{q+1}$ "buried" inside the the summation in (10.97). To express that explicitly, we extract the $i = 0$ term and then have the summation start from $i = 1$. This yields

$$\begin{aligned}\mathbf{D}^{q+1} &= \epsilon_0\epsilon_\infty\mathbf{E}^{q+1} + \epsilon_0\mathbf{E}^{q+1}\chi^0 + \epsilon_0\left(\mathbf{E}^q - \mathbf{E}^{q+1}\right)\xi^0 \\ &\quad +\epsilon_0\sum_{i=1}^{q}\left[\mathbf{E}^{q-i+1}\chi^i + \left(\mathbf{E}^{q-i} - \mathbf{E}^{q-i+1}\right)\xi^i\right].\end{aligned} \tag{10.98}$$

Ultimately we want to combine the summations in (10.95) and (10.98) and thus the limits of the summations must be the same. The limits of the summation in (10.98) can be adjusting by using a new index $i' = i - 1$ (thus $i = i' + 1$). Substituting i' for i, (10.98) can be written

$$\begin{aligned}\mathbf{D}^{q+1} &= \mathbf{E}^{q+1}\epsilon_0(\epsilon_\infty + \chi^0 - \xi^0) + \mathbf{E}^q\epsilon_0\xi^0 \\ &\quad +\epsilon_0\sum_{i'=0}^{q-1}\left[\mathbf{E}^{q-i'}\chi^{i'+1} + \left(\mathbf{E}^{q-i'-1} - \mathbf{E}^{q-i'}\right)\xi^{i'+1}\right].\end{aligned} \tag{10.99}$$

Since i' is just an index, we can return to calling is merely i.

The temporal finite-difference of the flux density is obtained by combining (10.95) and (10.99). The result is

$$\begin{aligned}\frac{\mathbf{D}^{q+1}-\mathbf{D}^{q}}{\Delta_t} &= \frac{1}{\Delta_t}\Bigg(\mathbf{E}^{q+1}\epsilon_0(\epsilon_\infty-\chi^0+\xi^0)+\mathbf{E}^{q}\epsilon_0(-\epsilon_\infty+\xi^0)\\ &\qquad -\epsilon_0\sum_{i=0}^{q-1}\left[\mathbf{E}^{q-i}\Delta\chi^i+\left(\mathbf{E}^{q-i-1}-\mathbf{E}^{q-i}\right)\Delta\xi^i\right]\Bigg) \qquad (10.100)\end{aligned}$$

where

$$\Delta\chi^i = \chi^i-\chi^{i+1}, \qquad (10.101)$$

$$\Delta\xi^i = \xi^i-\xi^{i+1}. \qquad (10.102)$$

The summation in (10.100) does not contain $\mathbf{E}^{q+1}$. Hence, using (10.100) to replace the right side of (10.96) and solving for $\mathbf{E}^{q+1}$ yields

$$\mathbf{E}^{q+1}=\frac{\epsilon_\infty-\xi^0}{\epsilon_\infty+\chi^0-\xi^0}\mathbf{E}^q+\frac{\frac{\Delta_t}{\epsilon_0}}{\epsilon_\infty+\chi^0-\xi^0}\nabla\times\mathbf{H}^{q+1/2}+\frac{1}{\epsilon_\infty+\chi^0-\xi^0}\boldsymbol{\Psi}^q \qquad (10.103)$$

where $\boldsymbol{\Psi}^q$, known as the recursive accumulator, is given by

$$\boldsymbol{\Psi}^q=\sum_{i=0}^{q-1}\left[\mathbf{E}^{q-i}\Delta\chi^i+\left(\mathbf{E}^{q-i-1}-\mathbf{E}^{q-i}\right)\Delta\xi^i\right] \qquad (10.104)$$

Equation (10.103) is used to update the electric field. It appears that a summation must be computed which requires knowledge of all the previous values of the electric field. Clearly this would be prohibitive if this were the case in practice. Fortunately, provided the material impulse response can be expressed in terms of exponentials, there is a recursive formulation which can be used to efficiently express this summation.

Consider $\boldsymbol{\Psi}^q$ with the $i=0$ term written explicitly, i.e.,

$$\boldsymbol{\Psi}^q=\mathbf{E}^q(\Delta\chi^0-\Delta\xi^0)+\mathbf{E}^{q-1}\Delta\xi^0+\sum_{i=1}^{q-1}\left[\mathbf{E}^{q-i}\Delta\chi^i+\left(\mathbf{E}^{q-i-1}-\mathbf{E}^{q-i}\right)\Delta\xi^i\right]. \qquad (10.105)$$

Employing a change of indices for the summation so that the new limits range from 0 to $q-2$, this can be written:

$$\boldsymbol{\Psi}^q=\mathbf{E}^q(\Delta\chi^0-\Delta\xi^0)+\mathbf{E}^{q-1}\Delta\xi^0+\sum_{i=0}^{q-2}\left[\mathbf{E}^{q-i-1}\Delta\chi^{i+1}+\left(\mathbf{E}^{q-i-2}-\mathbf{E}^{q-i-1}\right)\Delta\xi^{i+1}\right]. \qquad (10.106)$$

Now consider $\boldsymbol{\Psi}^{q-1}$ by writing (10.104) with q replaced by $q-1$:

$$\boldsymbol{\Psi}^{q-1}=\sum_{i=0}^{q-2}\left[\mathbf{E}^{q-i-1}\Delta\chi^i+\left(\mathbf{E}^{q-i-2}-\mathbf{E}^{q-i-1}\right)\Delta\xi^i\right] \qquad (10.107)$$

Note the similarity between the summation in (10.106) and the right-hand side of (10.107). These are the same except in (10.106) the summation involves $\Delta\chi^{i+1}$ and $\Delta\xi^{i+1}$ while in (10.107) the

summation involves $\Delta\chi^i$ and $\Delta\xi^i$. As we will see, for certain materials it is possible to relate these values to each other in a simple way. Specifically, we will find that these are related by

$$\Delta\chi^{i+1} = C_{rec}\Delta\chi^i, \tag{10.108}$$
$$\Delta\xi^{i+1} = C_{rec}\Delta\xi^i, \tag{10.109}$$

where C_{rec} is a "recursion constant" (which is yet to be determined). Given that this recursion relationship exists for $\Delta\chi^{i+1}$ and $\Delta\xi^{i+1}$, (10.106) can be written

$$\begin{aligned}\mathbf{\Psi}^q &= \mathbf{E}^q(\Delta\chi^0 - \Delta\xi^0) + \mathbf{E}^{q-1}\Delta\xi^0 + C_{rec}\sum_{i=0}^{q-2}\left[\mathbf{E}^{q-i-1}\Delta\chi^i + \left(\mathbf{E}^{q-i-2} - \mathbf{E}^{q-i-1}\right)\Delta\xi^i\right], \\ &= \mathbf{E}^q(\Delta\chi^0 - \Delta\xi^0) + \mathbf{E}^{q-1}\Delta\xi^0 + C_{rec}\mathbf{\Psi}^{q-1}, \end{aligned} \tag{10.110}$$

or, after replacing q with $q+1$, this becomes

$$\mathbf{\Psi}^{q+1} = \mathbf{E}^{q+1}(\Delta\chi^0 - \Delta\xi^0) + \mathbf{E}^q\Delta\xi^0 + C_{rec}\mathbf{\Psi}^q. \tag{10.111}$$

The PLRC algorithm is now, at least in the abstract sense, complete. The implementation is as follows:

1. Update the magnetic field in the usual way (perhaps using a dispersive formulation).
2. Update the electric field using (10.103) (being sure to first store the previous value of the electric field).
3. Updated the recursive accumulator as specified by (10.111) (using both the updated electric field and the stored value).
4. Repeat.

It now remains to determine the various constants for a given material. Specifically, one must know χ^0, ξ^0, ϵ_∞ (which appear in (10.103)), as well as $\Delta\chi^0$, $\Delta\xi^0$, and C_{rec} (which appear in (10.111)).

10.7 PLRC for Debye Material

The time-domain form of the susceptibility function for Deybe materials was given in (10.29). Using this in (10.93) and (10.94) yields

$$\chi^i = \epsilon_d\left(1 - e^{-\Delta_t/\tau_d}\right)e^{-i\Delta_t/\tau_d}, \tag{10.112}$$
$$\xi^i = \frac{\epsilon_d\tau_d}{\Delta_t}\left(1 - \left[\frac{\Delta_t}{\tau_d} + 1\right]e^{-\Delta_t/\tau_d}\right)e^{-i\Delta_t/\tau_d}. \tag{10.113}$$

Setting i equal to zero yields

$$\chi^0 = \epsilon_d\left(1 - e^{-\Delta_t/\tau_d}\right), \tag{10.114}$$
$$\xi^0 = \frac{\epsilon_d\tau_d}{\Delta_t}\left(1 - \left[\frac{\Delta_t}{\tau_d} + 1\right]e^{-\Delta_t/\tau_d}\right). \tag{10.115}$$

The time-constant τ_d can be expressed in terms of multiples of the time step Δ_t, e.g., $\tau_d = N_d\Delta_t$. Note that the time step is always divided by τ_d in these expressions so that the only important consideration is the ratio of these quantities, i.e., $\Delta_t/\tau_d = 1/N_d$.

From (10.112) we observe

$$\begin{aligned}\chi^{i+1} &= \epsilon_d\left(1-e^{-\Delta_t/\tau_d}\right)e^{-(i+1)\Delta_t/\tau_d},\\ &= e^{-\Delta_t/\tau_d}\epsilon_d\left(1-e^{-\Delta_t/\tau_d}\right)e^{-i\Delta_t/\tau_d},\\ &= e^{-\Delta_t/\tau_d}\chi^i. \end{aligned} \tag{10.116}$$

Now consider $\Delta\chi^i$ which is given by

$$\begin{aligned}\Delta\chi^i &= \chi^i - \chi^{i+1},\\ &= \chi^i - e^{-\Delta_t/\tau_d}\chi^i,\\ &= \chi^i(1-e^{-\Delta_t/\tau_d}). \end{aligned} \tag{10.117}$$

Thus $\Delta\chi^{i+1}$ can be written as

$$\begin{aligned}\Delta\chi^{i+1} &= \chi^{i+1}(1-e^{-\Delta_t/\tau_d}),\\ &= e^{-\Delta_t/\tau_d}\chi^i(1-e^{-\Delta_t/\tau_d}),\\ &= e^{-\Delta_t/\tau_d}\Delta\chi^i. \end{aligned} \tag{10.118}$$

Similar arguments pertain to ξ^i and $\Delta\xi^i$ resulting in

$$\Delta\xi^{i+1} = e^{-\Delta_t/\tau_d}\Delta\xi^i. \tag{10.119}$$

From (10.118) and (10.119), and in accordance with the description of C_{rec} given in (10.108) and (10.109), we conclude that

$$C_{rec} = e^{-\Delta_t/\tau_d}. \tag{10.120}$$

The factor $\exp(-\Delta_t/\tau_d)$, i.e., C_{rec}, appears in several of the terms given above. Writing the ratio Δ_t/τ_d as $1/N_d$, all the terms involved in the implementation of the PLRC method can be expressed as

$$C_{rec} = e^{-1/N_d}, \tag{10.121}$$

$$\chi^0 = \epsilon_d\left(1-C_{rec}\right), \tag{10.122}$$

$$\xi^0 = \epsilon_d N_d\left(1-\left[\frac{1}{N_d}+1\right]C_{rec}\right), \tag{10.123}$$

$$\Delta\chi^0 = \chi^0(1-C_{rec}), \tag{10.124}$$

$$\Delta\xi^0 = \xi^0(1-C_{rec}). \tag{10.125}$$

Nearly all the terms involved in the PLRC method are unitless and independent of scale. The only term which is not is Δ_t/ϵ_0 which multiplies the curl of the magnetic field in (10.103). However, this is a term which has appeared in all the electric-field update equations we have ever considered and, after extracting the $1/\delta$ inherent in the finite-difference form of the curl operator, it can be expressed in terms of the Courant number and characteristic impedance, i.e., $\Delta_t/(\delta\epsilon_0) = S_c\eta_0$.

Chapter 11

Perfectly Matched Layer

11.1 Introduction

The perfectly matched layer (PML) is generally considered the state-of-the-art for the termination of FDTD grids. There are some situations where specially designed ABC's can outperform a PML, but this is very much the exception rather than the rule. The theory behind a PML is typically pertinent to the continuous world. In the continuous world the PML should indeed work "perfectly" (as its name implies) for all incident angles and for all frequencies. However, when a PML is implemented in the discretized world of FDTD, there are always some imperfections (i.e., reflections) present.

There are several different PML formulations. However, all PML's essentially act as a lossy material. The lossy material, or lossy layer, is used to absorb the fields traveling away from the interior of the grid. The PML is anisotropic and constructed in such a way that there is no loss in the direction tangential to the interface between the lossless region and the PML (actually there can be loss in the non-PML region too, but we will ignore that fact for the moment). However, in the PML there is always loss in the direction normal to the interface.

The PML was originally proposed by J.-P. Bérenger in 1994. In that original work he split each field component into two separate parts. The actual field components were the sum of these two parts but by splitting the field Bérenger could create an (non-physical) anisotropic medium with the necessary phase velocity and conductivity to eliminate reflections at an interface between a PML and non-PML region. Since Bérenger first paper, others have described PML's using different approaches such as the complex coordinate-stretching technique put forward by Chew and Weedon, also in 1994.

Arguably the best PML formulation today is the Convolutional-PML (CPML). CPML constructs the PML from an anisotropic, dispersive material. CPML does not require the fields to be split and can be implemented in a relatively straightforward manner.

Before considering CPML, it is instructive to first consider a simple lossy layer. Recall that a lossy layer provided an excellent ABC for 1D grids. However, a traditional lossy layer does not work in higher dimensions where oblique incidence is possible. We will discuss this and show how the split-field PML fixes this problem.

[†]Lecture notes by John Schneider. `fdtd-pml.tex`

11.2 Lossy Layer, 1D

A lossy layer was previously introduced in Sec. 3.12. Here we will revisit lossy material but initially focus of the continuous world and time-harmonic fields. For continuous, time-harmonic fields, the governing curl equations can be written

$$\nabla \times \mathbf{H} = j\omega\epsilon\mathbf{E} + \sigma\mathbf{E} = j\omega\left(\epsilon - j\frac{\sigma}{\omega}\right)\mathbf{E} = j\omega\tilde{\epsilon}\,\mathbf{E}, \tag{11.1}$$

$$\nabla \times \mathbf{E} = -j\omega\mu\mathbf{H} - \sigma_m\mathbf{H} = -j\omega\left(\mu - j\frac{\sigma_m}{\omega}\right)\mathbf{H} = -j\omega\tilde{\mu}\mathbf{H}, \tag{11.2}$$

where the complex permittivity and permeability are given by

$$\tilde{\epsilon} = \epsilon - j\frac{\sigma}{\omega}, \tag{11.3}$$

$$\tilde{\mu} = \mu - j\frac{\sigma_m}{\omega}. \tag{11.4}$$

For now, let us restrict consideration to a 1D field that is z-polarized so that the electric field is given by

$$\mathbf{E} = \hat{\mathbf{a}}_z e^{-\gamma x} = \hat{\mathbf{a}}_z E_z(x) \tag{11.5}$$

where the propagation constant γ is yet to be determined. Given the electric field, the magnetic field is given by

$$\mathbf{H} = -\frac{1}{j\omega\tilde{\mu}}\nabla \times \mathbf{E} = -\hat{\mathbf{a}}_y \frac{\gamma}{j\omega\tilde{\mu}} E_z(x). \tag{11.6}$$

Thus the magnetic field only has a y component, i.e., $\mathbf{H} = \hat{\mathbf{a}}_y H_y(x)$.

The characteristic impedance of the medium η is the ratio of the electric field to the magnetic field (actually, in this case, the negative of that ratio). Thus,

$$\eta = -\frac{E_z(x)}{H_y(x)} = \frac{j\omega\tilde{\mu}}{\gamma}. \tag{11.7}$$

Since γ has not yet been determined, we have not actually specified the characteristic impedance yet. To determine γ we use the other curl equation where we solve for the electric field in terms of the magnetic field we just obtained:

$$\mathbf{E} = \frac{1}{j\omega\tilde{\epsilon}}\nabla \times \mathbf{H} = \frac{1}{j\omega\tilde{\epsilon}}\frac{\gamma^2}{j\omega\tilde{\mu}} e^{-\gamma x}\hat{\mathbf{a}}_z. \tag{11.8}$$

However, we already know the electric field since we started with that as a given, i.e., $\mathbf{E} = \exp(-\gamma x)\hat{\mathbf{a}}_z$. Thus, in order for (11.8) to be true, we must have

$$\frac{\gamma^2}{(j\omega)^2\tilde{\mu}\tilde{\epsilon}} = 1, \tag{11.9}$$

or, solving for γ (and only keeping the positive root)

$$\gamma = j\omega\sqrt{\tilde{\mu}\tilde{\epsilon}}. \tag{11.10}$$

Because $\tilde{\mu}$ and $\tilde{\epsilon}$ are complex, γ will be complex and we write $\gamma = \alpha + j\beta$ where α is the attenuation constant and β is the phase constant (or wave number).

Returning to the characteristic impedance as given in (11.7), we can now write

$$\eta = \frac{j\omega\tilde{\mu}}{j\omega\sqrt{\tilde{\mu}\tilde{\epsilon}}} = \sqrt{\frac{\tilde{\mu}}{\tilde{\epsilon}}}. \tag{11.11}$$

Alternatively, we can write

$$\eta = \sqrt{\frac{\mu\left(1 - j\frac{\sigma_m}{\omega\mu}\right)}{\epsilon\left(1 - j\frac{\sigma}{\omega\epsilon}\right)}}. \tag{11.12}$$

Let us now consider a z-polarized plane wave normally incident from a lossless material to a lossy material. There is a planar interface between the two media at $x = 0$. The (known) incident field is given by

$$E_z^i = e^{-j\beta_1 x} \qquad H_y^i = -\frac{1}{\eta_1}e^{-j\beta_1 x} \tag{11.13}$$

where $\beta_1 = \omega\sqrt{\mu_1\epsilon_1}$. The reflected field is given by

$$E_z^r = \Gamma e^{j\beta_1 x} \qquad H_y^r = \frac{\Gamma}{\eta_1}e^{j\beta_1 x} \tag{11.14}$$

where the only unknown is the reflection coefficient Γ. The transmitted field is given by

$$E_z^t = Te^{-\gamma_2 x} \qquad H_y^t = -\frac{T}{\eta_2}e^{-\gamma_2 x} \tag{11.15}$$

where the only unknown is the transmission coefficient T.

Both the electric field and the magnetic field are tangential to the interface at $x = 0$. Thus, the boundary conditions dictate that the sum of the incident and reflected field must equal the transmitted field at $x = 0$. Matching the electric fields at the interface yields

$$1 + \Gamma = T. \tag{11.16}$$

Matching the magnetic fields yields

$$-\frac{1}{\eta_1} + \frac{\Gamma}{\eta_1} = -\frac{T}{\eta_2} \tag{11.17}$$

or, rearranging slightly,

$$1 - \Gamma = \frac{\eta_1}{\eta_2}T. \tag{11.18}$$

Adding (11.16) and (11.18) and rearranging yields

$$T = \frac{2\eta_2}{\eta_2 + \eta_1}. \tag{11.19}$$

Plugging this back into (11.16) yields

$$\Gamma = \frac{\eta_2 - \eta_1}{\eta_2 + \eta_1}. \tag{11.20}$$

Consider the case where the media are related by

$$\frac{\mu_2}{\epsilon_2} = \frac{\mu_1}{\epsilon_1} \qquad \text{and} \qquad \frac{\sigma_m}{\mu_2} = \frac{\sigma}{\epsilon_2}. \tag{11.21}$$

We will call these conditions the "matching conditions." Under these conditions the impedances equal:

$$\eta_2 = \sqrt{\frac{\mu_2\left(1 - j\frac{\sigma_m}{\omega\mu_2}\right)}{\epsilon_2\left(1 - j\frac{\sigma}{\omega\epsilon_2}\right)}} = \sqrt{\frac{\mu_1\left(1 - j\frac{\sigma}{\omega\epsilon_2}\right)}{\epsilon_1\left(1 - j\frac{\sigma}{\omega\epsilon_2}\right)}} = \sqrt{\frac{\mu_1}{\epsilon_1}} = \eta_1. \tag{11.22}$$

When the impedances are equal, from (11.20) we see that the reflection coefficient must be zero (and the transmission coefficient is unity). We further note that this is true independent of the frequency.

As we have seen previously, this type of lossy layer can be implemented in an FDTD grid. To minimize numeric artifacts it is best to gradually increase the conductivity within the lossy region. Any field that makes it to the end of the grid will be reflected, but, because of the loss, this reflected field can be greatly attenuated. Furthermore, as the field propagates back through the lossy region toward the lossless region, it is further attenuated. Thus the reflected field from this lossy region (and the termination of the grid) can be made relatively inconsequential.

11.3 Lossy Layer, 2D

Since a lossy layer works so well in 1D and is so easy to implement, it is natural to ask if it can be used in 2D. The answer, we shall see, is that a simple lossy layer cannot be matched to the lossless region for obliquely traveling waves.

Consider a TM^z field where the incident electric field is given by

$$\begin{aligned} \mathbf{E}^i &= \hat{\mathbf{a}}_z e^{-j\mathbf{k}_1\cdot\mathbf{r}}, & (11.23)\\ &= \hat{\mathbf{a}}_z e^{-j\beta_1\cos(\theta_i)x - j\beta_1\sin(\theta_i)y}, & (11.24)\\ &= \hat{\mathbf{a}}_z e^{-j\beta_{1x}x - j\beta_{1y}y}. & (11.25) \end{aligned}$$

Knowing that the angle of incidence equals the angle of reflection (owing to the required phase matching along the interface), the reflected field is given by

$$\mathbf{E}^r = \hat{\mathbf{a}}_z \Gamma e^{j\beta_{1x}x - j\beta_{1y}y}. \tag{11.26}$$

Combining the incident and reflected field yields

$$\mathbf{E}_1 = \hat{\mathbf{a}}_z\left(e^{-j\beta_{1x}x} + \Gamma e^{j\beta_{1x}x}\right)e^{-j\beta_{1y}y} = \hat{\mathbf{a}}_z E_{1z}. \tag{11.27}$$

The magnetic field in the first medium is given by

$$\begin{aligned} \mathbf{H}_1 &= -\frac{1}{j\omega\mu_1}\nabla\times\mathbf{E}_1, & (11.28)\\ &= \hat{\mathbf{a}}_x\frac{\beta_{1y}}{\omega\mu_1}\left(e^{-j\beta_{1x}x} + \Gamma e^{\beta_{1x}x}\right)e^{-j\beta_{1y}y} + \hat{\mathbf{a}}_y\frac{\beta_{1x}}{\omega\mu_1}\left(-e^{-j\beta_{1x}x} + \Gamma e^{\beta_{1x}x}\right)e^{-j\beta_{1y}y}. & (11.29) \end{aligned}$$

The transmitted electric field is

$$\begin{aligned}\mathbf{E}^t &= \hat{\mathbf{a}}_z T e^{-\boldsymbol{\gamma}_2 \cdot \mathbf{r}} && (11.30)\\ &= \hat{\mathbf{a}}_z T e^{-\gamma_{2x} x - \gamma_{2y} y}, && (11.31)\\ &= \hat{\mathbf{a}}_z E_z^t. && (11.32)\end{aligned}$$

Plugging this expression into Maxwell's equations (or, equivalently, the wave equation) ultimately yields the constraint equation

$$\gamma_{2x}^2 + \gamma_{2y}^2 = -\omega^2 \tilde{\mu}_2 \tilde{\epsilon}_2. \tag{11.33}$$

Owing to the boundary condition that the fields must match at the interface, the propagation in the y direction (i.e., tangential to the boundary) must be the same in both media. Thus, $\gamma_{2y} = j\beta_{1y}$. Plugging this into (11.33) and solving for γ_{2x} yields

$$\gamma_{2x} = \sqrt{\beta_{1y}^2 - \omega^2 \tilde{\mu}_2 \tilde{\epsilon}_2} = \alpha_{2x} + j\beta_{2x}. \tag{11.34}$$

Note that when $\beta_{1y} = 0$, i.e., there is no propagation in the y direction and the field is normally incident on the interface, this reduces to $\gamma_{2x} = j\omega\sqrt{\tilde{\mu}_2 \tilde{\epsilon}_2}$ which is what we had for the 1D case. On the other hand, when $\sigma = \sigma_m = 0$ we obtain $\gamma_{2x} = j\left(\omega^2 \mu_2 \epsilon_2 - \beta_{1y}^2\right)^{1/2}$ where the term in parentheses is purely real (so that γ_{2x} will either be purely real or purely imaginary). The transmitted magnetic field is given by

$$\begin{aligned}\mathbf{H}^t &= -\frac{1}{j\omega\tilde{\mu}_2} \nabla \times \mathbf{E}^t, && (11.35)\\ &= \hat{\mathbf{a}}_x \frac{\beta_{1y}}{\omega \tilde{\mu}_2} E_z^t - \hat{\mathbf{a}}_y \frac{\gamma_{2x}}{j\omega\tilde{\mu}_2} E_z^t. && (11.36)\end{aligned}$$

Enforcing the boundary condition on the electric field and the y-component of the magnetic field at $x = 0$ yields

$$\begin{aligned}1 + \Gamma &= T, && (11.37)\\ \frac{\beta_{1x}}{\omega \mu_1}(-1 + \Gamma) &= -\frac{\gamma_{2x}}{j\omega\tilde{\mu}_2} T, && (11.38)\end{aligned}$$

or, rearranging the second equation,

$$1 - \Gamma = \frac{\mu_1 \gamma_{2x}}{j\tilde{\mu}_2 \beta_{1x}} T. \tag{11.39}$$

Adding (11.37) and (11.39) and rearranging yields

$$T = \frac{j\frac{2\tilde{\mu}_2}{\gamma_{2x}}}{j\frac{\tilde{\mu}_2}{\gamma_{2x}} + \frac{\mu_1}{\beta_{1x}}}. \tag{11.40}$$

Using this in (11.37) yields the reflection coefficient

$$\Gamma = \frac{j\frac{\tilde{\mu}_2}{\gamma_{2x}} - \frac{\mu_1}{\beta_{1x}}}{j\frac{\tilde{\mu}_2}{\gamma_{2x}} + \frac{\mu_1}{\beta_{1x}}}. \tag{11.41}$$

The reflection coefficient will be zero only if the terms in the numerator cancel. Let us consider these terms individually. Additionally, let us enforce a more restrictive form of the matching conditions where now $\mu_2 = \mu_1$, $\epsilon_2 = \epsilon_1$ and, as before, $\sigma/\epsilon_2 = \sigma_m/\mu_2$. The first term in the numerator can be written

$$j\frac{\tilde{\mu}_2}{\gamma_{2x}} = \frac{\tilde{\mu}_2}{\sqrt{\omega^2\tilde{\mu}_2\tilde{\epsilon}_2 - \beta_{1y}^2}}, \tag{11.42}$$

$$= \frac{\mu_1\left(1 - j\frac{\sigma}{\omega\epsilon_1}\right)}{\sqrt{\omega^2\mu_1\epsilon_1\left(1 - j\frac{\sigma}{\omega\epsilon_1}\right)^2 - \beta_{1y}^2}}, \tag{11.43}$$

$$= \frac{\mu_1}{\sqrt{\omega^2\mu_1\epsilon_1 - \frac{\beta_{1y}^2}{\left(1-j\frac{\sigma}{\omega\epsilon_1}\right)^2}}}. \tag{11.44}$$

The second term in the numerator of the reflection coefficient is

$$\frac{\mu_1}{\beta_{1x}} = \frac{\mu_1}{\sqrt{\omega^2\mu_1\epsilon_1 - \beta_{1y}^2}}. \tag{11.45}$$

Clearly (11.44) and (11.45) are not equal (unless we further require that $\sigma = 0$, but then the lossy layer is not lossy). Thus, for oblique incidence, the numerator of the reflection coefficient cannot be zero and there will always be some reflection from this lossy medium.

11.4 Split-Field Perfectly Matched Layer

To obtain a perfect match between the lossless and lossy regions, Bérenger proposed a non-physical anisotropic material known as a perfectly matched layer (PML). In a PML there is no loss in the direction tangential to the interface but there is loss normal to the interface.

First, consider the governing equations for the components of the magnetic fields for TM^z polarization. We have

$$j\omega\mu_2 H_y + \sigma_{mx}H_y = \frac{\partial E_z}{\partial x} \tag{11.46}$$

$$j\omega\mu_2 H_x + \sigma_{my}H_x = -\frac{\partial E_z}{\partial y} \tag{11.47}$$

where σ_{mx} and σ_{my} are the magnetic conductivities associated *not* with the x and y components of the magnetic field, but rather with propagation in the x or y direction. (Note that for 1D propagation in the x direction the non-zero fields are H_y and E_z while for 1D propagation in the y direction they are H_x and E_z.)

For the electric field the governing equation is

$$j\omega\epsilon_2 E_z + \sigma E_z = \frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} \tag{11.48}$$

Here there is a single conductivity and no possibility to have explicitly anisotropic behavior of the electrical conductivity. Thus, it would still be impossible to match the lossy region to the lossless one.

Bérenger's fix was to split the electric field into two (non-physical) components. To get the "actual" field, we merely sum these components. Thus we write

$$E_z = E_{zx} + E_{zy} \tag{11.49}$$

These components are governed by

$$j\omega\epsilon_2 E_{zx} + \sigma_x E_{zx} = \frac{\partial H_y}{\partial x}, \tag{11.50}$$

$$j\omega\epsilon_2 E_{zy} + \sigma_y E_{zy} = -\frac{\partial H_x}{\partial y}. \tag{11.51}$$

Note that there are now two electrical conductivities: σ_x and σ_y. If we set $\sigma_x = \sigma_y = \sigma$ and add these two equations together, we recover the original equation (11.48). Further note that if $\sigma_y = \sigma_{my} = 0$ but $\sigma_x \neq 0$ and $\sigma_{mx} \neq 0$, then a wave with components H_x and E_{zy} would not attenuate while a wave with components H_y and E_{zx} would attenuate.

Let us define the terms S_w and S_{mw} as

$$S_w = 1 + \frac{\sigma_w}{j\omega\epsilon_2} \tag{11.52}$$

$$S_{mw} = 1 + \frac{\sigma_{mw}}{j\omega\mu_2} \tag{11.53}$$

where w is either x or y. We can then write the governing equations as

$$j\omega\epsilon_2 S_x E_{zx} = \frac{\partial H_y}{\partial x}, \tag{11.54}$$

$$j\omega\epsilon_2 S_y E_{zy} = -\frac{\partial H_x}{\partial y}, \tag{11.55}$$

$$j\omega\mu_2 S_{mx} H_y = \frac{\partial}{\partial x}\left(E_{zx} + E_{zy}\right), \tag{11.56}$$

$$j\omega\mu_2 S_{my} H_x = -\frac{\partial}{\partial y}\left(E_{zx} + E_{zy}\right). \tag{11.57}$$

Taking the partial of (11.56) with respect to x and then substituting in the left-hand side of (11.54) yields

$$-\omega^2\mu_2\epsilon_2 S_x S_{mx} E_{zx} = \frac{\partial^2}{\partial x^2}\left(E_{zx} + E_{zy}\right). \tag{11.58}$$

Taking the partial of (11.57) with respect to y and then substituting in the left-hand side of (11.55) yields

$$-\omega^2\mu_2\epsilon_2 S_y S_{my} E_{zy} = \frac{\partial^2}{\partial y^2}\left(E_{zx} + E_{zy}\right). \tag{11.59}$$

Adding these two expressions together after dividing by the S terms yields

$$-\omega^2\mu_2\epsilon_2\left(E_{zx} + E_{zy}\right) = \left(\frac{1}{S_{mx}S_x}\frac{\partial^2}{\partial x^2} + \frac{1}{S_{my}S_y}\frac{\partial^2}{\partial y^2}\right)\left(E_{zx} + E_{zy}\right) \tag{11.60}$$

or, after rearranging and recalling that $E_{zx} + E_{zy} = E_z$,

$$\left(\frac{1}{S_{mx} S_x} \frac{\partial^2}{\partial x^2} + \frac{1}{S_{my} S_y} \frac{\partial^2}{\partial y^2} + \omega^2 \mu_2 \epsilon_2 \right) E_z = 0. \tag{11.61}$$

To satisfy this equation the transmitted field in the PML region would be given by

$$E_z^t = T e^{-j\sqrt{S_{mx} S_x}\beta_{2x} x - j\sqrt{S_{my} S_y}\beta_{2y} y} \tag{11.62}$$

where we must also have

$$\beta_{2x}^2 + \beta_{2y}^2 = \omega^2 \mu_2 \epsilon_2. \tag{11.63}$$

Using (11.56), the y component of the magnetic field in the PML is given by

$$\begin{aligned} H_y^t &= \frac{1}{j\omega\mu_2 S_{mx}} \frac{\partial E_z^t}{\partial x} && (11.64) \\ &= -\frac{\sqrt{S_{mx} S_x}\beta_{2x}}{\omega\mu_2 S_{mx}} E_z^t, && (11.65) \\ &= -\frac{\beta_{2x}}{\omega\mu_2} \sqrt{\frac{S_x}{S_{mx}}} E_z^t. && (11.66) \end{aligned}$$

As always, the tangential fields must match at the interface. Matching the electric fields again yields (11.37). Matching the y component of the magnetic fields yields

$$1 - \Gamma = \frac{\mu_1 \beta_{2x}}{\mu_2 \beta_{1x}} \sqrt{\frac{S_x}{S_{mx}}} T. \tag{11.67}$$

Using this and (11.37) to solve for the transmission and reflection coefficients yields

$$\begin{aligned} T &= \frac{2\frac{\beta_{1x}}{\mu_1}}{\frac{\beta_{1x}}{\mu_1} + \frac{\beta_{2x}}{\mu_2}\sqrt{\frac{S_x}{S_{mx}}}}, && (11.68) \\ \Gamma &= \frac{\frac{\beta_{1x}}{\mu_1} - \frac{\beta_{2x}}{\mu_2}\sqrt{\frac{S_x}{S_{mx}}}}{\frac{\beta_{1x}}{\mu_1} + \frac{\beta_{2x}}{\mu_2}\sqrt{\frac{S_x}{S_{mx}}}}. && (11.69) \end{aligned}$$

It is now possible to match the PML to the non-PML region so that Γ is zero. We begin by setting $\mu_2 = \mu_1$ and $\epsilon_2 = \epsilon_1$. Thus we have $\omega^2 \mu_2 \epsilon_2 = \omega^2 \mu_1 \epsilon_1$ and

$$\begin{aligned} \beta_{2x} &= \left(\omega^2 \mu_2 \epsilon_2 - \beta_{2y}^2\right)^{1/2}, && (11.70) \\ &= \left(\omega^2 \mu_1 \epsilon_1 - \beta_{2y}^2\right)^{1/2}. && (11.71) \end{aligned}$$

Recall that within the PML the propagation in the y direction is not given by β_{2y} but rather by $\sqrt{S_y S_{my}}\beta_{2y}$. Phase matching along the interface requires that

$$\sqrt{S_y S_{my}}\beta_{2y} = \beta_{1y}. \tag{11.72}$$

If we let $S_y = S_{my} = 1$, which can be realized by setting $\sigma_y = \sigma_{my} = 0$, then the phase matching condition reduces to

$$\beta_{2y} = \beta_{1y}. \tag{11.73}$$

Then, from (11.71), we have

$$\begin{aligned} \beta_{2x} &= \left(\omega^2\mu_1\epsilon_1 - \beta_{1y}^2\right)^{1/2}, & (11.74)\\ &= \beta_{1x}. & (11.75) \end{aligned}$$

Returning to (11.69), we now have

$$\begin{aligned} \Gamma &= \frac{\frac{\beta_{1x}}{\mu_1} - \frac{\beta_{1x}}{\mu_1}\sqrt{\frac{S_x}{S_{mx}}}}{\frac{\beta_{1x}}{\mu_1} + \frac{\beta_{1x}}{\mu_1}\sqrt{\frac{S_x}{S_{mx}}}}, & (11.76)\\ &= \frac{1 - \sqrt{\frac{S_x}{S_{mx}}}}{1 + \sqrt{\frac{S_x}{S_{mx}}}}. & (11.77) \end{aligned}$$

The last remaining requirement to achieve a perfect match is to have $S_x = S_{mx}$. This can be realized by having $\sigma_x/\epsilon_2 = \sigma_{mx}/\mu_2$.

To summarize, the complete set of matching conditions for a constant-x interface are

$$\begin{aligned} \epsilon_2 &= \epsilon_1 & (11.78)\\ \mu_2 &= \mu_1 & (11.79)\\ \sigma_y &= \sigma_{my} = 0 & (11.80)\\ \frac{\sigma_x}{\epsilon_2} &= \frac{\sigma_{mx}}{\mu_2}. & (11.81) \end{aligned}$$

Under these conditions propagation in the PML is governed by

$$\begin{aligned} e^{-jS_x\beta_{1x}x - j\beta_{1y}y} &= \exp\left(-j\left(1 + \frac{\sigma}{j\omega\epsilon_1}\right)\beta_{1x}x - j\beta_{1y}y\right), & (11.82)\\ &= \exp\left(-\frac{\beta_{1x}\sigma}{\omega\epsilon_1}x\right)e^{-j\beta_{1x}x - j\beta_{1y}y}. & (11.83) \end{aligned}$$

This shows that there is exponential decay in the x direction but otherwise the phase propagates in exactly the same way as it does in the non-PML region.

11.5 Un-Split PML

In the previous section we had S_w and S_{mw} where $w \in \{x, y\}$. However, once the matching condition has been applied, i.e., $\sigma_w/\epsilon_2 = \sigma_{mw}/\mu_2$, then we have $S_w = S_{mw}$. Hence we will drop the m from the subscript. Additionally, with the understanding that we are talking about the PML region, we will drop the subscript 2 from the material constants. We thus write

$$S_w = 1 + \frac{\sigma_w}{j\omega\epsilon}. \tag{11.84}$$

The conductivity in the PML is not dictated by underlying parameters in the physical space being modeled. Rather, this conductivity is set so as to minimize the reflections from the termination of the grid. In that sense σ_w is somewhat arbitrary. Therefore let us normalize the conductivity by the relative permittivity that pertains at that particular location, i.e.,

$$S_w = 1 + \frac{\sigma'_w}{j\omega\epsilon_0} \tag{11.85}$$

where $\sigma'_w = \sigma_w/\epsilon_r$. However, since the conductivity has not yet been specified, we drop the prime and merely write

$$S_w = 1 + \frac{\sigma_w}{j\omega\epsilon_0} \tag{11.86}$$

Note that there could potentially be a problem with S_w when the frequency goes to zero. In practice, in the curl equations this term is also multiplied by ω and in that sense this is not a major problem. However, if we want to move these S_w terms around, low frequencies may cause problems. To fix this, we add an additional factor to ensure that S_w remains finite as the frequency goes to zero.

We can further generalize S_w by allowing the leading term to take on values other than unity. This is effectively equivalent to allowing the relative permittivity in the PML to change. The general expression for S_w we will use is

$$S_w = \kappa_w + \frac{\sigma_w}{a_w + j\omega\epsilon_0}. \tag{11.87}$$

For the sake of considering 3D problems we also assume $w \in \{x, y, z\}$.

Dividing by the S terms, the governing equations for TM^z polarization are

$$j\omega\epsilon E_{zx} = \frac{1}{S_x}\frac{\partial H_y}{\partial x}, \tag{11.88}$$

$$j\omega\epsilon E_{zy} = -\frac{1}{S_y}\frac{\partial H_x}{\partial y}, \tag{11.89}$$

$$j\omega\mu H_y = \frac{1}{S_x}\frac{\partial E_z}{\partial x}, \tag{11.90}$$

$$j\omega\mu H_x = -\frac{1}{S_y}\frac{\partial E_z}{\partial y}. \tag{11.91}$$

Adding the first two equations together we obtain

$$j\omega\epsilon E_z = \frac{1}{S_x}\frac{\partial H_y}{\partial x} - \frac{1}{S_y}\frac{\partial H_x}{\partial y}. \tag{11.92}$$

Note that in all these equations each S_w term is always paired with the derivative in the "w" direction.

Let us define a new del operator $\tilde{\nabla}$ that incorporates this pairing

$$\tilde{\nabla} \equiv \hat{\mathbf{a}}_x \frac{1}{S_x}\frac{\partial}{\partial x} + \hat{\mathbf{a}}_y \frac{1}{S_y}\frac{\partial}{\partial y} + \hat{\mathbf{a}}_z \frac{1}{S_z}\frac{\partial}{\partial z}. \tag{11.93}$$

Using this operator Maxwell's curl equations become

$$j\omega\epsilon\mathbf{E} = \tilde{\nabla}\times\mathbf{H}, \tag{11.94}$$
$$-j\omega\mu\mathbf{H} = \tilde{\nabla}\times\mathbf{E}. \tag{11.95}$$

Note that these equations pertain to the general 3D case. This is known as the stretch-coordinate PML formulation since, as shown in (11.93), the complex S terms scale the various coordinate directions. Additionally, note there is no explicit mention of split fields in these equations. If we can find a way to implement these equations directly in the FDTD algorithm, we can avoid splitting the fields.

From these curl equations we obtain scalar equations such as (using the x-component of (11.94) and (11.95) as examples)

$$j\omega\epsilon E_x = \frac{1}{S_y}\frac{\partial H_z}{\partial y} - \frac{1}{S_z}\frac{\partial H_y}{\partial z}, \tag{11.96}$$
$$j\omega\mu H_x = -\frac{1}{S_y}\frac{\partial E_z}{\partial y} + \frac{1}{S_z}\frac{\partial E_y}{\partial z}. \tag{11.97}$$

Converting these to the time-domain yields

$$\epsilon\frac{\partial E_x}{\partial t} = \bar{S}_y \star \frac{\partial H_z}{\partial y} - \bar{S}_z \star \frac{\partial H_y}{\partial z}, \tag{11.98}$$
$$\mu\frac{\partial H_x}{\partial t} = -\bar{S}_y \star \frac{\partial E_z}{\partial y} + \bar{S}_z \star \frac{\partial E_y}{\partial z}, \tag{11.99}$$

where "$\star$" indicates convolution and $\bar{S}_w$ is the inverse Fourier transform of $1/S_w$, i.e.,

$$\bar{S}_w = \mathcal{F}^{-1}\left[\frac{1}{S_w}\right]. \tag{11.100}$$

The reciprocal of S_w is given by

$$\frac{1}{S_w} = \frac{1}{\kappa_w + \frac{\sigma_w}{a_w + j\omega\epsilon_0}} = \frac{a_w + j\omega\epsilon_0}{a_w\kappa_w + \sigma_w + j\omega\kappa_w\epsilon_0}. \tag{11.101}$$

This is of the form $(a + j\omega b)/(c + j\omega d)$ and we cannot do a partial fraction expansion since the order of the numerator and denominator are the same. Instead, we can divide the denominator into the numerator to obtain

$$\frac{a + j\omega b}{c + j\omega d} = \frac{b}{d} + \frac{a - c\frac{b}{d}}{c + j\omega d} = \frac{b}{d} + \frac{\frac{a}{c} - \frac{b}{d}}{1 + j\omega\frac{d}{c}}. \tag{11.102}$$

Recall the following Fourier transform pairs:

$$1 \Leftrightarrow \delta(t), \tag{11.103}$$
$$\frac{1}{1 + j\omega\tau} \Leftrightarrow \frac{1}{\tau}e^{-t/\tau}u(t), \tag{11.104}$$

where $\delta(t)$ is the Dirac delta function and $u(t)$ is the unit step function. Thus, we have

$$\mathcal{F}^{-1}\left[\frac{b}{d}+\frac{\frac{a}{c}-\frac{b}{d}}{1+j\omega\frac{d}{c}}\right]=\frac{b}{d}\delta(t)+\frac{ad-bc}{d^2}e^{-ct/d}u(t). \tag{11.105}$$

For the problem at hand, we have $a=a_w$, $b=\epsilon_0$, $c=a_w\kappa_w+\sigma_w$, and $d=\kappa_w\epsilon_0$. Using these values in (11.105) yields

$$\bar{S}_w=\frac{1}{\kappa_w}\delta(t)-\frac{\sigma_w}{\kappa_w^2\epsilon_0}\exp\left(-t\left[\frac{a_w}{\epsilon_0}+\frac{\sigma_w}{\kappa_w\epsilon_0}\right]\right)u(t). \tag{11.106}$$

Let us define $\zeta_w(t)$ as

$$\zeta_w(t)=-\frac{\sigma_w}{\kappa_w^2\epsilon_0}\exp\left(-t\left[\frac{a_w}{\epsilon_0}+\frac{\sigma_w}{\kappa_w\epsilon_0}\right]\right)u(t) \tag{11.107}$$

so that

$$\bar{S}_w=\frac{1}{\kappa_w}\delta(t)+\zeta_w(t). \tag{11.108}$$

Recall that the convolution of a Dirac delta function with another function yields the original function, i.e.,

$$\delta(t)\star f(t)=f(t). \tag{11.109}$$

Incorporating this fact into (11.98) yields

$$\begin{aligned}\epsilon\frac{\partial E_x}{\partial t} &= \frac{1}{\kappa_y}\frac{\partial H_z}{\partial y}-\frac{1}{\kappa_z}\frac{\partial H_y}{\partial z}\\ &\quad+\zeta_y(t)\star\frac{\partial H_z}{\partial y}-\zeta_z(t)\star\frac{\partial H_y}{\partial z}.\end{aligned} \tag{11.110}$$

Note that the first line of this equation is almost the usual governing equation. The only differences are the κ's. However, these are merely real constants. In the FDTD algorithm it is trivial to incorporate these terms in the update-equation coefficients. The second line again involves convolutions. Fortunately, these convolution are rather "benign" and, as we shall see, can be calculated efficiently using recursive convolution.

11.6 FDTD Implementation of Un-Split PML

We now wish to develop an FDTD implementation of the PML as formulated in the previous section. We start by defining the function $\Psi^q_{E_uw}$ as

$$\Psi^q_{E_uw} = \zeta_w(t)\star\left.\frac{\partial H_v}{\partial w}\right|_{t=q\Delta_t} \tag{11.111}$$

$$= \int_{\tau=0}^{q\Delta_t}\zeta_w(\tau)\frac{\partial H_v(q\Delta_t-\tau)}{\partial w}d\tau \tag{11.112}$$

where $E_u w$ in the subscript indicates this function will appear in the update of the E_u component of the electric field and it is concerned with the spatial derivative in the w direction. In (11.112) the derivative is of the H_v component of the magnetic field where w, u, and v are such that $\{w, u, v\} \in \{x, y, z\}$ and $w \neq u \neq v$.

In (11.112) note that $\zeta_w(\tau)$ is zero for $\tau < 0$ hence the integrand would be zero for $\tau < 0$. This fixes the lower limit of integration to zero. On the other hand, we assume the fields are zero prior to $t = 0$, i.e., $H_v(t)$ is zero for $t < 0$. In the convolution the argument of the magnetic field is $q\Delta_t - \tau$. This argument will be negative when $\tau > q\Delta_t$. Thus the integrand will be zero for $\tau > q\Delta_t$ and this fixes the upper limit of integration to $q\Delta_t$.

Let us assume the integration variable τ in (11.112) varies continuously but, since we are considering fields in the FDTD method, H_v varies discretely. We can still express H_v in terms of a continuously varying argument t, but it takes on discrete values. Specifically $\partial H_v(t)/\partial w$ can be represented by

$$\frac{\partial H_v(t)}{\partial w} = \sum_{i=0}^{I_{\max}} \frac{\partial H_v(i\Delta_t)}{\partial w} p_i(t) \tag{11.113}$$

where $p_i(t)$ is the unit pulse function given by

$$p_i(t) = \begin{cases} 1 & \text{if} \quad i\Delta_t \leq t < (i+1)\Delta_t, \\ 0 & \text{otherwise.} \end{cases} \tag{11.114}$$

To illustrate this further, for notational convenience let us write $f(t) = \partial H_v(t)/\partial w$. The stepwise representation of this function is shown in Fig. 11.1(a). Although not necessary, as is typical with FDTD simulations, this function is assumed to be zero for the first time-step. At $t = \Delta_t$ the function is f_1 and it remains constant until $t = 2\Delta_t$ when it changes to f_2. At $t = 3\Delta_t$ the function is f_3, and so on.

The convolution contains the function $f(q\Delta_t - \tau)$. At time-step zero (i.e., $q = 0$), this is merely $f(-\tau)$ which is illustrated in Fig. 11.1(b). Here all the sample points f_n are flipped symmetrically about the origin. We assume that the function is constant to the right of these sample points so that the function is f_1 for $-\Delta_t \leq \tau < 0$, it is f_2 for $-2\Delta_t \leq \tau < -\Delta$, f_3 for $-3\Delta_t \leq \tau < -2\Delta_t$, and so on.

Fig. 11.1(c) shows an example of $f(q\Delta_t - \tau)$ when $q \neq 0$, specifically for $q = 4$. Recall that in (11.112) the limits of integration are from zero to $q\Delta_t$—we do not need to concern ourselves with τ less than zero nor greater than $q\Delta_t$. As shown in Fig. 11.1(c), the first "pulse" extending to the right of $\tau = 0$ has a value of f_q, the pulse extending to the right of $\tau = \Delta_t$ has a value of f_{q-1}, the one to the right of $\tau = 2\Delta_t$ has a value of f_{q-2}, and so on. Thus, this shifted function can be written as

$$f(q\Delta_t - \tau) = \sum_{i=0}^{q-1} f_{q-i} p_i(\tau). \tag{11.115}$$

Returning to the derivative of the magnetic field, we write

$$\frac{\partial H_v(q\Delta_t - \tau)}{\partial w} = \sum_{i=0}^{q-1} \frac{\partial H_v^{q-i}}{\partial w} p_i(\tau) \tag{11.116}$$

where H_v^{q-i} is the magnetic field at time-step $q - i$ and, when implemented in the FDTD algorithm, the spatial derivative will be realized as a spatial finite difference.

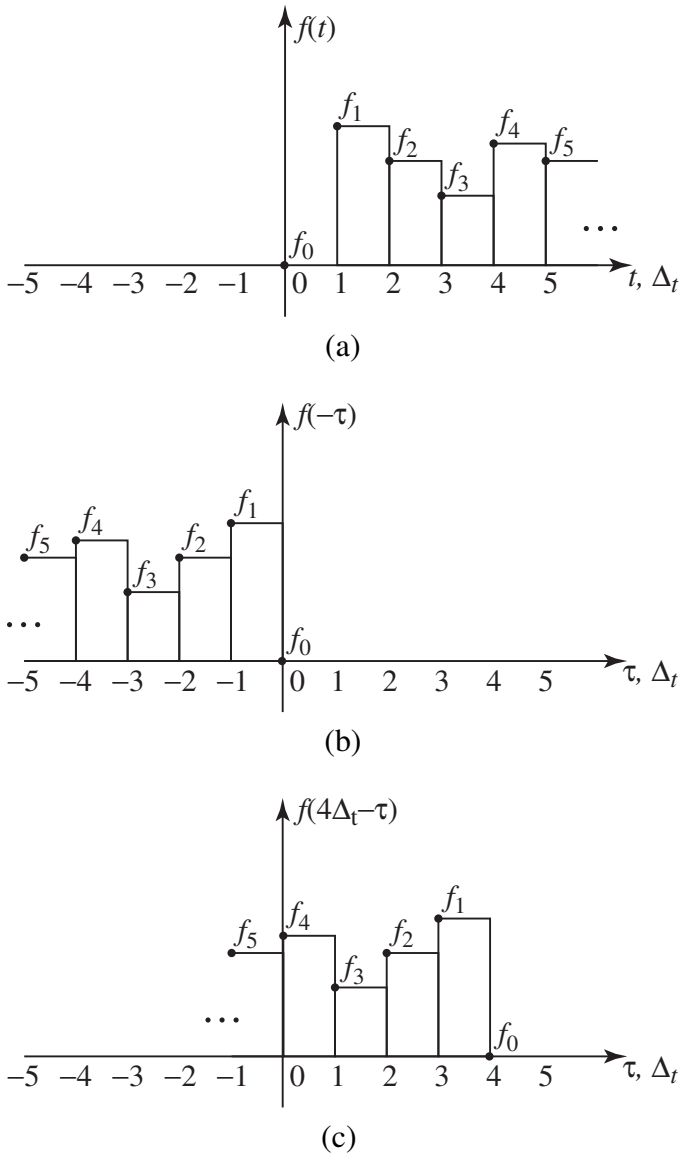


Figure 11.1: (a) Stepwise representation of a function $f(t)$. The function is a constant f_0 for $0 \leq t < \Delta_t$, f_1 for $\Delta_t \leq t < 2\Delta_t$, f_2 for $2\Delta_t \leq t < 3\Delta_t$, and so on. (b) Stepwise representation of a function $f(-\tau)$. Here the constants f_n are flipped about the origin but the pulses still extend for one time-step to the right of the corresponding point. Hence the function is a constant f_1 for $-\Delta_t \leq \tau < 0$, f_2 for $-2\Delta_t \leq \tau < -\Delta_t$, etc. (c) Stepwise representation of the function $f(q\Delta_t - \tau)$ when $q = 4$.

At time-step q, $\Psi_{E_u w}$ is given by

$$\Psi^q_{E_u w} = \int_{\tau=0}^{q\Delta_t} \zeta_w(\tau) \sum_{i=0}^{q-1} \frac{\partial H_v^{q-i}}{\partial w} p_i(\tau) d\tau \tag{11.117}$$

Interchanging the order of summation and integration yields

$$\begin{aligned} \Psi^q_{E_u w} &= \sum_{i=0}^{q-1} \frac{\partial H_v^{q-i}}{\partial w} \int_{\tau=0}^{q\Delta_t} \zeta_w(\tau) p_i(\tau) d\tau, & (11.118)\\ &= \sum_{i=0}^{q-1} \frac{\partial H_v^{q-i}}{\partial w} \int_{\tau=i\Delta_t}^{(i+1)\Delta_t} \zeta_w(\tau) d\tau, & (11.119) \end{aligned}$$

where, in going from (11.118) to (11.119), the pulse function was used to establish the limits of integration.

Consider the following integral

$$\begin{aligned} -\int_{i\Delta}^{(i+1)\Delta} e^{-at} dt &= \frac{1}{a} e^{-at} \Big|_{i\Delta}^{(i+1)\Delta} & (11.120)\\ &= \frac{1}{a} \left(e^{-a(i+1)\Delta} - e^{-ai\Delta} \right) & (11.121)\\ &= \frac{1}{a} \left(e^{-a\Delta} - 1 \right) e^{-ai\Delta} & (11.122)\\ &= \frac{1}{a} \left(e^{-a\Delta} - 1 \right) \left(e^{-a\Delta} \right)^i & (11.123) \end{aligned}$$

Keeping this in mind, the integration in (11.119) can be written as

$$\begin{aligned} \int_{\tau=i\Delta_t}^{(i+1)\Delta_t} \zeta_w(\tau) d\tau &= -\frac{\sigma_w}{\kappa_w^2 \epsilon_0} \int_{\tau=i\Delta_t}^{(i+1)\Delta_t} \exp\left(-\tau \left[\frac{a_w}{\epsilon_0} + \frac{\sigma_w}{\kappa_w \epsilon_0} \right] \right) d\tau & (11.124)\\ &= C_w (b_w)^i & (11.125) \end{aligned}$$

where

$$\begin{aligned} b_w &= \exp\left(- \left[\frac{a_w}{\epsilon_0} + \frac{\sigma_w}{\kappa_w \epsilon_0} \right] \Delta_t \right), & (11.126)\\ C_w &= \frac{\sigma_w}{\sigma_w \kappa_w + \kappa_w^2 a_w} \left(b_w - 1 \right). & (11.127) \end{aligned}$$

Note that in (11.125) b_w is raised to the power i which is an integer index. It is now possible to express $\Psi^q_{E_u w}$ as

$$\Psi^q_{E_u w} = \sum_{i=0}^{q-1} \frac{\partial H_v^{q-i}}{\partial w} C_w (b_w)^i. \tag{11.128}$$

Let us explicitly separate the $i = 0$ term from the rest of the summation:

$$\Psi^q_{E_u w} = C_w \frac{\partial H^q_v}{\partial w} + \sum_{i=1}^{q-1} \frac{\partial H^{q-i}_v}{\partial w} C_w (b_w)^i. \tag{11.129}$$

Replacing the index i with $i' = i - 1$ (so that $i = i' + 1$), this becomes

$$\Psi^q_{E_u w} = C_w \frac{\partial H^q_v}{\partial w} + \sum_{i'=0}^{q-2} \frac{\partial H^{q-i'-1}_v}{\partial w} C_w (b_w)^{i'+1}. \tag{11.130}$$

Dropping the prime from the index and rearranging slightly yields

$$\Psi^q_{E_u w} = C_w \frac{\partial H^q_v}{\partial w} + b_w \sum_{i=0}^{[q-1]-1} \frac{\partial H^{[q-1]-i}_v}{\partial w} C_w (b_w)^i. \tag{11.131}$$

Comparing the summation in this expression to the one in (11.128) one sees that this expression can be written as

$$\Psi^q_{E_u w} = C_w \frac{\partial H^q_v}{\partial w} + b_w \Psi^{q-1}_{E_u w}. \tag{11.132}$$

Note that $\Psi_{E_u w}$ at time-step q is a function of $\Psi_{E_u w}$ at time-step $q - 1$. Thus $\Psi_{E_u w}$ can be updated recursively—there is no need to store the entire history of $\Psi_{E_u w}$ to obtain the next value. As is typical with FDTD, one merely needs to know $\Psi_{E_u w}$ at the previous time step.

We now have all the pieces in place to implement a PML in the FDTD method. The algorithm to update the electric fields is

1. Update the $\Psi_{E_u w}$ terms employing the recursive update equation given by (11.132). Recall that these $\Psi_{E_u w}$ functions represent the convolutions given in the second line of (11.110).

2. Update the electric fields in the standard way. However, incorporate the κ's where appropriate. Essentially one employs the update equation implied by the top line of (11.110) (where that equation applies to E_x and similar equations apply to E_y and E_z).

3. Apply, i.e., add or subtract, $\Psi_{E_u w}$ to the electric field as indicated by the second line of (11.110).

This completes the update of the electric field.

The magnetic fields are updated in a completely analogous manner. First the Ψ functions that pertain to the magnetic fields are updated (in this case there are $\Psi_{H_u w}$ functions that involve the spatial derivatives of the electric fields), then the magnetic fields are updated in the usual way (accounting for any κ's), and finally the Ψ functions are applied to the magnetic fields (i.e., added or subtracted).

Chapter 12

Acoustic FDTD Simulations

12.1 Introduction

The FDTD method employs finite-differences to approximate Ampere's and Faraday's laws. Ampere's and Faraday's laws are first-order differential equations that couple the electric and magnetics fields. As we have seen, with a judicious discretization of space and time, the resulting equations can be solved for "future" fields in terms of known past fields.

Other physical phenomena are also described by coupled first-order differential equations where the temporal derivative of one field is related to the spatial derivative of another field. Both acoustics and elastic wave propagation are such phenomena. Here we will consider only acoustic propagation. Specifically we will consider small-signal acoustics which can be described in terms of the scalar pressure field $P(x, y, z, t)$ and the vector velocity $\mathbf{v}(x, y, z, t)$. The material parameters are the speed of sound c_a and the density ρ (both of which can vary as a function of position).

The governing acoustic equations in three dimensions are

$$\frac{\partial P}{\partial t} = -\rho c_a^2 \nabla \cdot \mathbf{v}, \tag{12.1}$$

$$\frac{\partial \mathbf{v}}{\partial t} = -\frac{1}{\rho} \nabla P, \tag{12.2}$$

or, expanded in terms of the components,

$$\frac{\partial P}{\partial t} = -\rho c_a^2 \left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} \right), \tag{12.3}$$

$$\frac{\partial v_x}{\partial t} = -\frac{1}{\rho} \frac{\partial P}{\partial x}, \tag{12.4}$$

$$\frac{\partial v_y}{\partial t} = -\frac{1}{\rho} \frac{\partial P}{\partial y}, \tag{12.5}$$

$$\frac{\partial v_z}{\partial t} = -\frac{1}{\rho} \frac{\partial P}{\partial z}. \tag{12.6}$$

[†]Lecture notes by John Schneider. `fdtd-acoustics.tex`

Equation (12.2) is essentially a variation of Newton's second law, $\mathbf{F} = m\mathbf{a}$, where instead of acceleration $\mathbf{a}$ there is the derivative of the velocity, instead of mass m there is the mass density, and instead of force $\mathbf{F}$ there is the derivative of pressure. Pressure is force per area and the negative sign accounts for the fact that if pressure is building in a particular direction that tends to cause acceleration in the opposite direction. Equation (12.1) comes from an equation of state for the material (with various approximations assumed along the way).

Taking the divergence of (12.2) and interchanging the order of temporal and spatial differentiation yields

$$\frac{\partial}{\partial t}\nabla \cdot \mathbf{v} = -\frac{1}{\rho}\nabla^2 P. \tag{12.7}$$

Taking the temporal derivative of (12.1) and using (12.7) yields

$$\frac{\partial^2 P}{\partial t^2} = -\rho c_a^2 \frac{\partial}{\partial t}\nabla \cdot \mathbf{v} = c_a^2 \nabla^2 P. \tag{12.8}$$

Rearranging this yields the wave equation

$$\nabla^2 P - \frac{1}{c_a^2}\frac{\partial^2 P}{\partial t^2} = 0. \tag{12.9}$$

Thus the usual techniques and solutions one is familiar with from electromagnetics carry over to acoustics. For example, a harmonic plane wave given by

$$P(x, y, z, t) = P_0 e^{-j\boldsymbol{\beta}\cdot\mathbf{r}} e^{j\omega t} \tag{12.10}$$

is a valid solution to the governing equations where P_0 is a constant and the wave vector $\boldsymbol{\beta}$ can be written

$$\boldsymbol{\beta} = \beta_x \hat{\mathbf{a}}_x + \beta_y \hat{\mathbf{a}}_y + \beta_z \hat{\mathbf{a}}_z = \beta \hat{\mathbf{a}}_\beta = (\omega/c_a)\hat{\mathbf{a}}_\beta. \tag{12.11}$$

Substituting (12.10) into (12.4) and assuming $\exp(j\omega t)$ temporal dependence yields

$$j\omega v_x = \frac{1}{\rho}(-j\beta_x)P. \tag{12.12}$$

Rearranging terms gives

$$v_x = \frac{\beta_x}{\rho\omega}P. \tag{12.13}$$

Following the same steps for the y and z components produces

$$v_y = \frac{\beta_y}{\rho\omega}P, \tag{12.14}$$

$$v_z = \frac{\beta_z}{\rho\omega}P. \tag{12.15}$$

Thus the harmonic velocity is given by

$$\mathbf{v} = v_x \hat{\mathbf{a}}_x + v_y \hat{\mathbf{a}}_y + v_z \hat{\mathbf{a}}_z = \frac{1}{\rho\omega}(\beta_x \hat{\mathbf{a}}_x + \beta_y \hat{\mathbf{a}}_y + \beta_z \hat{\mathbf{a}}_z)P = \frac{\beta}{\rho\omega}P\hat{\mathbf{a}}_\beta. \tag{12.16}$$

Since the wave number, i.e., the magnitude of the wave vector, is given by $\beta = \omega/c_a$, the ratio of the magnitude of pressure to the velocity is given by

$$\left|\frac{P}{\mathbf{v}}\right| = \rho c_a. \tag{12.17}$$

The term on the right-hand side is known as the characteristic impedance of the medium which is often written as Z.

12.2 Governing FDTD Equations

To obtain an FDTD algorithm for acoustic propagation, the pressure and components of velocity are discretized in both time and space. In electromagnetics there were two vector fields and hence six field-components that had to be arranged in space-time. In acoustics there is one scalar field and one vector field. Thus there are only four field-components.

To implement a 3D acoustic FDTD algorithm, a suitable arrangement of nodes is as shown in Fig. 12.1. A pressure node is surrounded by velocity components such that the components are oriented along the line joining the component and the pressure node. This should be contrasted to the arrangement of nodes in electromagnetic grids where the components of the magnetic field swirled around the components of the electric field, and vice versa. In electromagnetics one is modeling coupled curl equations where the partial derivatives are related to behavior orthogonal to the direction of the derivative. In acoustics, where the governing equations involve the divergence and gradient, the partial derivatives are associated with behavior in the direction of the derivative.

The arrangement of nodes in a 2D grid is illustrated in Fig. 12.2. This should be compared to the 2D electromagnetic grids, i.e., Fig. 8.1 for the TM^z case and Fig. 8.9 for the TE^z case. (Because pressure is inherently a scalar field, there are not two different polarization associated with 2D acoustic simulations—nor is there a notion of polarization in three dimensions.)

In addition to the spatial offsets, the pressure nodes are assumed to be offset a half temporal step from the velocity nodes (but all the velocity components exist at the same time-step). The following notation will be used with an implicit understanding of spatial offsets

$$\begin{aligned}
P(x,y,z,t) &= P(m\Delta_x, n\Delta_y, p\Delta_z, q\Delta_t) = P^q[m,n,p], && (12.18)\\
v_x(x,y,z,t) &= v_x\left([m+1/2]\Delta_x, n\Delta_y, p\Delta_z, [q+1/2]\Delta_t\right) = v_x^{q+1/2}[m,n,p], && (12.19)\\
v_y(x,y,z,t) &= v_y\left(m\Delta_x, [n+1/2]\Delta_y, p\Delta_z, [q+1/2]\Delta_t\right) = v_y^{q+1/2}[m,n,p], && (12.20)\\
v_z(x,y,z,t) &= v_z\left(m\Delta_x, n\Delta_y, [p+1/2]\Delta_z, [q+1/2]\Delta_t\right) = v_z^{q+1/2}[m,n,p]. && (12.21)
\end{aligned}$$

We will assume the spatial step sizes are the same, i.e., $\Delta_x = \Delta_y = \Delta_z = \delta$.

Replacing the derivatives in (12.3) with finite differences and using the discretization of (12.18)–(12.21) yields the following update equation:

$$\begin{aligned}
P^q[m,n,p] = P^{q-1}[m,n,p] - \rho c_a^2 \frac{\Delta_t}{\delta}\Big(&v_x^{q-1/2}[m,n,p] - v_x^{q-1/2}[m-1,n,p] + \\
&v_y^{q-1/2}[m,n,p] - v_y^{q-1/2}[m,n-1,p] + \\
&v_z^{q-1/2}[m,n,p] - v_z^{q-1/2}[m,n,p-1]\Big). \qquad (12.22)
\end{aligned}$$

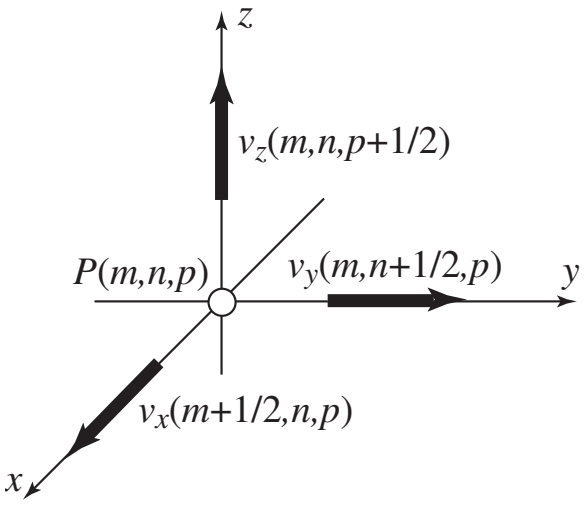


Figure 12.1: An acoustic unit cell in three dimensions showing the arrangement of velocity nodes relative to the pressure node with the same spatial indices.

The sound speed and the density can be functions of space. Let us assume that the density and sound speed are specified at the grid points corresponding to the location of pressure nodes. Additionally, assume that the sound speed can be defined in terms of a background sound speed c_0 and a relative sound speed c_r:

$$c_a = c_r c_0. \tag{12.23}$$

The background sound speed corresponds to the fastest speed of propagation at any location in the grid so that $c_r \leq 1$. The coefficient of the spatial finite-difference in (12.22) can now be written

$$\rho c_a^2 \frac{\Delta_t}{\delta} = \rho c_r^2 c_0 \frac{c_0 \Delta_t}{\delta} = \rho c_r^2 c_0 S_c \tag{12.24}$$

where, similar to electromagnetics, the Courant number is $S_c = c_0 \Delta_t / \delta$. The explicit spatial dependence of the density and sound speed can be emphasized by writing the coefficient as

$$\rho[m, n, p] c_r^2[m, n, p] c_0 S_c \tag{12.25}$$

where $\rho[m, n, p]$ is the density that exists at the same point as the pressure node $P[m, n, p]$ and $c_r[m, n, p]$ is the relative sound speed at this same point. Note that the Courant number S_c and the background sound speed c_0 are independent of position. Furthermore, the entire coefficient is independent of time.

The update equation for the x component of velocity is obtained from the discretized version of (12.4) which yields

$$v_x^{q+1/2}[m, n, p] = v_x^{q-1/2}[m, n, p] - \frac{1}{\rho} \frac{\Delta_t}{\delta} \left(P^q[m + 1, n, p] - P^q[m, n, p] \right) \tag{12.26}$$

The coefficient of this equation does not contain the Courant number but that can be obtained by multiplying and dividing by the background sound speed

$$\frac{1}{\rho} \frac{\Delta_t}{\delta} = \frac{1}{\rho c_0} \frac{c_0 \Delta_t}{\delta} = \frac{1}{\rho c_0} S_c. \tag{12.27}$$

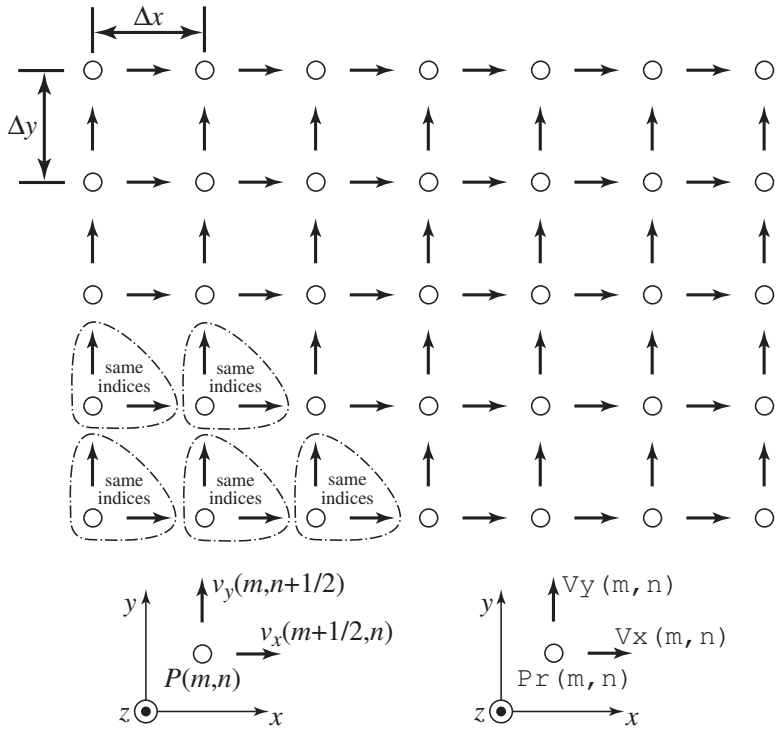


Figure 12.2: The arrangement of nodes in a 2D acoustic simulation. In a computer FDTD implementation the nodes shown within the dashed enclosures will have the same spatial indices. This is illustrated by the two depictions of a unit cell at the bottom of the figure. The one on the left shows the nodes with the spatial offsets given explicitly. The one on the right shows the corresponding node designations that would be used in a computer program. (Here `Pr` is used for the pressure array.)

We wish to define the density only at the pressure nodes. Since the x-component of the velocity is offset from the pressure a half spatial step in the x direction, what is the appropriate velocity to use? The answer, much as it was in the case of an interface between two different materials in electromagnetics, is the average of the densities to either side of the pressure node (where the notion of "either side" is dictated by the orientation of the velocity node). Therefore the coefficient can be written

$$\frac{1}{\left(\frac{\rho[m+1,n,p]+\rho[m,n,p]}{2}\right)c_0}S_c = \frac{2S_c}{(\rho[m+1,n,p]+\rho[m,n,p])c_0}. \tag{12.28}$$

The update equations for the velocity components can now be written as

$$\begin{aligned} v_x^{q+1/2}[m,n,p] &= v_x^{q-1/2}[m,n,p] - \\ &\quad \frac{2S_c}{(\rho[m,n,p]+\rho[m+1,n,p])c_0}\left(P^q[m+1,n,p]-P^q[m,n,p]\right), \end{aligned} \tag{12.29}$$

$$\begin{aligned} v_y^{q+1/2}[m,n,p] &= v_y^{q-1/2}[m,n,p] - \\ &\quad \frac{2S_c}{(\rho[m,n,p]+\rho[m,n+1,p])c_0}\left(P^q[m,n+1,p]-P^q[m,n,p]\right), \end{aligned} \tag{12.30}$$

$$\begin{aligned} v_z^{q+1/2}[m,n,p] &= v_z^{q-1/2}[m,n,p] - \\ &\quad \frac{2S_c}{(\rho[m,n,p]+\rho[m,n,p+1])c_0}\left(P^q[m,n,p+1]-P^q[m,n,p]\right). \end{aligned} \tag{12.31}$$

12.3 Two-Dimensional Implementation

Let us consider a 2D simulation in which the fields vary in the x and y directions. The grid would be as shown in Fig. 12.2 and it is assumed that $\Delta_x = \Delta_y = \delta$. Assume the arrays `pr`, `vx`, and `vy` hold the pressure, x component of the velocity, and the y component of the velocity, respectively. Similar to the electromagnetic implementation, assume the macros `Pr`, `Vx`, and `Vy` have been created to facilitate accessing these arrays (ref. Sec. 8.2). The update equations can be written

```
Vx(m,n) = Vx(m,n) - Cvxp(m,n)*(Pr(m+1,n) - Pr(m,n));
Vy(m,n) = Vy(m,n) - Cvyp(m,n)*(Pr(m,n+1) - Pr(m,n));
Pr(m,n) = Pr(m,n) - Cprv(m,n)*((Vx(m,n) - Vx(m-1,n))
                             + (Vy(m,n) - Vy(m,n-1)));
```

where the coefficient arrays are given by

$$C_{vxp}(m,n) = \left.\frac{1}{\rho c_0}S_c\right|_{(m+1/2)\delta,n\delta} = \frac{2S_c}{(\rho[m+1,n]+\rho[m,n])c_0}, \tag{12.32}$$

$$C_{vyp}(m,n) = \left.\frac{1}{\rho c_0}S_c\right|_{m\delta,(n+1/2)\delta} = \frac{2S_c}{(\rho[m,n]+\rho[m,n+1])c_0}, \tag{12.33}$$

$$C_{prv}(m,n) = \rho[m,n]c_r^2[m,n]c_0S_c. \tag{12.34}$$

These update equations are little different from those for the TM^z case. Referring to Sec. 8.3 for lossless materials, the TM^z update equations are

```
Hy(m,n) = Hy(m,n) + Chye(m,n)*(Ez(m+1,n) - Ez(m,n));
Hx(m,n) = Hx(m,n) - Chxe(m,n)*(Ez(m,n+1) - Ez(m,n));
Ez(m,n) = Ez(m,n) + Cezh(m,n)*((Hy(m,n) - Hy(m-1,n))
                              -(Hx(m,n) - Hx(m,n-1)));
```

There is a one-to-one mapping between these sets of equations. One can equate values as follows

$$v_x \Leftrightarrow -H_y, \tag{12.35}$$
$$v_y \Leftrightarrow H_x, \tag{12.36}$$
$$P \Leftrightarrow E_z, \tag{12.37}$$
$$C_{vxp} \Leftrightarrow C_{hye}, \tag{12.38}$$
$$C_{vyp} \Leftrightarrow C_{hxe}, \tag{12.39}$$
$$C_{prv} \Leftrightarrow C_{ezh}. \tag{12.40}$$

Thus, converting 2D programs that were written to model electromagnetic field propagation to ones that can model acoustic propagation is surprisingly straightforward. Essentially, all one has to do is change some labels and a few signs.

For TE^z simulations, the updated equations for a lossless medium were

```
Hz(m,n) = Hz(m,n) +
          Chze(m,n)*((Ex(m,n+1)-Ex(m,n))-(Ey(m+1,n)-Ey(m,n)));
Ex(m,n) = Ex(m,n) + Cexh(m,n)*(Hz(m,n)-Hz(m,n-1));
Ey(m,n) = Ey(m,n) - Ceyh(m,n)*(Hz(m,n)-Hz(m-1,n));
```

In this case the conversion from the electromagnetic equations to the acoustic equations can be accomplished with the following mapping

$$v_x \Leftrightarrow E_y, \tag{12.41}$$
$$v_y \Leftrightarrow -E_x, \tag{12.42}$$
$$P \Leftrightarrow H_z, \tag{12.43}$$
$$C_{vxp} \Leftrightarrow C_{eyh}, \tag{12.44}$$
$$C_{vyp} \Leftrightarrow C_{exh}, \tag{12.45}$$
$$C_{prv} \Leftrightarrow C_{hxe}. \tag{12.46}$$

For three dimensions 3D acoustic code is arguably simpler than the electromagnetic case since there are not two vector fields. However porting 3D electromagnetic algorithms to the acoustic case is not as trivial as in two dimensions.

Chapter 13

Parallel Processing

The FDTD method is said to be "trivially parallelizable," meaning that there are several simple ways in which the algorithm can be divided into tasks that can be executed simultaneously. For example, in a 3D simulation one might write an FDTD program that simultaneously updates the E_x, E_y, and E_z components of the electric field. These updates depend on the magnetic field and previous values of themselves—they are not a function of each other and hence can be updated in parallel. Then, H_x, H_y, and H_z might be updated simultaneously. Alternatively, one might divide the computational domain into distinct, non-overlapping regions and assign different processors to update the fields in those regions. This way fields in each of the regions could be updated simultaneously.

Here we will example two approaches to parallelizing a program. The threading approach typically use one computer to run a program. A threaded program is designed in such a way as to split the total computation between two or more "threads." If the computer has multiple processors, these threads can be executed simultaneously. Each of the threads can share the same memory space, i.e., the same set of variables. An alternative approach to parallelization uses the Message Passing Interface (MPI) protocol. This protocol allows different computers to run programs that pass information back and forth. MPI is ideally suited to partitioning the computational domain into multiple non-overlapping regions. Different computers are used to update the fields in the different regions. To update the fields on the interfaces with different regions, the computers have to pass information back and forth about the tangential fields along those interfaces.

In this chapter we provide some simple examples illustrating the use of threads and MPI.

13.1 Threads

There are different threading packages available. Perhaps the most common is the POSIX threads (pthreads) package. To use pthreads, you must include the header file `pthread.h` in your program. When linking, you must link to the pthread library (which is accomplished with the compiler flag `-lpthread`).

There are many functions related to pthreads. On a UNIX-based system on which pthreads are installed, a list of these functions can typically be obtained with the command `man -k pthread` and then one can see the individual man-pages to obtain details about a specific function.

Lecture notes by John Schneider. `parallel-processing.tex`

Despite all these functions, it is possible to do a great deal of useful programming using only two functions: `pthread_create()` and `pthread_join()`. As the name implies, a thread is created by the function `pthread_create()`. You can think of a thread as a separate process that happens to share all the variables and memory with the rest of your program. One of the arguments of `pthread_create()` will specify what this thread should do, specifically what function it should run.

The prototype of `pthread_create()` is:

```
1  int pthread_create(
2    pthread_t *thread_id,       // ID number for thread
3    const pthread_attr_t *attr, // controls thread attributes
4    void *(*function)(void *),  // function to be executed
5    void *arg                   // argument of function
6  );
```

The first argument is a pointer to the thread identifier (which is simply a number but we do not actually care about the details of how the ID is specified). This ID is set by `pthread_create()`, i.e., one would typically be interested in the returned value—it is not something that is set prior to `pthread_create()` being called.

The second argument is a pointer to a variable that controls the attributes of the thread. In this case, the value of this variable is established prior to the call of the function. This pointer can be set to `NULL` in which case the thread is created with the default attributes. Attributes control things like the "joinability" of the thread and the scheduling of threads. Typically one can simply use the default settings. The `pthread_t` and `pthread_attr_t` data types are defined in `pthread.h`.

The third and fourth arguments specify what function the thread should call and what argument should be passed to the function. Notice that the prototype says the function takes a void pointer as an argument and returns a void pointer. Keep in mind that "void" pointers are, in fact, simply generic pointers to memory. We can typecast these pointers to what they actually are. Thus, in practice, it would be perfectly acceptable for the function to take, for example, an argument of a pointer to a structure and return a pointer to a double. One would merely have to do the appropriate typecasting. If the function does not take an argument, the fourth argument of `pthread_create()` is set to `NULL`.

Once a new thread is created using `pthread_create()`, the program continues execution at the next command—the program does not wait for the thread to complete whatever the thread has been assigned to do. The function `pthread_join()` is used to block further execution of commands until the specified thread has completed. `pthread_join()` can also be used to access the return-value of the function that was run in a thread. The prototype of `pthread_join()` is:

```
1  int pthread_join(
2    pthread_t thread_id, // ID of thread to "join"
3    void **value_pntr    // address of function's return value
4  );
```

13.2 Thread Examples

To demonstrate the use of pthreads, let us first consider a standard serial implementation of a program where first one function is called and then another is called. The program is shown in Program 13.1.

Program 13.1 `serial-example.c`: Standard serial implementation of a program where first one function is called and then another. (These function are merely intended to perform a lengthy calculation. They do not do anything particularly useful.)

```
1  /* serial (i.e., non-threaded) implementation */
2  #include <stdio.h>
3
4  void func1();
5  void func2();
6
7  double a, b;  // global variables
8
9  int main() {
10
11   func1();  // call first function
12   func2();  // call second function
13
14   printf("a: %f\n", a);
15   printf("b: %f\n", b);
16
17   return 0;
18 }
19
20 /* do some lengthy calculation which sets the value of the the global
21    variable "a"*/
22 void func1() {
23   int i, j;
24
25   for (j=0;j<4000;j++)
26     for (i=0;i<1000000;i++)
27       a = 3.1456*j+i;
28
29   return;
30 }
31
32 /* do another lengthy calculation which happens to be the same as
33    done by func1() except here the value of global variable "b" is set
34    */
35 void func2() {
36   int i, j;
```

```
37
38    for (j=0;j<4000;j++)
39      for (i=0;i<1000000;i++)
40        b = 3.1456*j+i;
41
42    return;
43  }
```

In this program the functions `func1()` and `func2()` do not take any arguments nor do they explicitly return any values. Instead, the global variables `a` and `b` are used to communicate values back to the main function. Neither `func1()` nor `func2()` are intended to do anything useful. There are merely used to perform some lengthy calculation. Assuming the executable version of this program is named `serial-example`, the execution time can be obtained, on a typical UNIX-based system, by issuing the command "`time serial-example`".

Now, let us consider a threaded implementation of this same program. The appropriate code is shown in Program 13.2.

Program 13.2 `threads-example1.c`: A threaded implementation of the program shown in Program 13.1.

```
1  /* threaded implementation */
2  #include <stdio.h>
3  #include <pthread.h>
4
5  void *func1();
6  void *func2();
7
8  double a, b;
9
10 int main() {
11   pthread_t thread1, thread2;  // ID's for threads
12
13   /* create threads which run in parallel -- one for each function */
14   pthread_create(&thread1, NULL, func1, NULL);
15   pthread_create(&thread2, NULL, func2, NULL);
16
17   /* wait for first thread to complete */
18   pthread_join(thread1,NULL);
19   printf("a: %f\n", a);
20
21   /* wait for second thread to complete */
22   pthread_join(thread2,NULL);
23   printf("b: %f\n", b);
24
25   return 0;
```

```
26 }
27
28 void *func1() {
29   int i, j;
30
31   for (j=0;j<4000;j++)
32     for (i=0;i<1000000;i++)
33       a = 3.1456*j+i;
34
35   return NULL;
36 }
37
38 void *func2() {
39   int i, j;
40
41   for (j=0;j<4000;j++)
42     for (i=0;i<1000000;i++)
43       b = 3.1456*j+i;
44
45   return NULL;
46 }
```

In Program 13.2 `func1()` and `func2()` are slightly different from the functions of the same name used in 13.1. In both these programs these functions perform the same calculations, but in 13.1 these functions returned nothing. However `pthread_create()` assumes the function returns a void pointer (i.e., a generic pointer to memory). Since in this example these functions do not need to return anything, they simply return `NULL` (which is effectively zero).

If Program 13.2 is run on a computer that has two (or more) processors, one should observe that the execution time (as measured by a "wall clock") is about half of what it was for Program 13.2. Again, assuming the executable version of Program 13.2 is named `threads-example1`, the execution time can be obtained by issuing the command "`time threads-example1`". This timing command will typically return three values: the "wall-clock" time (the actual time that elapsed from the start to the completion of the program), the CPU time (the sum of time spent by all processors used to run the program), and system time (time used by the operating system to run things necessary for your program to run, but not directly associated with your program). You should observe that ultimately nearly the same amount of CPU time was used by both the threaded and serial programs but the threaded program required about half the wall-clock time. In the case of the second program two processors were working simultaneously and hence the wall-clock time was half as much, or nearly so. In fact, there is slightly more computation involved in the threaded program than the serial program since there is some computational overhead associated with the threads.

Let us now modify the first function so that it returns a value, specifically a pointer to a double where we simply store an arbitrary number (in this case 10.0). The appropriate code is shown in Program 13.3.

Program 13.3 `threads-example2.c`: Modified version of Program 13.2 where now `func1()` has a return value.

```
1  /* threaded implementation -- returning a value */
2  #include <stdio.h>
3  #include <stdlib.h> // needed for malloc()
4  #include <pthread.h>
5
6  double *func1();  // now returns a pointer to a double
7  void *func2();
8
9  double a, b;
10
11 int main() {
12   double *c; // used for return value from func1
13   pthread_t thread1, thread2; // ID's for threads
14
15   // typecast the return value of func1 to a void pointer
16   pthread_create(&thread1, NULL, (void *)func1, NULL);
17   pthread_create(&thread2, NULL, func2, NULL);
18
19   // typecast the address of c to a void pointer to a pointer
20   pthread_join(thread1,(void **)&c);
21   printf("a,c: %f %f\n", a, *c);
22
23   pthread_join(thread2,NULL);
24   printf("b: %f\n", b);
25
26   return 0;
27 }
28
29 double *func1() {
30   int i, j;
31
32   double *c;  // c is a pointer to a double
33
34   // allocate space to store a double
35   c=(double *)malloc(sizeof(double));
36   *c = 10.0;
37
38   for (j=0;j<4000;j++)
39     for (i=0;i<1000000;i++)
40       a = 3.1456*j+i;
41
42   return c;
43 }
44
```

```
45 void *func2() {
46   int i, j;
47
48   for (j=0;j<4000;j++)
49     for (i=0;i<1000000;i++)
50       b = 3.1456*j+i;
51
52   return NULL;
53 }
```

Note that in this new version of `func1()` we declare `c` to be a pointer to a double and then, in line 35, allocate space where the double can be stored and then, finally, store 10.0 at this location. This is rather complicated and it might seem that a simpler approach would be merely to declare `c` to be a double and then return the address of `c`, i.e., end the function with `return &c`. Unfortunately this would not work. The problem with that approach is that declaring `c` to be a double would make it a local variable (one only known to `func1()`) whose memory would disappear when the function returned.

The second argument of `pthread_join()` in line 20 provides the pointer to the return value of the function that was executed by the thread. Since `c` by itself is a pointer to a double, `&c` is a pointer to a pointer to a double, i.e., of type `(double **)`. However, `pthread_join()` assumes the second argument is a void pointer to a pointer and hence a typecast is used to keep the compiler from complaining.

In the next example, shown in Program 13.4, `func1()` and `func2()` are modified so that they each take an argument. These arguments are the double variables `e` and `d` that are set in `main()`.

Program 13.4 `threads-example3.c`: Functions `func1()` and `func2()` have been modified so that they now take arguments.

```
1  /* threaded implementation -- passing arguments and
2     returning a value */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <pthread.h>
6
7  double *func1(double *);
8  void *func2(double *);
9
10 double a, b;
11
12 int main() {
13   double *c; // used for return value from func1
14   double d=3.0, e=2.0; // arguments passed to functions
15   pthread_t thread1, thread2; // ID's for threads
```

```
16
17    pthread_create(&thread1, NULL, (void *)func1, (void *)&d);
18    pthread_create(&thread2, NULL, (void *)func2, (void *)&e);
19
20    pthread_join(thread1,(void **)&c);
21    printf("a,c: %f %f\n", a, *c);
22
23    pthread_join(thread2,NULL);
24    printf("b: %f\n", b);
25
26    return 0;
27  }
28
29  double *func1(double *arg) {
30    int i, j;
31
32    double *c;
33
34    c=(double *)malloc(sizeof(double));
35    *c = 10.0;
36
37    for (j=0;j<4000;j++)
38      for (i=0;i<1000000;i++)
39        a = (*arg)*j+i;
40
41    return c;
42  }
43
44  void *func2(double *arg) {
45    int i, j;
46
47    for (j=0;j<4000;j++)
48      for (i=0;i<1000000;i++)
49        b = (*arg)*j+i;
50
51    return NULL;
52  }
```

In all these examples `func1()` and `func2()` have performed essentially the same computation. The only reason there were two separate functions is that `func1()` set the global variable `a` while `func2()` set the global variable `b`. However, knowing that we can both pass arguments and obtain return values, it is possible to have a single function in our program. It can be called multiple times and simultaneously. Provided the function does not use global variables, the different calls will not interfere with each other.

A program that uses a single function to accomplish what the previous programs used two function for is shown in Program 13.5.

Program 13.5 `threads-example4.c`: The global variables have been removed and a single function `func()` is called twice. The function `func()` and `main()` communicate by passing arguments and checking returns values (instead of via global variables).

```
/* threaded implementation -- passing an argument and checking
   return the value from a single function */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

double *func(double *);

int main() {
  double *a, *b; // used for return values
  double d=3.0, e=2.0;
  pthread_t thread1, thread2; // ID's for threads

  pthread_create(&thread1, NULL, (void *)func, (void *)&d);
  pthread_create(&thread2, NULL, (void *)func, (void *)&e);

  pthread_join(thread1,(void **)&a);
  printf("a: %f\n", *a);

  pthread_join(thread2,(void **)&b);
  printf("b: %f\n", *b);

  return 0;
}

double *func(double *arg) {
  int i, j;

  double *a;

  a=(double *)malloc(sizeof(double));

  for (j=0;j<4000;j++)
    for (i=0;i<1000000;i++)
      *a = (*arg)*j+i;

  return a;
}
```

Threads provide a simple way to obtain parallelization. However, one may find that in practice they do not provide the benefits one might expect when applied to FDTD programs. FDTD is

both computational expensive and memory-bandwidth intensive. A great deal of data must be passed between memory and the CPU. Often the bottleneck is not the CPU but rather the speed of the "bus" that carries data between memory and the CPU. Multi-processor machines do not have multiple memory busses. Thus, splitting an FDTD computation between multiple CPU's on the same computer will have those CPU's all requesting memory from a bus that is already acting at full capacity. These CPU's will have to wait on the arrival of the requested memory. Therefore, in practice when using threaded code with N threads on a computer with N processors, one is unlikely to see a computation-time reduction that is anywhere close to the hypothetical maximum reduction of $1/N$.

13.3 Message Passing Interface

The message passing interface (MPI) is a standardized protocol, or set of protocols, which have been implemented on a wide range of platforms. MPI facilitates the communication between processes whether they are running on a single host or multiple hosts. As with pthreads, MPI provides a large number of functions. These functions allow the user to control many aspects of the communication or they greatly simplify what would otherwise be quite cumbersome tasks (such as the efficient distribution of data to a large number of hosts). Despite the large number of MPI functions, just six are needed to begin exploiting the benefits of parallelization.

Before considering those six functions, it needs to be said that one must have the supporting MPI framework installed on each of the hosts to be used. Different implementations of the MPI protocol (or the MPI 2 protocol) are available from the Web. For example, LAM MPI is available from `www.lam-mpi.org` but it is no longer being actively developed. Instead, several MPI-developers have joined together to work on OpenMPI which is available from `www.open-mpi.org`. Alternatively, MPICH2 is available from `www.mcs.anl.gov/research/projects/mpich2`. Installation of these packages is relatively trivial (at least that is the case when using Linux or Mac OS X), but some of the details associated with getting jobs to run can be somewhat complicated (for example, ensuring that access is available to remote machines without requiring explicit typing of a password).

There is a great deal of MPI documentation available from the Web (there are also a few books written on the subject). A good resource for getting started is `computing.llnl.gov/tutorials/mpi/`. In fact, Lawrence Livermore has many useful pages related to threads, MPI, and other aspects of high-performance and parallel processing. You can find the material by going to `computing.llnl.gov` and following the link to "Training."

Returning to the six functions needed to use MPI in a meaningful way, two of them concern initializing and closing the MPI set-up, two deal with sending and receiving information, and two deal with determining the number of processes and which particular process number is associated with the given invocation. Programs which use MPI must include the header file `mpi.h`. Before any other MPI functions are called, the function `MPI_Init()` must be called. The last MPI function called should be `MPI_Finalize()`. Thus, a valid, but useless, MPI program is shown in Program 13.6.

Program 13.6 `useless-mpi.c`: A trivial, but valid, MPI program.

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

  MPI_Init(&argc,&argv);

  printf("I don't do anything useful yet.\n");

  MPI_Finalize();

  return 0;
}
```

When an MPI program is run, each process runs the same program. In this case, there is nothing to distinguish between the processes. They will all generate the same line of output. If there were 100 processes, you would see 100 lines of "`I don't do anything useful yet.`"

13.4 Open MPI Basics

At this point we need to ask the question: What does it mean to run an MPI program? Ultimately many copies of the same program are run. Each copy may reside on a different computer and, in fact, multiple copies may run on the same computer. The details concerning how one gets these copies to run are somewhat dependent on the MPI package one uses. Here we will briefly describe the steps associated with the Open MPI package.

When Open MPI is installed, several executable files will be placed on your system(s), e.g., `mpicc`, `mpiexe`, `mpirun`, etc. On most systems by default these files will be installed in the directory `/usr/local/bin` (but one can specify that the files should be installed elsewhere if so desired). One must ensure that the directory where these excutables reside is in the search path.

When using MPI it is usually necessary to compile the source code in a special way. Instead of using the `gcc` compiler on UNIX/Linux machines, one would use `mpicc` (`mpicc` is merely a wrapper that ultimately calls the underlying compiler that one would have used normally). So, for example, to compile the MPI program given above, one would issue a command such as

```
mpicc -Wall -O useless-mpi.c -o useless-mpi
```

One can now run the executable file `useless-mpi`. The command to do this is either `mpirun` or `mpiexec` (these commands are synonymous). There are numerous arguments that can be specified with the most important being the number of processes. The following command says to run four copies of `useless-mpi`:

```
mpiexec -np 4 useless-mpi
```

But where, precisely, is this run? In this case four copies of the program are run on the local host. That is not precisely what we want—we are interested in distributing the job to different machines. There are multiple ways in which one can excercise control of the running of MPI programs and we will explore just a few.

First, let us assume a "multicomputer" consists of five nodes with names `node01`, `node02`, `node03`, `node04`, and `node05`. To make things more interesting, let us further assume that `node01` is one particular brand of computer and the other nodes are a different brand (e.g., perhaps `node01` is an Intel-based machine while the other nodes are PowerPC-based machines). Additionally assume that `node01` has four processors while each of the other nodes has two processors.

We can specify some of this information in a "hostfile." For now, let us exclude `node01` since it is a different architecture. A hostfile that describes a multicomputer consisting of the remaining four nodes might be

```
node05 slots=2
node04 slots=2
node03 slots=2
node02 slots=2
```

Let us assume this information is stored in the file `my_hostfile`. The `slots` are the number of processors on a particular machine. If one does not specify the number of slots, it is assumed to be one.

Let us further assume the executable `useless-mpi` exists in a director called `~/Ompi` (where the tilde is recognized as a shorthand for the user's home directory on a UNIX/Linux machine). If all the computers mount the same file structure, this may actually be the exact same directory that all the machines are sharing. In that case there would only be one copy of `useless-mpi`. Alternatively, each of the computers may have their own local copy of a directory named `~/Ompi`. In that case there would have to be a local copy of the excutable file `useless-mpi` present on each of the individual computers.

One could now run eight copies of the program by issuing the following command:

```
mpirun -np 8 -hostfile my_hostfile ~/Ompi/useless-mpi
```

This command could be issued from any of the nodes. Note that the number of processes does not have to match the number of slots. The following command will launch 12 copies of the program

```
mpirun -np 12 -hostfile my_hostfile ~/Ompi/useless-mpi
```

However, it will generally be best if one can match the job to the physical configuration of the multicomputer, i.e., one job per "slot."

In order to incorporate `node01` into the multicomputer, things become slightly more complicated because executables compiled for `node01` will not run on the other nodes and vice versa. Thus one must compile separate versions of the program on the different machines. Let's assume that was done and on each of the nodes a copy of `useless-mpi` was place in the local directory `/tmp` (i.e., there is a copy of this directory and this executable on each of the nodes). The hostfile `my_hostfile` could then be changed to

```
node05 slots=2
node04 slots=2
node03 slots=2
node02 slots=2
node01 slots=4
```

Note that there are four slots specified for `node01` instead of two. The command to run 12 copies of the program would now be

```
mpirun -np 12 -hostfile my_hostfile /tmp/useless-mpi
```

By introducing more arguments to the command line, one can exercise more fine-grained control of the executation of the program. Let us again assume that there is one common directory `~/Ompi` that all the machines share. Let us further assume two version of `useless-mpi` have been compiled: one for PowerPC-based machines called `useless-mpi-ppc` and one for Intel-based machines called `useless-mpi-intel`. We can use a command that does away with the hostfile and instead provide all the details explicitly:

```
mpirun -host node05,node04,node03,node02 \
       -np 8 ~/Ompi/useless-mpi-ppc \
       -host node01 -np 4 ~/Ompi/useless-mpi-intel
```

Note that the backslashes here are quoting the end of the line. This command can be given on a single line or can be given on multiple lines, as shown here, if one "quotes" the carriage return.

13.5 Rank and Size

To do more meaningful tasks, it is typically necessary for each processor to know how many total processes there are and which process number is assigned to a particular invocation. In this way, each processor can do something different based on its process number. In MPI the process number is known as the rank. The number of processes can be determined with the function `MPI_Comm_size()` and the rank can be determined with `MPI_Comm_rank()`. The code shown in Program 13.7 is a slight modification of the previous program that now incorporates these functions.

Program 13.7 `find-rank.c`: An MPI program where each process can determine the total number of processes and its individual rank (i.e., process number).

```
1 #include <mpi.h>
2 #include <stdio.h>
3
4 int main(int argc, char *argv[]) {
5   int rank, size;
6
7   MPI_Init(&argc,&argv);
```

```
8
9     MPI_Comm_size(MPI_COMM_WORLD, &size);
10    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11    printf("I have rank %d out of %d total.\n",rank,size);
12
13    MPI_Finalize();
14
15    return 0;
16  }
```

Assume this is run with four total processes. The output will be similar to this:

```
I have rank 1 out of 4 total.
I have rank 0 out of 4 total.
I have rank 2 out of 4 total.
I have rank 3 out of 4 total.
```

Note that the `size` is 4, but `rank` ranges between 0 and 3 (i.e., `size` $-$ 1). Also note that there is no guarantee that the processes will report in rank order.

The argument `MPI_COMM_WORLD` is known as an MPI communicator. A communicator essentially specifies the processes which are grouped together. One can create different communicators, i.e., group different sets of processes together, and this can simplify handling certain tasks for a particular problem. However, we will simply use `MPI_COMM_WORLD` which specifies all the processes.

13.6 Communicating Between Processes

To communicate between processes we can use the commands `MPI_Send()` and `MPI_Recv()`. `MPI_Send()` has arguments of the form:

```
MPI_Send(&buffer, // address where data stored
         count,   // number of items to send
         type,    // type of data to send
         dest,    // rank of destination process
         tag,     // programmer-specified ID
         comm);   // MPI communicator
```

where `buffer` is an address where the data to be sent is stored (for example, the address of the start of an array), `count` is the number of elements or items to be sent, `type` is the type of data to be sent, `dest` is the rank of the process to which this information is being sent, `tag` is a programmer-specified number to identify this data, and `comm` is an MPI communicator (which we will leave as `MPI_COMM_WORLD`). The `type` is similar to the standard C data types, but it is specified using MPI designations. Some of those are: `MPI_INT`, `MPI_FLOAT`, and `MPI_DOUBLE`, corresponding to the C data types of int, float, and double (other types, some of which are specific to MPI, such as `MPI_BYTE` and `MPI_PACKED`, exist too).

`MPI_Recv()` has arguments of the form:

```
MPI_Recv(&buffer,count,type,source,tag,comm,&status);
```

In this case `buffer` is the address where the received data is to be stored. The meaning of `count`, `type`, `tag`, and `comm` are unchanged from before. `source` is the rank of the process sending the data. The `status` is a pointer to a structure, specifically an `MPI_status` structure which is specified in `mpi.h`. This structure contains the rank of the source and the `tag` number.

Program 13.8 demonstrates the use of `MPI_Send()` and `MPI_Recv()`. Here the process with rank 0 serves as the master process. It collects input from the user which will subsequently be sent to the other processes. Specifically, the parent process prompts the user for as many values (doubles) as there are number of processes minus one. The master process then sends one number to each of the other processes. These processes do a calculation based on the number they receive and then send the result back to the master. The master prints this received data and then the program terminates.

Program 13.8 `sendrecv.c`: An MPI program that sends information back and forth between a master process and slave processes.

```
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  int i, rank, size, tag_out=10, tag_in=11;
  MPI_Status status;

  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank==0) {
    /* "master" process collects and distributes input */
    double *a, *b;

    /* allocate space for input and result */
    a=malloc((size-1)*sizeof(double));
    b=malloc((size-1)*sizeof(double));

    /* prompt user for input */
    printf("Enter %d numbers: ",size-1);
    for (i=0; i<size-1; i++)
      scanf("%lf",a+i);

    /* send values to other processes */
    for (i=0; i<size-1; i++)
      MPI_Send(a+i,1,MPI_DOUBLE,i+1,tag_out,MPI_COMM_WORLD);
```

```
30
31      /* receive results calculated by other process */
32      for (i=0; i<size-1; i++)
33        MPI_Recv(b+i,1,MPI_DOUBLE,i+1,tag_in,MPI_COMM_WORLD,&status);
34
35      for (i=0; i<size-1; i++)
36        printf("%f\n",b[i]);
37
38    } else {
39      /* "slave" process */
40      int j;
41      double c, d;
42
43      /* receive input from the master process */
44      MPI_Recv(&c,1,MPI_DOUBLE,0,tag_out,MPI_COMM_WORLD,&status);
45
46      /* do some silly number crunching */
47      for (j=0;j<4000;j++)
48        for (i=0;i<100000;i++)
49          d = c*j+i;
50
51      /* send the result back to master */
52      MPI_Send(&d,1,MPI_DOUBLE,0,tag_in,MPI_COMM_WORLD);
53    }
54
55    MPI_Finalize();
56
57    return 0;
58  }
```

The six commands covered so far are sufficient to parallelize any number of problems. However, there is some computational overhead associated with parallelizing the code. Additionally, there is often a significant cost associated with communication between processes, especially if those processes are running on different hosts and the network linking those hosts is slow.

The functions `MPI_Send()` and `MPI_Recv()` are blocking commands. They do not return until they have accomplished the requested send or receive. In some cases, especially if there is a large amount of data to transmit, this can be costly. There are also nonblocking or "immediate" versions of these functions. For these functions control is returned to the calling function without a guarantee of the send or receive having been accomplished. In this way the program can continue some other useful task while the communication is taking place. When one must ensure that the communication is finished, the function `MPI_Wait()` provides a blocking mechanism that suspends execution until the specified communication is completed. The immediate send and receive functions are of the form:

```
MPI_Isend(&buffer,count,type,dest,tag,comm,&request);
MPI_Irecv(&buffer,count,type,source,tag,comm,&request);
```

The arguments to these functions are the same as the blocking version except the final argument is now a pointer to an `MPI_Request` structure instead of an `MPI_Status`. The wait command has the following form:

```
MPI_Wait(&request,&status);
```

Note that the communication for which the waiting is being done is specified by the "`request`," not the "`status`." So, if there are multiple transmissions which are being done asynchronously, one may have to create an array of `MPI_Request`'s. If one is not concerned with the status of the transmissions, one does not have to define a separate status for each transmission.

The code shown in Program 13.9 illustrates the use of non-blocking send and receive. In this case the master process sends the numbers to the other processes via `MPI_Isend()`. However, the master does not bother to ensure that the send was performed. Instead, the master will ultimately wait for the other process to communicate the result back. The fact that the other processes are sending information back serves as confirmation that the data was sent from the master. After sending the data, the master process then calls `MPI_Irecv()`. There is one call for each of the "slave" processes. After calling these functions, `MPI_Wait()` is used to ensure the data has been received before printing the results. The code associated with the slave processes is unchanged from before.

Program 13.9 `nonblocking.c`: An MPI program that uses non-blocking sends and receives.

```
1  #include <mpi.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  int main(int argc, char *argv[]) {
6    int i, rank, size, tag_out=10, tag_in=11;
7    MPI_Status status;
8    MPI_Request *request_snd, *request_rcv;
9
10   MPI_Init(&argc,&argv);
11
12   MPI_Comm_size(MPI_COMM_WORLD, &size);
13   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14
15   if (rank==0) {
16     /* "master" process collects and distributes input */
17     double *a, *b;
18
19     /* allocate space for input and result */
20     a=malloc((size-1)*sizeof(double));
21     b=malloc((size-1)*sizeof(double));
22
23     /* allocate space for the send and receive requests */
24     request_snd=malloc((size-1)*sizeof(MPI_Request));
```

```
25      request_rcv=malloc((size-1)*sizeof(MPI_Request));
26
27      /* prompt user for input */
28      printf("Enter %d numbers: ",size-1);
29      for (i=0; i<size-1; i++)
30        scanf("%lf",a+i);
31
32      /* non-blocking send of values to other processes */
33      for (i=0; i<size-1; i++)
34        MPI_Isend(a+i,1,MPI_DOUBLE,i+1,tag_out,MPI_COMM_WORLD,request_snd+i);
35
36      /* non-blocking reception of results calculated by other process */
37      for (i=0; i<size-1; i++)
38        MPI_Irecv(b+i,1,MPI_DOUBLE,i+1,tag_in,MPI_COMM_WORLD,request_rcv+i);
39
40      /* wait until we have received all the results */
41      for (i=0; i<size-1; i++)
42        MPI_Wait(request_rcv+i,&status);
43
44      for (i=0; i<size-1; i++)
45        printf("%f\n",b[i]);
46
47    } else {
48      /* "slave" process */
49      int j;
50      double c, d;
51
52      /* receive input from the master process */
53      MPI_Recv(&c,1,MPI_DOUBLE,0,tag_out,MPI_COMM_WORLD,&status);
54
55      /* do some silly number crunching */
56      for (j=0;j<4000;j++)
57        for (i=0;i<100000;i++)
58          d = c*j+i;
59
60      /* send the result back to master */
61      MPI_Send(&d,1,MPI_DOUBLE,0,tag_in,MPI_COMM_WORLD);
62    }
63
64    MPI_Finalize();
65
66    return 0;
67  }
```

Compared to the previous version of this program, this version runs over 30 percent faster on a dual-processor G5 when using five processes. (By "over 30 percent faster" is meant that if

the execution time for the previous code is normalized to 1.0, the execution time using the non-blocking calls is approximately 0.68.)

Chapter 14

Near-to-Far-Field Transformation

14.1 Introduction

As we have seen, the FDTD method provides the fields throughout some finite region of space, i.e., the fields throughout the computational domain. However, in practice, we are often interested in the fields far away from the region we have modeled. For example, an FDTD implementation may have modeled an antenna or some scatterer. But, the fields in the immediate vicinity of that antenna or scatterer may not be the primary concern. Rather, the distant or "far" fields may be the primary concern. In this chapter we show how the near fields, which are essentially the fields within the FDTD grid, can be used to obtain the far fields. We start with a brief review of the underlying theory that pertains in the continuous world and then discuss the implementation details for the FDTD method.

14.2 The Equivalence Principle

Recall the boundary conditions that pertain to the electric and magnetic fields tangential to an interface:

$$\hat{\mathbf{n}}' \times (\mathbf{E}_1 - \mathbf{E}_2) = -\mathbf{M}_s, \tag{14.1}$$

$$\hat{\mathbf{n}}' \times (\mathbf{H}_1 - \mathbf{H}_2) = \mathbf{J}_s, \tag{14.2}$$

where $\hat{\mathbf{n}}'$ is normal to the interface, pointing toward region 1. The subscript 1 indicates the fields immediately adjacent to one side of the interface and the subscript 2 indicates the fields just on the other side of the interface. The "interface" can either be a physical boundary between two media or a fictitious boundary with the same medium to either side. The current $\mathbf{M}_s$ is a magnetic surface current, i.e., a current that only flows tangential to the interface. In practice there is no magnetic charge and thus no magnetic current. Therefore (14.1) states that the tangential components of $\mathbf{E}$ must be continuous across the boundary. However, in theory, we can imagine a scenario where the tangential fields are discontinuous. If this were the case, the magnetic current $\mathbf{M}_s$ must be non-zero to account for this discontinuity. In a little while we will see why it is convenient to envision such a scenario. The current $\mathbf{J}_s$ in (14.2) is the usual electric surface current.

As depicted in Fig. 14.1(a), consider a space in which there is a source or scatterer that radiates (or scatters) some fields. We can define a fictitious boundary that surrounds this source or scatterer. Let us then imagine that the fields exterior to this boundary are unchanged but the fields interior to the boundary are set to zero as depicted in Fig. 14.1(b). By setting the fields interior to the boundary to zero, we will create discontinuities in the tangential components on either side of the fictitious boundary. These discontinuities are perfectly fine provided we account for them by having the appropriate surface currents flow over the boundary. These currents are given by (14.1) and (14.2) where the fields in region 2 are now zero. Thus,

$$\mathbf{M}_s = -\hat{\mathbf{n}}' \times \mathbf{E}_1, \tag{14.3}$$

$$\mathbf{J}_s = \hat{\mathbf{n}}' \times \mathbf{H}_1. \tag{14.4}$$

As you may recall and as will be discussed in further detail below, it is fairly simple to find the fields radiated by a current (whether electric or magnetic) when that current is radiating in a homogeneous medium. Unfortunately, as shown in Fig. 14.1(b), the surface currents are not radiating in a homogeneous medium. But, in fact, since the fields within the fictitious boundary are zero, we can place anything, or nothing, within the boundary and that will have no effect on the fields exterior to the boundary. So, let us maintain the same surface currents but discard any inhomogeneity that were within the boundary. That leaves a homogeneous region as depicted in Fig. 14.1(c) and it is fairly straightforward to find these radiated fields.

14.3 Vector Potentials

Before proceeding further, let us briefly review vector potentials. First, consider the case (which corresponds to the physical world) where there is no magnetic charge—electric currents can flow but magnetic currents cannot. Thus,

$$\nabla \cdot \mathbf{B}_A = \nabla \cdot \mu \mathbf{H}_A = 0 \tag{14.5}$$

where the subscript A indicates we are considering the case of no magnetic charge. There is a vector identity that the divergence of the curl of any vector field is identically zero. Therefore (14.5) will automatically be satisfied if we write

$$\mathbf{H}_A = \frac{1}{\mu} \nabla \times \mathbf{A} \tag{14.6}$$

where $\mathbf{A}$ is a yet-to-be-determined field known as the magnetic vector potential. Now, using Faraday's law we obtain

$$\nabla \times \mathbf{E}_A = -j\omega\mu \mathbf{H}_A = -j\omega \nabla \times \mathbf{A}. \tag{14.7}$$

Using the terms on the left and the right and regrouping yields

$$\nabla \times (\mathbf{E}_A + j\omega \mathbf{A}) = 0. \tag{14.8}$$

The curl of the gradient of any function is identically zero. Thus we can set the term in parentheses equal to the (negative of the) gradient of some unknown scalar electric potential function Φ_e and in this way (14.8) will automatically be satisfied. Therefore we have

$$\mathbf{E}_A + j\omega \mathbf{A} = -\nabla \Phi_e \tag{14.9}$$

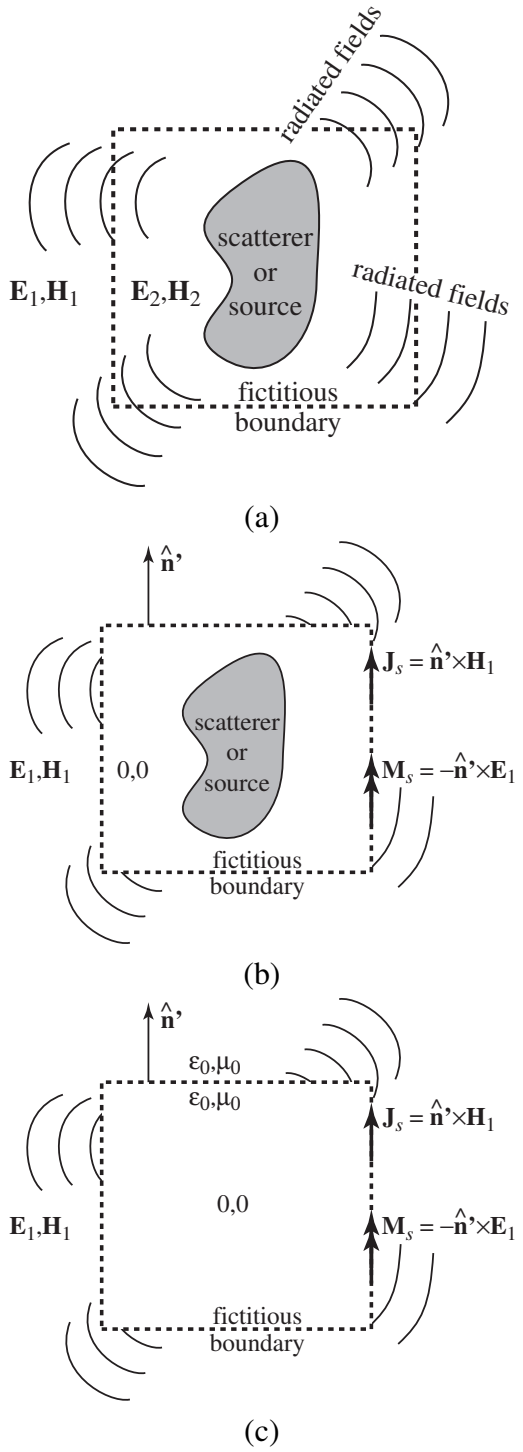


Figure 14.1: (a) A space containing a source or scatterer that is surrounded by a fictitious boundary which is indicated by the dashed line. The fields are continuous across this boundary. (b) The fields are set to zero within the boundary. Surface currents must be used to account for the discontinuity across the boundary. (c) Since the fields are zero within the boundary, any inhomogeneities within the boundary can be discarded.

or, after rearranging,

$$\mathbf{E}_A = -j\omega\mathbf{A} - \nabla\Phi_e. \tag{14.10}$$

Using the remaining curl equation, Ampere's law, we can write

$$\nabla \times \mathbf{H}_A = \mathbf{J} + j\omega\epsilon\mathbf{E}_A \tag{14.11}$$

$$\nabla \times \frac{1}{\mu}\nabla \times \mathbf{A} = \mathbf{J} + j\omega\epsilon(-j\omega\mathbf{A} - \nabla\Phi_e) \tag{14.12}$$

Multiplying through by μ and expanding the curl operations yields

$$\nabla(\nabla \cdot \mathbf{A}) - \nabla^2\mathbf{A} = \mu\mathbf{J} + \omega^2\mu\epsilon\mathbf{A} - j\omega\mu\epsilon\nabla\Phi_e. \tag{14.13}$$

Regrouping terms yields

$$\nabla^2\mathbf{A} + \omega^2\mu\epsilon\mathbf{A} = -\mu\mathbf{J} + \nabla(\nabla \cdot \mathbf{A} + j\omega\mu\epsilon\Phi_e). \tag{14.14}$$

So far we have said what the curl of $\mathbf{A}$ must be, but that does not fully describe the field. To fully describe a vector field one must specify the curl, the divergence, and the value at a point (which we will ultimately assume is zero at a infinite distance from the origin). We are free to make the divergence of $\mathbf{A}$ any convenient value. Let us use the "Lorentz gauge" of

$$\nabla \cdot \mathbf{A} = -j\omega\mu\epsilon\Phi_e. \tag{14.15}$$

By doing this, (14.14) reduces to

$$\nabla^2\mathbf{A} + k^2\mathbf{A} = -\mu\mathbf{J}. \tag{14.16}$$

where $k = \omega\sqrt{\mu\epsilon}$.

Distinct from the scenario described above, let us imagine a situation where there is no free electric charge. Magnetic currents can flow, but electric currents cannot. Thus the divergence of the electric flux density is

$$\nabla \cdot \mathbf{D}_F = \nabla \cdot \epsilon\mathbf{E}_F = 0 \tag{14.17}$$

where here the subscript F is used to indicate the case of no electric charge. Again, this equation will be satisfied automatically if we represent the electric field as the curl of some potential function $\mathbf{F}$. To this end we write

$$\mathbf{E}_F = -\frac{1}{\epsilon}\nabla \times \mathbf{F} \tag{14.18}$$

where $\mathbf{F}$ is known as the electric vector potential.

Following steps similar to the ones we used to obtain (14.16), one can obtain the differential equation that governs $\mathbf{F}$, namely,

$$\nabla^2\mathbf{F} + k^2\mathbf{F} = -\epsilon\mathbf{M}. \tag{14.19}$$

Thus both $\mathbf{A}$ and $\mathbf{F}$ are governed by the wave equation. We see that the source of $\mathbf{A}$, i.e., the forcing function that creates $\mathbf{A}$ is the electric current $\mathbf{J}$. Similarly, the source of $\mathbf{F}$ is the magnetic current $\mathbf{M}$. (We have not yet restricted these currents to be surface currents. At this point they can be any current distribution, whether distributed throughout a volume, over a surface, or along a line.)

Note that both the Laplacian (∇^2) and the constant k^2 that appear in (14.16) and (14.19) are scalar operators. They do not change the orientation of a vector. Thus, the x component of $\mathbf{J}$ gives

rise to the x component of $\mathbf{A}$, the y component of $\mathbf{M}$ gives rise to the y component of $\mathbf{F}$, and so on. In this way, (14.16) and (14.19) could each be broken into their three Cartesian components and we would be left with six scalar equations.

These equations have relatively straightforward solutions. Let us consider a slightly simplified problem, the solution of which can easily be extended to the full general problem. Consider the case of an incremental current of length $d\ell$ that is located at the origin and oriented in the z direction. In this case (14.16) reduces to

$$\nabla^2 A_z(\mathbf{r}) + k^2 A_z(\mathbf{r}) = -\mu I d\ell \delta(\mathbf{r}) \tag{14.20}$$

where I is the amount of current and $\delta(\mathbf{r})$ is the 3D Dirac delta function. The Dirac delta function is zero expect when its argument is zero. For an argument of zero, $\delta(\mathbf{r})$ is singular, i.e., infinite. However, this singularity is integrable. A volume integral of any region of space that includes the Dirac delta function at the origin (i.e., $\mathbf{r} = 0$) will yield unit volume. For any observation point other than the origin, (14.20) can be written

$$\nabla^2 A_z(\mathbf{r}) + k^2 A_z(\mathbf{r}) = 0 \qquad \mathbf{r} \neq 0. \tag{14.21}$$

It is rather easy to show that a general solution to this is

$$A_z(\mathbf{r}) = C_1 \frac{e^{-jkr}}{r} + C_2 \frac{e^{jkr}}{r}. \tag{14.22}$$

We discard the second term on the right-hand side since that represents a spherical wave propagating in toward the origin. Thus we are left with

$$A_z(\mathbf{r}) = C_1 \frac{e^{-jkr}}{r} \tag{14.23}$$

where we must now determine the constant C_1 based on the "driving function" on the right side of (14.20).

To obtain C_1, we integrate both side of (14.20) over a small spherical volume of radius r_0 and take the limit as r_0 approaches zero:

$$\lim_{r_0 \to 0} \int_V \left[\nabla^2 A_z + k^2 A_z\right] dv = \lim_{r_0 \to 0} \int_V -\mu I d\ell \delta(\mathbf{r}) dv. \tag{14.24}$$

Using the sifting property of the delta function, the right-hand side of (14.24) is simply $-\mu I d\ell$. For the left-hand side, we first expand the integral associated with the second term in the square brackets

$$\lim_{r_0 \to 0} \int_{r=0}^{r_0} \int_{\theta=0}^{\pi} \int_{\phi=0}^{2\pi} k^2 C_1 \frac{e^{-jkr}}{r} r^2 \sin\theta \, d\phi \, d\theta \, dr. \tag{14.25}$$

Including the term $r^2 \sin\theta$, which is contributed by the volume element dv, the entire integrand is proportional to r. Therefore as r_0 (the upper limit of integration in r) goes to zero, this integral goes to zero.

The remaining integral on the left side of (14.24) is

$$\lim_{r_0\to 0}\int_V \nabla\cdot\nabla A_z dv = \lim_{r_0\to 0}\oint_S \nabla A_z\cdot\mathbf{ds} \tag{14.26}$$

where we have used the divergence theorem to convert the volume integral to a surface integral (and used the fact that $\nabla^2 = \nabla\cdot\nabla$). The integrand of the surface integral is given by

$$\nabla A_z|_{r=r_0} = \hat{r}\frac{\partial A_z}{\partial r}\bigg|_{r=r_0} = \hat{r}\left(-\frac{e^{-jkr_0}}{r_0^2} - jk\frac{e^{-jkr_0}}{r_0}\right)C_1. \tag{14.27}$$

The surface element $\mathbf{ds}$ is given by $\hat{r}\,r_0^2\sin\theta d\phi\,d\theta$ so that the entire surface integral is given by

$$\lim_{r_0\to 0}\int_{\theta=0}^{\pi}\int_{\phi=0}^{2\pi} C_1 e^{-jkr_0}(-1-jkr_0)\sin\theta d\phi\,d\theta = \lim_{r_0\to 0} -C_1 e^{-jkr_0}(1+jkr_0)4\pi = -C_1 4\pi. \tag{14.28}$$

Equating this with the right-hand side of (14.20), we can solve for C_1. The final result is

$$C_1 = \frac{\mu I d\ell}{4\pi}. \tag{14.29}$$

It should be noted that there is actually nothing special about having r_0 approach zero. The same coefficient is obtained for any value of r_0. Letting r_0 approach zero merely simplifies the problem a bit.

We now have that a filamentary current I of length $d\ell$ located at the origin and oriented in the z direction produces the vector potential

$$A_z(\mathbf{r}) = \frac{\mu I d\ell}{4\pi}\frac{e^{-jkr}}{r}. \tag{14.30}$$

This is simply a spherical wave radiating symmetrically away from the origin. If the source is located at the point $\mathbf{r}'$ instead of the origin, one merely needs to account for this displacement. The vector potential in that case is

$$A_z(\mathbf{r}) = \frac{\mu I d\ell}{4\pi}\frac{e^{-jk|\mathbf{r}-\mathbf{r}'|}}{|\mathbf{r}-\mathbf{r}'|}. \tag{14.31}$$

If the current was oriented in the x or y direction, that would produce a vector potential that only had a x or y component, respectively.

For a current source, the "strength" of the source is, in one way of thinking, determined by the amount of current that is flowing times the length over which that current flows. For a filament we have that the "source strength" is given by $Id\ell$. For a surface current the equivalent concept is $J_s ds$ where J_s is the surface current density in Amperes/meter and ds is an incremental surface area. Similarly, for a volumetric current, the equivalent term is Jdv where J is the current density in Amperes/meter^2 and dv is an incremental volume.

Instead of having just a point source, currents can be distributed throughout space. To get the corresponding vector potentials, we merely have to sum the contributions of the current wherever it

exists, accounting for the location (displacement from the origin), the orientation, and the amount of current.

For surface currents, the vector potentials are given by

$$\mathbf{A}(\mathbf{r}) = \mu \oint_S \mathbf{J}_s(\mathbf{r}') \frac{e^{-jk|\mathbf{r}-\mathbf{r}'|}}{4\pi|\mathbf{r}-\mathbf{r}'|} ds', \tag{14.32}$$

$$\mathbf{F}(\mathbf{r}) = \epsilon \oint_S \mathbf{M}_s(\mathbf{r}') \frac{e^{-jk|\mathbf{r}-\mathbf{r}'|}}{4\pi|\mathbf{r}-\mathbf{r}'|} ds', \tag{14.33}$$

where, as before, $\mathbf{r}$ is the observation point, $\mathbf{r}'$ is the location of the source point (i.e., the location of the currents), and S is the surface over which the current flows.

Equations (14.32) and (14.33) give the vector potentials at an arbitrary observation point $\mathbf{r}$ in three dimensions. The surface S is a surface that exists in the 3D space (such as the surface of sphere or a cube).

Let us consider the two-dimensional case. We can consider the 2D case as a special case in 3D in which there is no variation in the z direction. The observation point is a point in the xy plane specified by the vector $\boldsymbol{\rho}$. Thus, the magnetic vector potential could be written as

$$\mathbf{A}(\boldsymbol{\rho}) = \mu \oint_L \int_{z'=-\infty}^{\infty} \mathbf{J}_s(\boldsymbol{\rho}') \frac{e^{-jk\sqrt{|\boldsymbol{\rho}-\boldsymbol{\rho}'|^2+z'^2}}}{4\pi\sqrt{|\boldsymbol{\rho}-\boldsymbol{\rho}'|^2+z'^2}} dz' d\ell', \tag{14.34}$$

$$= \mu \oint_L \mathbf{J}_s(\boldsymbol{\rho}') \left(\int_{z'=-\infty}^{\infty} \frac{e^{-jk\sqrt{|\boldsymbol{\rho}-\boldsymbol{\rho}'|^2+z'^2}}}{4\pi\sqrt{|\boldsymbol{\rho}-\boldsymbol{\rho}'|^2+z'^2}} dz' \right) d\ell'. \tag{14.35}$$

The term in parentheses can be integrated to obtain

$$\int_{z'=-\infty}^{\infty} \frac{e^{-jk\sqrt{|\boldsymbol{\rho}-\boldsymbol{\rho}'|^2+z'^2}}}{4\pi\sqrt{|\boldsymbol{\rho}-\boldsymbol{\rho}'|^2+z'^2}} dz' = -\frac{j}{4} H_0^{(2)}(k|\boldsymbol{\rho}-\boldsymbol{\rho}'|) \tag{14.36}$$

where $H_0^{(2)}$ is the zeroth-order Hankel function of the second kind. This represents a cylindrical wave radiating from the point $\boldsymbol{\rho}'$. Similar steps can be done for $\mathbf{F}$ and in this way z is eliminated from the expressions for the vector potentials. We are left with a 2D representation of the fields, namely,

$$\mathbf{A}(\boldsymbol{\rho}) = -j\frac{\mu}{4} \oint_L \mathbf{J}(\boldsymbol{\rho}') H_0^{(2)}(k|\boldsymbol{\rho}-\boldsymbol{\rho}'|) d\ell', \tag{14.37}$$

$$\mathbf{F}(\boldsymbol{\rho}) = -j\frac{\epsilon}{4} \oint_L \mathbf{M}(\boldsymbol{\rho}') H_0^{(2)}(k|\boldsymbol{\rho}-\boldsymbol{\rho}'|) d\ell'. \tag{14.38}$$

where the explicit s subscript has been dropped from the currents.

The approximation of the zeroth-order Hankel function of the second kind as the argument ξ gets large is

$$H_0^{(2)}(k\xi) \approx \sqrt{\frac{j2}{\pi k\xi}} e^{-jk\xi}. \tag{14.39}$$

Now let $\xi = |\boldsymbol{\rho} - \boldsymbol{\rho}'|$ where ρ is large enough that the following approximations are valid:

$$\xi \approx \begin{cases} \rho - \rho' \cos\psi & \text{for the phase,} \\ \rho & \text{for the magnitude.} \end{cases} \tag{14.40}$$

where, referring to Fig. 14.2, ψ is the angle between the observation angle and the angle to the source point. (Because we are taking the cosine of ψ, and cosine is an even function, it doesn't matter if we define ψ as $\phi - \phi'$ or $\phi' - \phi$ but we will take ψ to be $\phi - \phi'$.) Thus the Hankel function can be written

$$H_0^{(2)}(k|\boldsymbol{\rho} - \boldsymbol{\rho}'|) \approx \sqrt{\frac{j2}{\pi k\rho}} e^{-jk\rho} e^{jk\rho' \cos\psi}. \tag{14.41}$$

The $\mathbf{A}$ vector potential for a 2D problem can thus be written

$$\begin{aligned} \mathbf{A}(\boldsymbol{\rho}) &= -j\frac{\mu}{4} \oint_L \mathbf{J}(\boldsymbol{\rho}') H_0^{(2)}(k|\boldsymbol{\rho} - \boldsymbol{\rho}'|) d\ell', && (14.42) \\ &\approx -j\frac{\mu}{4} \sqrt{\frac{j2}{\pi k\rho}} e^{-jk\rho} \oint_L \mathbf{J}(\boldsymbol{\rho}') e^{jk\rho' \cos\psi} d\ell', && (14.43) \\ &= -j\frac{\mu}{4} \sqrt{\frac{j2}{\pi k\rho}} e^{-jk\rho} \mathbf{N}_{2D}, && (14.44) \end{aligned}$$

where

$$\mathbf{N}_{2D} = \oint_L \mathbf{J}(\boldsymbol{\rho}') e^{jk\rho' \cos\psi} d\ell'. \tag{14.45}$$

Correspondingly, the $\mathbf{F}$ vector potential can be written

$$\begin{aligned} \mathbf{F}(\boldsymbol{\rho}) &= -j\frac{\epsilon}{4} \oint_L \mathbf{M}(\boldsymbol{\rho}') H_0^{(2)}(k|\boldsymbol{\rho} - \boldsymbol{\rho}'|) d\ell', && (14.46) \\ &\approx -j\frac{\epsilon}{4} \sqrt{\frac{j2}{\pi k\rho}} e^{-jk\rho} \oint_L \mathbf{M}(\boldsymbol{\rho}') e^{jk\rho' \cos\psi} d\ell', && (14.47) \\ &= -j\frac{\epsilon}{4} \sqrt{\frac{j2}{\pi k\rho}} e^{-jk\rho} \mathbf{L}_{2D}, && (14.48) \end{aligned}$$

where

$$\mathbf{L}_{2D} = \oint_L \mathbf{M}(\boldsymbol{\rho}') e^{jk\rho' \cos\psi} d\ell'. \tag{14.49}$$

Nominally $\mathbf{N}_{2D}$ and $\mathbf{L}_{2D}$ are functions of $\boldsymbol{\rho}$. However, within these functions the only thing that depends on $\boldsymbol{\rho}$ is ψ. The angle ψ only changes for large changes in $\boldsymbol{\rho}$—incremental changes of $\boldsymbol{\rho}$ will not affect ψ. Thus, derivatives of $\mathbf{N}_{2D}$ or $\mathbf{L}_{2D}$ with respect to ρ, ϕ, or z (i.e., derivatives with respect to the unprimed coordinates) are zero. The geometry is depicted in Fig. 14.2.

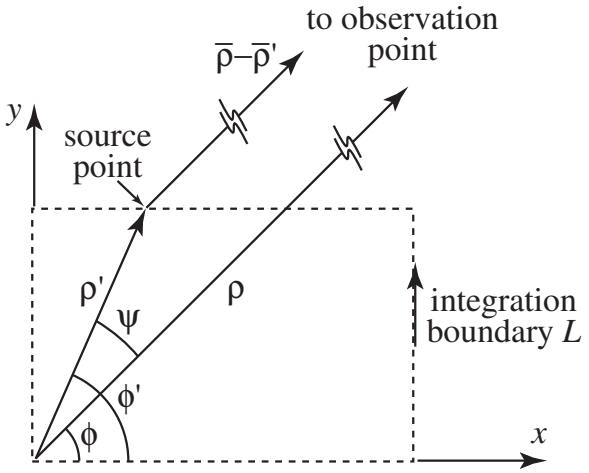


Figure 14.2: Geometry associated with the near-to-far-field transformation in 2D.

It is convenient to think of the currents, and subsequently $\mathbf{N}_{2D}$ and $\mathbf{L}_{2D}$ (and ultimately the potentials), in terms of cylindrical coordinates, i.e.,

$$\mathbf{J}(\boldsymbol{\rho}) = J_\rho \hat{\mathbf{a}}_\rho + J_\phi \hat{\mathbf{a}}_\phi + J_z \hat{\mathbf{a}}_z, \tag{14.50}$$

$$\mathbf{M}(\boldsymbol{\rho}) = M_\rho \hat{\mathbf{a}}_\rho + M_\phi \hat{\mathbf{a}}_\phi + M_z \hat{\mathbf{a}}_z. \tag{14.51}$$

Combining the contributions from both $\mathbf{A}$ and $\mathbf{F}$, the electric and magnetic fields are given by

$$\mathbf{E}(\boldsymbol{\rho}) = -j\omega \left[\mathbf{A} + \frac{1}{k^2} \nabla(\nabla \cdot \mathbf{A}) \right] - \frac{1}{\epsilon} \nabla \times \mathbf{F}, \tag{14.52}$$

$$\mathbf{H}(\boldsymbol{\rho}) = -j\omega \left[\mathbf{F} + \frac{1}{k^2} \nabla(\nabla \cdot \mathbf{F}) \right] + \frac{1}{\mu} \nabla \times \mathbf{A}. \tag{14.53}$$

By plugging (14.44) and (14.48) into (14.52) and (14.53) and performing the various operations in cylindrical coordinates and discarding any terms that fall off faster than $1/\sqrt{\rho}$, one can obtain expressions for the electric and magnetic fields in the far field. (Note that the ∇ operator acts on the unprimed coordinates and, as mentioned above, $\mathbf{N}_{2D}$ and $\mathbf{L}_{2D}$ are not considered functions of the unprimed coordinates.)

14.4 Electric Field in the Far-Field

Following the steps outlined in the previous section, the scattered electric field $\mathbf{E}^s$ at the far-field point $\boldsymbol{\rho}$ can be obtained from the scattered "near" fields using

$$\mathbf{E}^s(\boldsymbol{\rho}) = \sqrt{\frac{j}{8\pi k}} \frac{e^{-jk\rho}}{\sqrt{\rho}} \left\{ \hat{\mathbf{a}}_\phi \left(\omega\mu_0 \mathbf{N}_{2\mathrm{D}} \cdot \hat{\mathbf{a}}_\phi + k \mathbf{L}_{2\mathrm{D}} \cdot \hat{\mathbf{a}}_z \right) - \hat{\mathbf{a}}_z \left(\omega\mu_0 \mathbf{N}_{2\mathrm{D}} \cdot \hat{\mathbf{a}}_z - k \mathbf{L}_{2\mathrm{D}} \cdot \hat{\mathbf{a}}_\phi \right) \right\}, \tag{14.54}$$

where, as stated previously,

$$\mathbf{N}_{\text{2D}} = \oint_L \mathbf{J}(\boldsymbol{\rho}')e^{jk\rho'\cos\psi}d\ell', \tag{14.55}$$

$$\mathbf{L}_{\text{2D}} = \oint_L \mathbf{M}(\boldsymbol{\rho}')e^{jk\rho'\cos\psi}d\ell', \tag{14.56}$$

L is the closed path of integration, ψ, given by $\phi - \phi'$, is the angle between the source point and observation point, $\mathbf{M} = -\hat{\mathbf{n}}' \times \mathbf{E}$, $\mathbf{J} = \hat{\mathbf{n}}' \times \mathbf{H}$, and $\hat{\mathbf{n}}'$ is a unit vector normal to the integration contour on the same side of the contour as the observation point (i.e., the outward normal). Unprimed coordinates correspond to the observation point while primed coordinates indicate the "source" location (i.e., points along the contour).

Let us now restrict consideration to TM^z polarization where the non-zero field are H_x, H_y, and E_z. Since the outward normal is restricted to exist in the xy plane, $\hat{\mathbf{n}}' \times \mathbf{H}$ only has a non-zero component in the z direction while $-\hat{\mathbf{n}}' \times \mathbf{E}$ only has non-zero components in the xy plane. Thus, for this polarization only the z component of the electric field is non-zero—the ϕ component of (14.54) is zero. The electric field can be written

$$E_z^s(\boldsymbol{\rho}) = -\sqrt{\frac{j}{8\pi k}}\frac{e^{-jk\rho}}{\sqrt{\rho}}\oint_L \left(\omega\mu_0\mathbf{J}(\boldsymbol{\rho}')\cdot\hat{\mathbf{a}}_z - k\mathbf{M}(\boldsymbol{\rho}')\cdot\hat{\mathbf{a}}_\phi\right)e^{jk\rho'\cos\psi}d\ell'. \tag{14.57}$$

Usually the scattering width is of more interest than the field itself. For TM^z polarization the two-dimensional scattering width is defined to be

$$\sigma_{\text{2D}}(\phi) = \lim_{|\boldsymbol{\rho}-\boldsymbol{\rho}'|\to\infty} 2\pi\rho\frac{|E_z^s(\boldsymbol{\rho})|^2}{|E_z^i|^2} \tag{14.58}$$

Noting that $\omega\mu_0 = k\eta_0$ and plugging (14.57) into (14.58) and normalizing by the wavelength yields

$$\frac{\sigma_{\text{2D}}(\phi)}{\lambda} = \frac{1}{8\pi|E_z^i|^2}\left|\oint_L \left\{\eta_0\mathbf{J}(\boldsymbol{\rho}')\cdot\hat{\mathbf{a}}_z - \mathbf{M}(\boldsymbol{\rho}')\cdot\hat{\mathbf{a}}_\phi\right\}e^{jkr'\cos(\phi-\phi')}kd\ell'\right|^2. \tag{14.59}$$

The term $r'\cos(\phi-\phi')$ which appears in the exponent can be written as $\hat{\mathbf{a}}_\rho\cdot\boldsymbol{\rho} = \hat{\mathbf{a}}_\rho\cdot(x'\hat{\mathbf{a}}_x + y'\hat{\mathbf{a}}_y) = x'\cos\phi + y'\sin\phi$. This last form is especially useful since ϕ is fixed by the observation direction and therefore the sine and cosine functions can be determined outside of any loop (rather than over and over again as we move along the integration contour).

For TM^z polarization the unit vector normal to the integration path is restricted to lie in the xy plane, i.e., $\hat{\mathbf{n}}' = n_x'\hat{\mathbf{a}}_x + n_y'\hat{\mathbf{a}}_y$ where $(n_x'^2 + n_y'^2)^{1/2} = 1$. Thus, the electric current $\mathbf{J}$ is given by

$$\mathbf{J} = \hat{\mathbf{n}}' \times \mathbf{H} = \begin{vmatrix} \hat{\mathbf{a}}_x & \hat{\mathbf{a}}_y & \hat{\mathbf{a}}_z \\ n_x' & n_y' & 0 \\ H_x & H_y & 0 \end{vmatrix} = \hat{\mathbf{a}}_z(n_x'H_y - n_y'H_x). \tag{14.60}$$

The dot product of $\hat{\mathbf{a}}_z$ and $\mathbf{J}$ yields

$$\mathbf{J}\cdot\hat{\mathbf{a}}_z = n_x'H_y - n_y'H_x. \tag{14.61}$$

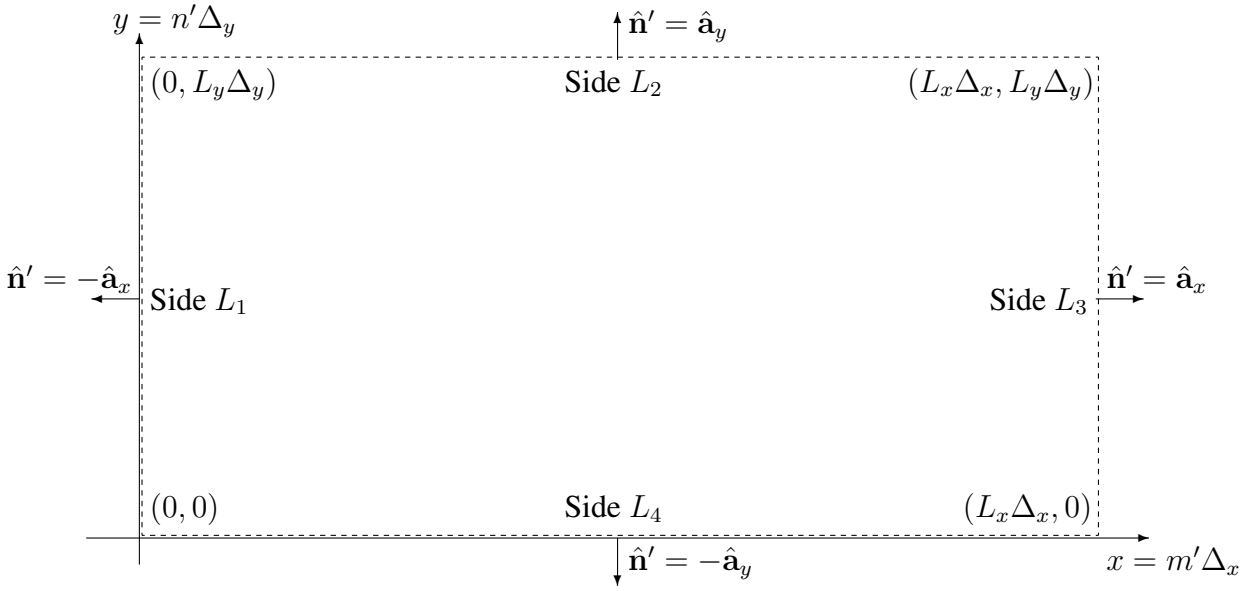


Figure 14.3: Depiction of integration boundary for near-to-far-field transformation in the FDTD grid.

The magnetic current is given by

$$\mathbf{M} = -\hat{\mathbf{n}}' \times \mathbf{E} = \begin{vmatrix} \hat{\mathbf{a}}_x & \hat{\mathbf{a}}_y & \hat{\mathbf{a}}_z \\ n'_x & n'_y & 0 \\ 0 & 0 & E_z \end{vmatrix} = -(\hat{\mathbf{a}}_x n'_y E_z - \hat{\mathbf{a}}_y n'_x E_z) \tag{14.62}$$

The dot product of $\hat{\mathbf{a}}_\phi$ and $\mathbf{M}$ yields

$$\mathbf{M} \cdot \hat{\mathbf{a}}_\phi = -\hat{\mathbf{a}}_x \cdot \hat{\mathbf{a}}_\phi n'_y E_z + \hat{\mathbf{a}}_y \cdot \hat{\mathbf{a}}_\phi n'_x E_z = (n'_y \sin\phi + n'_x \cos\phi) E_z \tag{14.63}$$

Incorporating (14.61) and (14.63) into (14.59) yields a general expression for the scattering width:

$$\frac{\sigma_{2D}(\phi)}{\lambda} = \frac{1}{8\pi |E_z^i|^2} \left| \oint_L \left\{ \eta_0 (n'_x H_y - n'_y H_x) - (n'_y \sin\phi + n'_x \cos\phi) E_z \right\} e^{jkr' \cos(\phi - \phi')} k d\ell' \right|^2 . \tag{14.64}$$

We now want to specialize this equation to a rectangular box which is typical of the integration boundary which would be employed in an FDTD simulation.

Assume the integration boundary corresponds to the dashed box shown in Fig. 14.3. The width of this rectangle is $L_x\Delta_x$ and the height is $L_y\Delta_y$. In an FDTD grid there are $L_x + 1$ samples of the fields along the top and bottom (i.e., spanning the width) and $L_y + 1$ total samples along the left and right (i.e., spanning the height). The integration over the closed path L consists of the integration over the four sides of this box, i.e., $L = L_1 + L_2 + L_3 + L_4$. Using this geometry, the quantities needed to perform each integral are presented in the following two tables.

	$\hat{\mathbf{n}}'$		$\mathbf{J}\cdot\hat{\mathbf{a}}_z$	$\mathbf{M}\cdot\hat{\mathbf{a}}_\phi$
Side L_1	$n'_x=-1$	$n'_y=0$	$-H_y$	$-\cos\phi E_z$
Side L_2	$n'_x=0$	$n'_y=1$	$-H_x$	$\sin\phi E_z$
Side L_3	$n'_x=1$	$n'_y=0$	H_y	$\cos\phi E_z$
Side L_4	$n'_x=0$	$n'_y=-1$	H_x	$-\sin\phi E_z$

	ϕ'	ρ'	$\hat{\mathbf{a}}_\rho\cdot\boldsymbol{\rho}'$
Side L_1	$\pi/2$	$n'\Delta_y$	$n'\Delta_y\sin\phi$
Side L_2	$\tan^{-1}(L_y/m')$	$\sqrt{(m'\Delta_x)^2+(L_y\Delta_y)^2}$	$m'\Delta_x\cos\phi+L_y\Delta_y\sin\phi$
Side L_3	$\tan^{-1}(n'/L_x)$	$\sqrt{(L_x\Delta_x)^2+(n'\Delta_y)^2}$	$L_x\Delta_x\cos\phi+n'\Delta_y\sin\phi$
Side L_4	0	$m'\Delta_x\cos\phi$	$m'\Delta_y$

Note that in the second table the value n' represents the vertical displacement along Side L_1 or L_3. It varies between 0 and L_y and should not be confused with the outward normal $\hat{\mathbf{n}}'$ and its components n'_x and n'_y.

Assume that the spatial step sizes are equal so that $\Delta_x=\Delta_y=\delta$. Further assume that the problem has be discretized using N_λ points per wavelength so that the wavenumber can be written $k=2\pi/\lambda=2\pi/(N_\lambda\delta)$. Combining all together, the scattering width is given by

$$\begin{aligned}\frac{\sigma_{2D}(\phi)}{\lambda} &= \frac{1}{8\pi|E_z^i|^2}\left|\int_{m'=0}^{L_x}\Big\{[\eta_0 H_x(m',0)+\sin\phi E_z(m',0)]\,e^{j\frac{2\pi}{N_\lambda}m'\cos\phi}\right.\\ &\qquad\left.-\,[\eta_0 H_x(m',L_y)+\sin\phi E_z(m',L_y)]\,e^{j\frac{2\pi}{N_\lambda}(m'\cos\phi+L_y\sin\phi)}\Big\}\frac{2\pi}{N_\lambda}dm'\right.\\ &\quad+\int_{n'=0}^{L_y}\Big\{-[\eta_0 H_y(0,n')-\cos\phi E_z(0,n')]\,e^{j\frac{2\pi}{N_\lambda}n'\sin\phi}\\ &\qquad+\,[\eta_0 H_y(L_x,n')-\cos\phi E_z(L_x,n')]\,e^{j\frac{2\pi}{N_\lambda}(L_x\cos\phi+n'\sin\phi)}\Big\}\frac{2\pi}{N_\lambda}dn'\Big|^2 \end{aligned}\quad(14.65)$$

Again we note that the integration variable for the second integral is n' which corresponds to displacement along the vertical sides. The fields in this expression are phasor (frequency-domain) quantities and hence are complex. Although this equation may look rather messy, as described in the next two sections, the phasors can be obtained rather simply with a running DFT and the integration can be calculated with a sum.

Because of the staggered nature of the FDTD grid, the electric and magnetic fields will not be collocated on the integration boundary—they will be offset in both space and time. A spatial average of either the electric or magnetic field will have to be performed to obtain the fields at the proper location. In the past, a simple arithmetic average was typically used to account for the spatial offsets. However, as will discussed, one can do better by using a harmonic average in space. For the temporal offset, a simple phase correction can be used to collocate the fields in time.

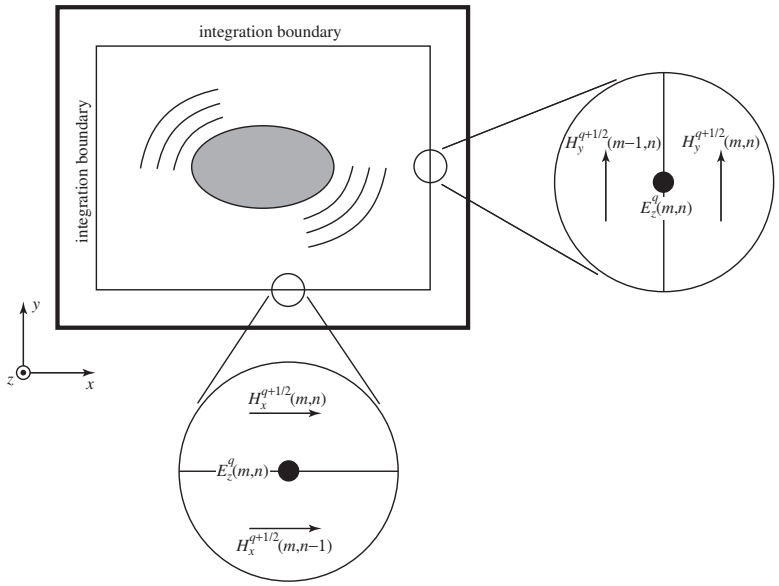


Figure 14.4: Depiction of a TM^z grid showing the integration boundary. The boundary is assumed to be aligned with electric-field nodes. The expanded views show the offset of the magnetic-field nodes from the boundary.

14.5 Simpson's Composite Integration

Assume we wish to integrate the function $f(x)$ over the interval $0 \leq x \leq L$ where L is an even integer. The integral can be obtained using Simpson's composite integration as follows

$$\int_0^L f(x)dx \approx \frac{1}{3}\left[f(0) + 2\sum_{m=1}^{L/2-1} f(2m) + 4\sum_{m=1}^{L/2} f(2m-1) + f(L)\right] \tag{14.66}$$

Note that this approximation requires a total of $L+1$ samples of the function (so we need an odd number of samples). Using Simpson's approximation yields quite a bit of additional accuracy over what would be obtained using a straight Riemann sum and it costs essentially nothing (it just requires slightly more bookkeeping).

14.6 Collocating the Electric and Magnetic Fields: The Geometric Mean

Figure 14.4 depicts an integration boundary in a TM^z grid. The boundary is assumed to be aligned with the electric-field nodes, i.e., the E_z nodes. The expanded views show a portion of the boundary along the right side and the bottom. The field notation employs superscripts to indicate time steps while spatial indices are given as arguments within parentheses. Half-step spatial offsets

are implicitly understood. Thus, the nodes in space-time and the corresponding notation are

$$H_x^{(q-1/2)\Delta_t}(m\Delta_x, [n+1/2]\Delta_y) = H_x^{q-1/2}(m,n), \tag{14.67}$$

$$H_y^{(q-1/2)\Delta_t}([m+1/2]\Delta_x, n\Delta_y) = H_y^{q-1/2}(m,n), \tag{14.68}$$

$$E_z^{q\Delta_t}(m\Delta_x, n\Delta_y) = E_z^q(m,n), \tag{14.69}$$

where Δ_x and Δ_y are the spatial steps in the x and y directions, respectively, and Δ_t is the temporal step. The index q indicates the temporal step and we assume it varies between 1 and N_T which is the total number of time-steps.

Near-to-far-field (NTFF) transforms require that the fields be defined over a single surface and use the same phase reference. For harmonic fields, the temporal offset can be easily accounted for with a phase factor. Assume the magnetic fields have been recorded at times of $q = 1/2, 3/2, 5/2, \ldots$ while the electric field has been recorded at times of $q = 1, 2, 3, \ldots$ For the harmonic transforms of interest here, the time-domain near-fields are Fourier transformed to the frequency domain. For example, the harmonic electric field on the boundary is given by

$$\hat{E}_z^k(m,n) = \mathcal{F}\left[E_z^q(m,n)\right], \tag{14.70}$$

$$= \frac{1}{N_T}\sum_{q=\langle N_T\rangle} E_z^q(m,n)e^{-jk\frac{2\pi}{N_T}q}. \tag{14.71}$$

where $\mathcal{F}$ is the discrete Fourier transform. For situations where the entire spectrum is not of interest, typically a running discrete Fourier transform will be used only at the particular frequencies of interest. A frequency is specified by the index k which varies between zero (dc) and $N_T - 1$. Regardless of the implementation used, the resulting spectral terms $\hat{E}_z^k$ will be the same.

The time-domain series $E_z^q(m,n)$ can be obtained from $\hat{E}_z^k(m,n)$ via

$$E_z^q(m,n) = \sum_{k=\langle N_T\rangle} \hat{E}_z^k(m,n)e^{jk\frac{2\pi}{N_T}q}. \tag{14.72}$$

Because of the temporal offset between the electric and magnetic fields, the desired field is actually $E_z^{q-1/2}(m,n)$, thus, plugging $q - 1/2$ into (14.72) yields

$$E_z^{q-1/2}(m,n) = \sum_{k=\langle N_T\rangle} \left(\hat{E}_z^k(m,n)e^{-jk\frac{\pi}{N_T}}\right)e^{jk\frac{2\pi}{N_T}q}. \tag{14.73}$$

In practice one calculates $\hat{E}_z^k(m,n)$ and the spectral representation of the magnetic fields in the same way, i.e., as in (14.71). Then one multiplies $\hat{E}_z^k(m,n)$ by $\exp(-jk\pi/N_T)$ to account for the temporal offset.

The spatial offset is slightly more problematic than the temporal offset. As shown in Fig. 14.4, the integration boundary can be aligned with only one of the fields. The magnetic field tangential to the integration boundary is found from the nodes that are a spatial half-step to either side of the boundary.

To obtain the magnetic field on the boundary, the traditional approach has been to use a spatial average of the nodes to either side of the boundary. For example, along the right side of the boundary, the harmonic magnetic field would be given by

$$\hat{H}_y^k(m,n) = \frac{1}{2}\mathcal{F}\left[H_y^{q-1/2}(m-1,n) + H_y^{q-1/2}(m,n)\right]. \tag{14.74}$$

Because of this spatial average, $\hat{H}_y(m,n)$ and $\hat{E}_z(m,n)$ are assumed to be collocated and, with the temporal phase correction, can be used to determine the equivalent currents over the integration boundary (which are then used in the NTFF transform itself).

Unfortunately the arithmetic mean used in (14.74) introduces errors. To illustrate this, assume a harmonic plane wave is propagating in the grid. The temporal frequency ω is $2\pi k'/N_T$ where k' is an integer constant and, as before, N_T is the total number of time-steps in a simulation. The y component of the magnetic field is given by

$$H_y^{q-1/2}(m,n) = \cos\left(\omega\left[q-\frac{1}{2}\right]\Delta_t - \xi\right) \tag{14.75}$$

$$= \frac{e^{j\left(k'\frac{2\pi}{N_T}[q-\frac{1}{2}]\Delta_t+\xi\right)} + e^{-j\left(k'\frac{2\pi}{N_T}[q-\frac{1}{2}]\Delta_t+\xi\right)}}{2}, \tag{14.76}$$

where $\xi = \beta_x(m+1/2)\Delta_x + \beta_y n\Delta_y$, and β_x and β_y are the x and y components of the wave vector, respectively. Taking the discrete Fourier transform of (14.76), i.e.,

$$\hat{H}_y^k(m,n) = \frac{1}{N_T}\sum_{q=\langle N_T\rangle} H_y^{q-1/2}(m,n)e^{-jk\frac{2\pi}{N_T}\left(q-\frac{1}{2}\right)}, \tag{14.77}$$

one notes that the sum yields zero when k is anything other than k' or $N_T - k'$. The values of k that yield non-zero correspond to the positive and negative frequency of the continuous world and, like the continuous world, the corresponding spectral values are complex conjugates. Without loss of generality, we will continue the discussion in terms of the spectral component corresponding to the positive frequency, i.e.,

$$\hat{H}_y^{k'}(m,n) = \frac{1}{2}\exp(-j[\beta_x(m+1/2)\Delta_x + \beta_y n\Delta_y]). \tag{14.78}$$

(Note that since the time-domain functions are real-valued, in practice one does not need to calculate the transform at any of the negative frequencies. They are merely the complex conjugates of the values at the positive frequencies.)

Because the Fourier transform is a linear operator, using (14.76) in (14.74) yields

$$\hat{H}_y^{k'}(m,n) = e^{-j\beta_y n\Delta_y}\frac{e^{-j\beta_x\left(m-\frac{1}{2}\right)\Delta_x} + e^{-j\beta_x\left(m+\frac{1}{2}\right)\Delta_x}}{4}, \tag{14.79}$$

$$= \frac{1}{2}e^{-j(\beta_x m\Delta_x+\beta_y n\Delta_y)}\cos\left(\frac{\beta_x\Delta_x}{2}\right). \tag{14.80}$$

The exact expression for the magnetic field on the integration boundary is $\exp(-j[\beta_x m\Delta_x + \beta_y n\Delta_y])/2$. Thus, the cosine term represents an error—one which vanishes only in the limit as the spatial-step size goes to zero.

Instead of taking the Fourier transform of the average of the time-domain fields, let us take the Fourier transform of the fields to either side of the boundary. We define the transforms as

$$\hat{H}_y^+(m,n) = \mathcal{F}\left[H_y^{q-1/2}(m,n)\right], \tag{14.81}$$

$$\hat{H}_y^-(m,n) = \mathcal{F}\left[H_y^{q-1/2}(m-1,n)\right]. \tag{14.82}$$

Still assuming a single harmonic plane wave, for the "positive frequency" corresponding to $k = k'$, $\hat{H}_y^+(m,n)$ and $\hat{H}_y^-(m,n)$ are given by

$$\hat{H}_y^+(m,n) = \frac{1}{2}e^{-j(\beta_x(m+1/2)\Delta_x+\beta_y n\Delta_y)}, \tag{14.83}$$

$$\hat{H}_y^-(m,n) = \frac{1}{2}e^{-j(\beta_x(m-1/2)\Delta_x+\beta_y n\Delta_y)}. \tag{14.84}$$

Were one to calculate the arithmetic mean of $\hat{H}_y^+(m,n)$ and $\hat{H}_y^-(m,n)$, the result would be the same as given in (14.80). However, consider the geometric mean (where the geometric mean of a and b is $\sqrt{ab}$) of $\hat{H}_y^+(m,n)$ and $\hat{H}_y^-(m,n)$:

$$\hat{H}_y^{k'}(m,n) = \left(\hat{H}_y^+(m,n)\hat{H}_y^-(m,n)\right)^{1/2}, \tag{14.85}$$

$$= \frac{1}{2}e^{-j(\beta_x m\Delta_x+\beta_y n\Delta_y)}. \tag{14.86}$$

This is precisely the correct answer. There is no error introduced by the geometric mean. A note of caution: when calculating the square root of these complex quantities, one must ensure that the proper branch cut is selected. Thus, when $\hat{H}_y^+(m,n)$ and $\hat{H}_y^-(m,n)$ have phases near $\pm\pi$ the geometric mean should also have a phase near $\pm\pi$ rather than near zero.

In practice, at any given frequency there will be an angular spectrum of wave vectors present and hence any averaging, whether geometric or arithmetic, will introduce some errors. However, for a single wave vector the geometric mean is exact and it has been our experience that the geometric mean provides superior results for nearly all discretizations and scattering angles. The following section demonstration the use of the geometric mean in several scenarios.

14.7 NTFF Transformations Using the Geometric Mean

14.7.1 Double-Slit Radiation

To demonstrate the difference between the arithmetic and geometric mean, we begin by considering the radiation from a double-slit aperture in a perfect electrical-conductor (PEC) screen which is illuminated by a normally incident pulsed plane wave. TM^z polarization is assumed. As shown in Fig. 14.5(a), in this case the boundary over which the fields are measured is three-sided and exists on only one side of the screen.

Given the fields over the three-sided boundary, one then assumes the fields "interior" to this boundary (i.e., the region which includes the slits) are zero while the fields exterior to the boundary are unchanged. To account for the discontinuity in the fields across the integration boundary, surface currents must be present. Since the fields are zero within the boundary, one can replace the actual interior with anything without affecting the exterior fields. One thus assumes that the slits are not present—that the PEC plane is unbroken. The surface currents over the three-sided boundary are now radiating in the presence of an infinite plane. The far-field radiation can be calculated with a three-sided integral where one uses the Green's function for a source above an infinite plane. This, equivalently, from image theory, is simply the radiation from the original (measured) current and the image of the current. Both the measured current and the image current are radiating in free

space. In this way the three-sided boundary can be replaced with a closed four-sided boundary as shown in Fig. 14.5(b). The corresponding currents over this surface, i.e., the measured currents over half the boundary and the image currents over the other half, are transformed to the far field.

The incident field is introduced over a total-field/scattered-field (TFSF) boundary which only exists to the left side of the screen. The grid is terminated with an eight-cell perfectly matched layer (PML).

The right side of the integration surface is D cells away from the PEC screen. The length of the right side of the integration boundary is held fixed at 75 cells. In principle, the location of the integration boundary should make no difference to the far-fields. However, when using the arithmetic mean, the far-fields are sensitive to the boundary location, i.e., sensitive to the value of D. Note that were a single component of the field integrated over the aperture, as advocated by [1], averaging is not an issue. However, that approach is restricted to screens which are planar and there are no inhomogeneities present other than the screen. The approach we advocate can accommodate any screen or scatterer geometry provided it can be contained within the integration boundary. Nevertheless, we will employ the aperture-based approach as a reference solution.

The simulation uses "slits" which are 15 cells wide. The PEC between the slits is 30 cells wide. The excitation is a Ricker wavelet discretized such that there are 30 cells per wavelength at the most energetic frequency. The simulation is run at the 2D Courant limit $(1/\sqrt{2})$ for 1024 time steps.

Figure 14.6(a) shows the far-field radiation pattern which is obtained using the arithmetic mean. The pattern is symmetric about zero degrees which corresponds to the direction normal to the screen. The radiation pattern is calculated using

$$\frac{1}{\lambda}\lim_{\rho\to\infty}\left[2\pi\rho\frac{|\hat{E}_z(\phi)|^2}{|\hat{E}_z^i|^2}\right] \tag{14.87}$$

where ϕ is the scattering angle, ρ is the distance from the slits, $\hat{E}_z(\phi)$ is the field radiated in the ϕ direction, and $\hat{E}_z^i$ is the complex amplitude of the incident plane wave at the frequency of interest. Results are shown for a frequency corresponding to 10.0566 cells per wavelength. Figure 14.6(a) shows the pattern when D is either 5, 6, or 7 cells.

Note that there are significant differences in the central peak depending on the displacement D between the right-side integration boundary and the PEC screen. Figure 14.6(b) shows an expanded view of the pattern in the neighborhood of the peak. As can be seen, displacing the integration boundary by two cells causes a change in the peak of approximately 9 percent. The results as a function of displacement are nearly periodic, e.g., displacements of 5 and 15 cells (not shown) yield nearly the same results as do displacements of 6 and 16 cells, and so on. (The period of 10 cells is a consequence of examining a frequency corresponding to approximately 10 cells per wavelength.) Also shown as plus signs in Fig. 14.6(b) are the results obtained when the transform uses the electric field (i.e., the equivalent magnetic current) over the aperture. No averaging is involved in this case. The aperture-based results are seen to agree well with the arithmetic-mean results when the displacement D is 5 cells. Unfortunately one does not know *a priori* that this agreement will exist nor does this displacement provide similar agreement for other frequencies.

On the other hand, when using the geometric mean, there is almost no variation in the radiation pattern as the integration boundary is displaced. Figure 14.6(c) shows the same results as presented in Fig. 14.6(b) except now the geometric mean of the harmonic fields is used to obtain the magnetic

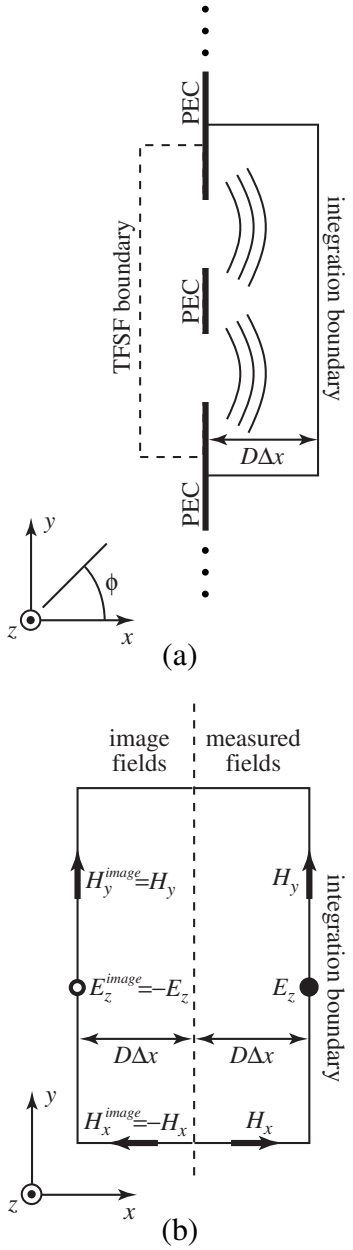
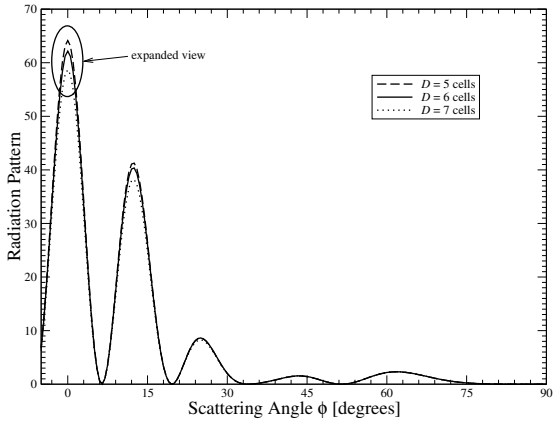
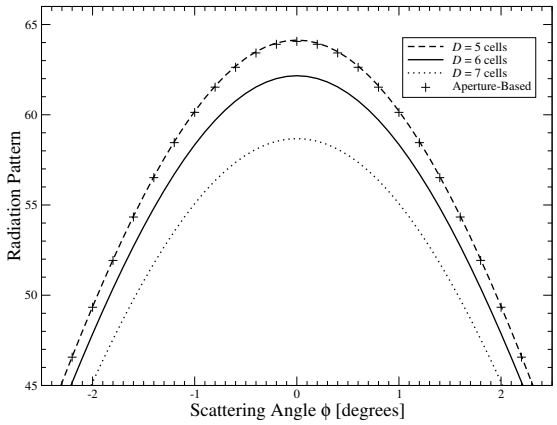


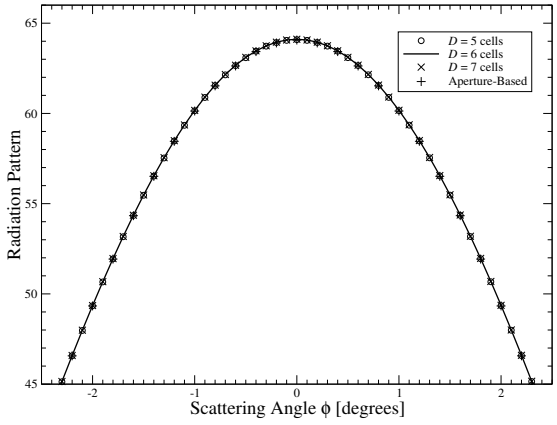
Figure 14.5: (a) Geometry of the double-slit experiment. A pulsed plane wave is introduced via a TFSF boundary on the left side of the screen. The field are recorded over the three-sided integration boundary to the left of the screen. (b) Image theory is used to create a four-sided closed surface over which the currents are transformed to the far-field. The dashed line corresponds to the location where the PEC plane had been.



(a)



(b)



(c)

Figure 14.6: (a) Radiation from the double slit for angles between -5 and 90 degrees (the pattern is symmetric about zero degrees). Results are shown for boundary displacements D of 5, 6, and 7 cells. The arithmetic mean is used. (b) Expanded view of the central peak using the arithmetic mean. Also shown are the fields obtained when a single field component is recorded over the aperture and transformed to the far field. (c) Same as (b) except the geometric mean is used. The variation of the fields caused by the displacement of the boundary is now essentially negligible.

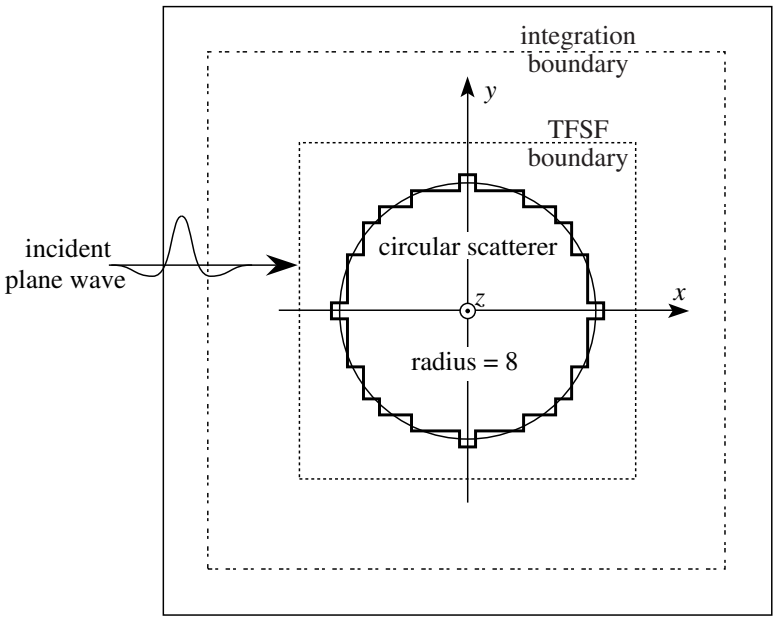


Figure 14.7: Geometry of the circular PEC cylinder.

fields on the integration boundary. The variation between these peaks is less than 0.022, i.e., a reduction in variation by a factor of approximately 270. This demonstrates that, unlike with the arithmetic mean, the location of the integration boundary is effectively irrelevant when using the GM-NTFF transform.

Naturally, at finer discretizations, the difference between the geometric mean and the arithmetic mean are less dramatic, but the geometric mean consistently performs better than the arithmetic mean.

(Simpson's rule was used for all integrations except for the integration of the aperture fields where a Riemann sum was used.)

14.7.2 Scattering from a Circular Cylinder

Consider scattering from a PEC circular cylinder under TM^z polarization as shown in Fig. 14.7. The Dey-Mittra scheme is employed to help reduce the effects of staircasing [2]. The cylinder has eight cells along its radius. A pulsed plane wave which travels in the x direction is introduced via a TFSF boundary. The simulation is run 512 time steps. Because the Dey-Mittra scheme is used, the Courant number was reduced to approximately 35 percent of the 2D limit in order to ensure stability [3].

Figure 14.8 shows the wavelength-normalized scattering width of the cylinder as a function of scattering angle obtained using the series solution for a circular cylinder [4], the arithmetic-mean NTFF transform, and the GM-NTFF transform. (The normalized scattering width is the same as the radiation pattern given in (14.87) where ρ is now taken to be the distance from the center of the cylinder.) The discretization is such that there are 9.92526 cells per wavelength at the frequency being considered here. One should keep in mind that the discretization of the cylinder introduces some errors and hence the "exact" solution for a circular cylinder is not truly exact for the scatterer

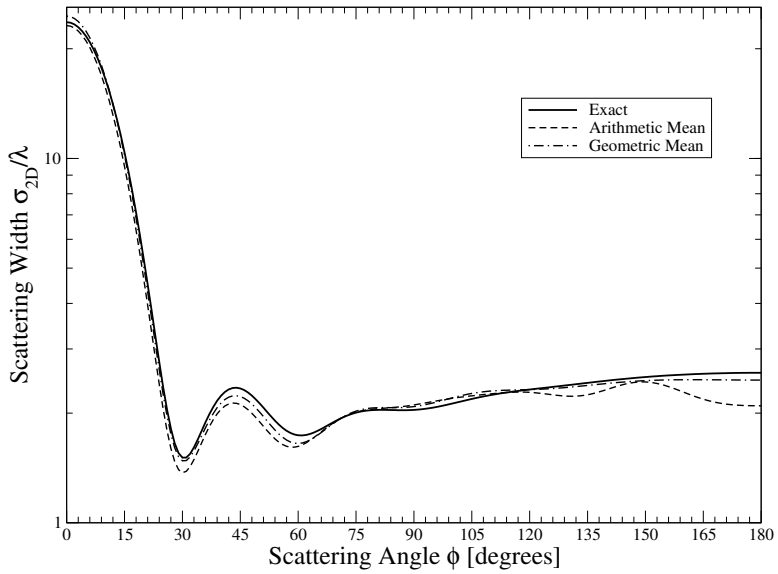


Figure 14.8: Scattering width of a circular cylinder. The radius is eight cells and the frequency corresponds to 9.92526 cells per wavelength.

present in the simulation. Therefore, the reference solution does not provide a perfect way with which to judge the solutions. Nevertheless, one hopes that the FDTD scatterer, when employing the Dey-Mittra scheme, is a close approximation to a true circular scatterer and thus the exact solution from the continuous world provides a reasonable basis for comparison.

The difference between the solutions in Fig. 14.8 are seen to be relatively small. Figure 14.9 shows a plot of the magnitude of the difference between the exact solution and the FDTD-based solutions. Although there are angles where the arithmetic mean performs better than the geometric mean, in general the geometric mean is better. The integrated error for angles between 0 and π is 0.6936 for the arithmetic mean and 0.3221, i.e., the error is reduced by more than a factor of two by using the geometric mean.

14.7.3 Scattering from a Strongly Forward-Scattering Sphere

Finally, consider scattering from a dielectric sphere, depicted in Fig. 14.10, which has a relative permittivity ϵ_r of 1.21. Such a sphere was considered in [5] and can also be found in Sec. 8.7 of [6]. In this case the transformation traditionally entails finding the tangential fields over the six sides of a cuboid which bounds the sphere. The sphere is discretized such that there are 60 cells along the radius. A staircase representation is used (where a node is simply either inside or outside the sphere). The simulation is run at 95 percent of the 3D Courant limit ($0.95/\sqrt{3}$) for 2048 time steps. The grid is terminated with an eight-cell perfectly-matched layer.

Figure 14.11 shows the normalized scattering cross section as a function of the equatorial angle ϕ. The frequency corresponds to 20.06 cells per wavelength. The exact solution was obtained via the Mie series (see, e.g., [7]). As was the case in 2D, both the arithmetic and geometric mean perform reasonably well, but the geometric mean is generally more accurate than the arithmetic mean. In Fig. 14.11 visible errors are only present near the back-scattering direction of $\phi = 180$

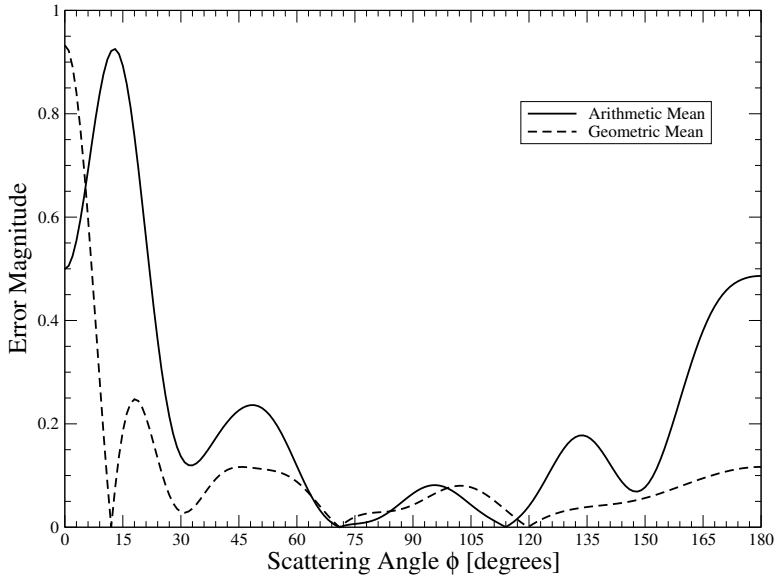


Figure 14.9: Magnitude of the difference between the FDTD-based solutions and the nominally exact solution.

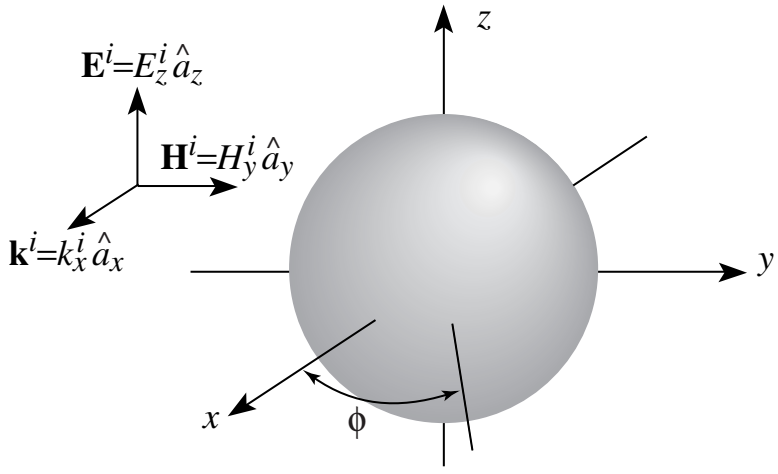


Figure 14.10: Geometry of the dielectric sphere. The relative permittivity ϵ_r is 1.21. This incident field is polarized in the z direction and travels in the x direction. The equatorial angle ϕ is in the xy-plane with $\phi = 0$ corresponding to the $+x$ direction.

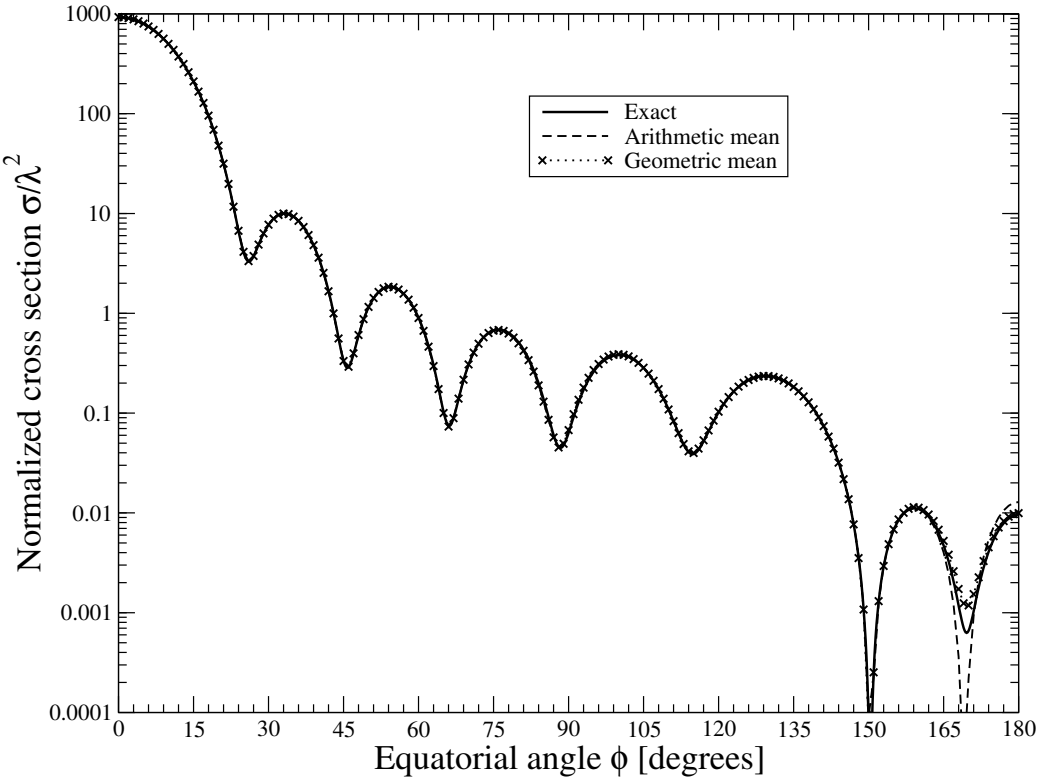


Figure 14.11: Scattering cross section of a dielectric sphere versus the equatorial angle ϕ. The sphere is discretized with 60 cells along the radius and the frequency used here corresponds to 20.06 cells per wavelength.

degrees. Note that there is a large difference, approximately five orders of magnitude, between the scattering in the forward and backward directions.

In order to improve the results in the back-scattering direction Li *et al.* [5] advocated calculating the transformation using the five faces other than the forward-scattering face. Figure 14.12 shows the normalized backscattering cross section versus wavelength (expressed in terms of number of cells per wavelength) calculated using the arithmetic mean. The normalized cross section is given by

$$\frac{1}{\lambda^2} \lim_{r\to\infty} \left[4\pi r^2 \frac{|\hat{E}_z(\theta,\phi)|^2}{|\hat{E}_z^i|^2} \right] \tag{14.88}$$

where r is the distance from the center of the sphere, θ is the azimuthal angle and ϕ is the equatorial angle. For backscatter, θ is $\pi/2$ and ϕ is π.

The NTFF transform results shown in Fig. 14.12 were calculated using either the fields over all six faces of the integration boundary or the fields over the five faces advocated by Li *et al.* The results in this figure correspond to those shown in Fig. 1(b) of [5] for the sphere with a radius of 3 μm. (However, for the sake of generality, here the results are plotted in terms of unitless quantities.) Note that the six-sided arithmetic-mean results presented here are better than those presented in [5]. We were able to duplicate the results presented in [5] by not applying a temporal phase-correction factor (or, similarly, by applying a correction factor which is twice the factor given here). Nevertheless, the recommendation of Li *et al.* is true that the five-sided computation

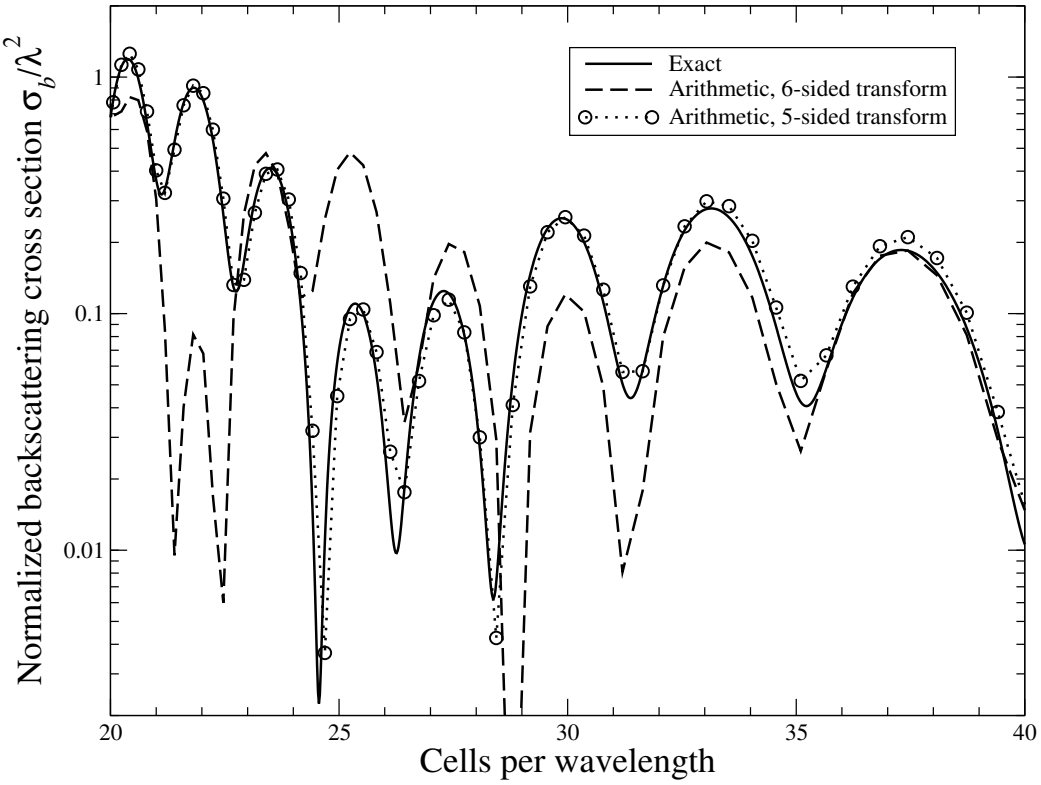


Figure 14.12: Backscatter from a sphere with $\epsilon_r = 1.21$ versus the wavelength (expressed in terms of number of cells). The FDTD transformations are calculated using the arithmetic mean and either a five- or six-sided transformation boundary.

is better than the six-sided one for calculating the backscattering when using the arithmetic mean. However, for directions other than backscatter or for other sizes, one does not know *a priori* if a face should or should not be discarded.

Figure 14.13 is the same as Fig. 14.12 except the transformation is done using the GM-NTFF transform. In this case discarding data from the forward-scattering face actually slightly degrades the quality of the transform. Thus, when using the geometric mean there is no need to discard data. One can use it confidently for all scattering angles and all sizes.

To summarize, unlike the traditional arithmetic mean, for a single harmonic plane wave the geometric mean accounts for the spatial offset of the fields in a way that is exact. In practice, where a spectrum of wave vectors are present, the geometric mean typically performs significantly better than the arithmetic mean. The geometric mean is much less sensitive to the integration-boundary location than is the arithmetic mean. For strongly forward-scattering objects, the use of the geometric mean obviates the need to discard the fields over the forward face (as has been advocated previously) when calculating the backscatter. The geometric mean does entail a slight increase in computational cost because for each node along the integration boundary a DFT must be calculated for three fields instead of two. However, this cost is typically minor compared to the overall simulation cost.

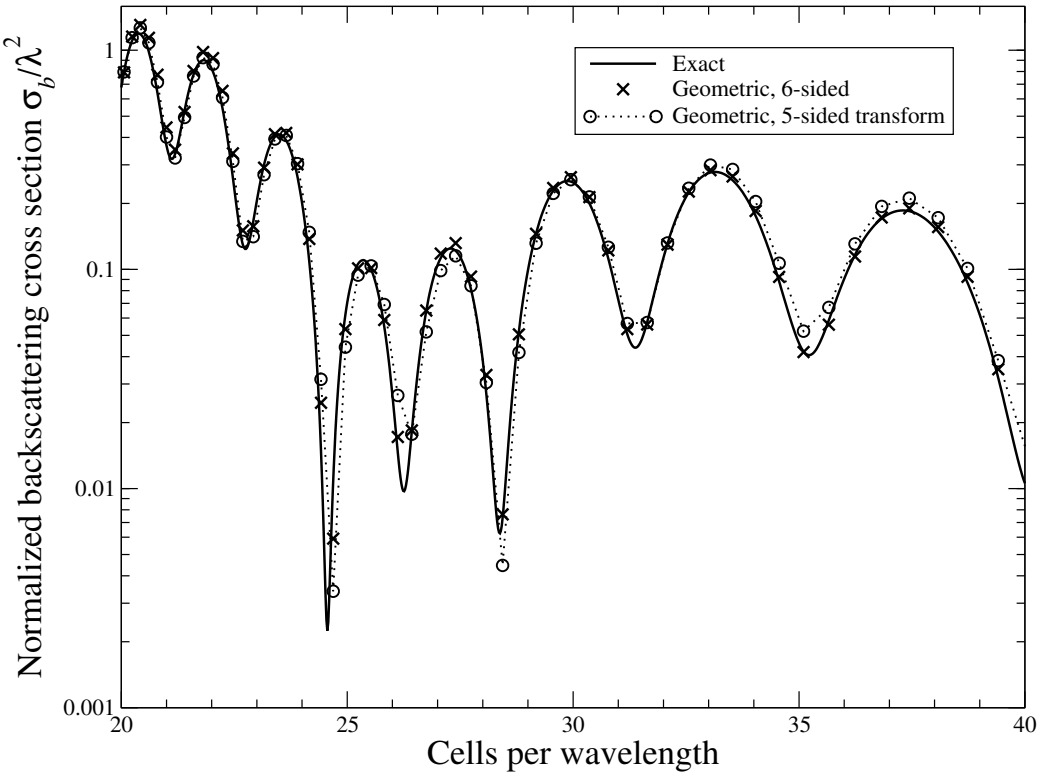


Figure 14.13: Backscatter from a sphere with $\epsilon_r = 1.21$ versus the wavelength (expressed in terms of number of cells). The transformations use the geometric mean and the fields over either five or six faces of the integration boundary.

Appendix A

Construction of Fourth-Order Central Differences

Assuming a uniform spacing of δ between sample points, we seek an approximation of the derivative of a function at x_0 which falls midway between two sample point. Taking the Taylor series expansion of the function at the four sample points nearest to x_0 yields

$$f\left(x_0+\frac{3\delta}{2}\right) = f(x_0)+\frac{3\delta}{2}f'(x_0)+\frac{1}{2!}\left(\frac{3\delta}{2}\right)^2 f''(x_0)+\frac{1}{3!}\left(\frac{3\delta}{2}\right)^3 f'''(x_0)+\ldots, \quad \text{(A.1)}$$

$$f\left(x_0+\frac{\delta}{2}\right) = f(x_0)+\frac{\delta}{2}f'(x_0)+\frac{1}{2!}\left(\frac{\delta}{2}\right)^2 f''(x_0)+\frac{1}{3!}\left(\frac{\delta}{2}\right)^3 f'''(x_0)+\ldots, \quad \text{(A.2)}$$

$$f\left(x_0-\frac{\delta}{2}\right) = f(x_0)-\frac{\delta}{2}f'(x_0)+\frac{1}{2!}\left(\frac{\delta}{2}\right)^2 f''(x_0)-\frac{1}{3!}\left(\frac{\delta}{2}\right)^3 f'''(x_0)+\ldots, \quad \text{(A.3)}$$

$$f\left(x_0-\frac{3\delta}{2}\right) = f(x_0)-\frac{3\delta}{2}f'(x_0)+\frac{1}{2!}\left(\frac{3\delta}{2}\right)^2 f''(x_0)-\frac{1}{3!}\left(\frac{3\delta}{2}\right)^3 f'''(x_0)+\ldots \quad \text{(A.4)}$$

Subtracting (A.3) from (A.2) and (A.4) from (A.1) yields

$$f\left(x_0+\frac{\delta}{2}\right)-f\left(x_0-\frac{\delta}{2}\right) = \delta f'(x_0)+\frac{2}{3!}\left(\frac{\delta}{2}\right)^3 f'''(x_0)+\ldots \quad \text{(A.5)}$$

$$f\left(x_0+\frac{3\delta}{2}\right)-f\left(x_0-\frac{3\delta}{2}\right) = 3\delta f'(x_0)+\frac{2}{3!}\left(\frac{3\delta}{2}\right)^3 f'''(x_0)+\ldots \quad \text{(A.6)}$$

The goal now is to eliminate the term containing $f'''(x_0)$. This can be accomplished by multiplying (A.5) by 27 and then subtracting (A.6). The result is

$$27f\left(x_0+\frac{\delta}{2}\right)-27f\left(x_0-\frac{\delta}{2}\right)-f\left(x_0+\frac{3\delta}{2}\right)+f\left(x_0-\frac{3\delta}{2}\right)=24\delta f'(x_0)+O(\delta^5). \quad \text{(A.7)}$$

Solving for $f'(x_0)$ yields

$$\left.\frac{df(x)}{dx}\right|_{x=x_0}=\frac{9}{8}\frac{f\left(x_0+\frac{\delta}{2}\right)-f\left(x_0-\frac{\delta}{2}\right)}{\delta}-\frac{1}{24}\frac{f\left(x_0+\frac{3\delta}{2}\right)-f\left(x_0-\frac{3\delta}{2}\right)}{\delta}+O(\delta^4). \quad \text{(A.8)}$$

The first term on the right-hand side is the contribution from the sample points nearest x_0 and the second term is the contribution from the next nearest points. The highest-order term not shown is fourth-order in terms of δ.

Appendix B

Generating a Waterfall Plot and Animation

Assume we are interested in plotting multiple snapshots of one-dimensional data. A waterfall plot displays all the snapshots in a single image where each snapshot is offset slightly from the next. On the other hand, animations display one image at a time, but cycle through the images quickly enough so that one can clearly visualize the temporal behavior of the field. Animations are a wonderful way to ascertain what is happening in an FDTD simulation but, since there is no way to put an animation on a piece of paper, waterfall plots also have a great deal of utility.

We begin by discussing waterfall plots. First, the data from the individual frames must be loaded into Matlab. The m-file for a function to accomplish this is shown in Program B.1. This function, `readOneD()`, reads each frame and stores the data into a matrix—each row of which corresponds to one frame. `readOneD()` takes a single argument corresponding to the base name of the frames. For example, issuing the command `z = readOneD('sim');` would create a matrix `z` where the first row corresponded to the data in file `sim.0`, the second row to the data in file `sim.1`, and so on.

Note that there is a `waterfall()` function built into Matlab. One could use that to display the data by issuing the command `waterfall(z)`. However, the built-in command is arguably overkill. Its hidden-line removal and colorization slow the rendering and do not necessarily aide in the visualization of the data.

The plot shown in Fig. 3.9 did not use Matlab's `waterfall()` function. Instead, it was generated using a function called `simpleWaterfall()` whose m-file is shown in Program B.2. This command takes three arguments. The first is the matrix which would be created by `readOneD()`, the second is the vertical offset between successive plots, and the third is a vertical scale factor.

Given these two m-files, Fig. 3.9 was generated using the following commands:

```
z = readOneD('sim');
simpleWaterfall(z, 1, 1.9)  % vertical offset = 1, scale factor = 1.9
xlabel('Space [spatial index]')
ylabel('Time [frame number]')
```

Program B.1 `readOneD.m` Matlab code to read one-dimensional data from a series of frames.

```
function z = readOneD(basename)
%readOneD(BASENAME) Read 1D data from a series of frames.
%  [Z, dataLength, nFrames] = readOneD(BASENAME) Data
%  is read from a series of data files all which have
%  the common base name given by the string BASENAME,
%  then a dot, then a frame index (generally starting
%  with zero).  Each frame corresponds to one row of Z.

% read the first frame and establish length of data
nFrames = 0;
filename = sprintf('%s.%d', basename, nFrames);
nFrames = nFrames + 1;
if exist(filename, 'file')
  z = dlmread(filename, '\n');
  dataLength = length(z);
else
  return;
end

% loop through other frames and break out of loop
% when next frame does not exist
while 1
  filename = sprintf('%s.%d',basename,nFrames);
  nFrames = nFrames + 1;
  if exist(filename, 'file')
    zTmp = dlmread(filename, '\n');
    if length(zTmp) ~= dataLength  % check length matches
      error('Frames have different sizes.')
      break;
    end
    z = [z zTmp]; % append new data to z
  else
    break;
  end
end

% reshape z to appropriate dimensions
z = reshape(z, dataLength, nFrames - 1);
z = z';

return;
```

Program B.2 `simpleWaterfall.m` Matlab function to generate a simple waterfall plot.

```
function simpleWaterfall(z, offset, scale)
%simpleWaterfall Waterfall plot from offset x-y plots.
%  simpleWaterfall(Z, OFFSET, SCALE) Plots each row of z
%  where successive plots are offset from each other by
%  OFFSET and each plot is scaled vertically by SCALE.

hold off                % release any previous plot
plot(scale * z(1, :))   % plot the first row
hold on                 % hold the plot

for i = 2:size(z, 1)    % plot the remaining rows
  plot(scale * z(i, :) + offset * (i - 1))
end

hold off                % release the plot

return
```

A function to generate an animation of one-dimensional data sets is shown in Program B.3. There are multiple ways to accomplish this goal and thus one should keep in mind that Program B.3 is not necessarily the best approach for a particular situation. The function in Program B.3 is called `oneDmovie()` and it takes three arguments: the base name of the snapshots, and the minimum and maximum values of the vertical axis. The function uses a loop, starting in line 25, to read each of the snapshots. Each snapshot is plotted and the plot recorded as a frame of a Matlab "movie" (see the Matlab command `movie()` for further details). The `oneDmovie()` function returns an array of movie frames. As an example, assume the base name is "`sim`" and the user wants the plots to range from -1.0 to 1.0. The following commands would display the animation 10 times (the second argument to the `movie()` command controls how often the movie is repeated):

```
reel = oneDmovie('sim',-1,1);
movie(reel,10)
```

An alternative implementation might read all the data first, as was done with the waterfall plot, and then determine the "global" minimum and maximum values from the data itself. This would free the user from specify those value as `oneDmovie()` currently requires. Such an implementation is left to the interested reader.

Program B.3 `oneDmovie.m` Matlab function which can be used to generate an animation for multiple one-dimensional data sets. For further information on Matlab movies, see the Matlab command `movie`.

```
function reel = oneDmovie(basename, y_min, y_max)
% oneDmovie Create a movie from data file with a common base
%    name which contain 1D data.
```

```
%
% basename = common base name of all files
% y_min    = minimum value used for all frames
% y_max    = maximum value used for all frames
%
% reel = movie which can be played with a command such as:
%          movie(reel, 10)
%    This would play the movie 10 times.  To control the frame
%    rate, add a third argument specifying the desired rate.

% open the first frame (i.e., first data file).
frame = 1;
filename = sprintf('%s.%d', basename, frame);
fid = fopen(filename, 'rt');

% to work around rendering bug under Mac OS X see:
% <www.mathworks.com/support/solutions/
%  data/1-VW0GM.html?solution=1-VW0GM>
figure; set(gcf, 'Renderer', 'zbuffer');

% provided fid is not -1, there is another file to process
while fid ~= -1
  data=fscanf(fid, '%f');    % read the data
  plot(data)                 % plot the data
  axis([0 length(data) y_min y_max]) % scale axes appropriately
  reel(frame) = getframe;   % capture the frame for the movie

  % construct the next file name and try to open it
  frame = frame + 1;
  filename = sprintf('%s.%d', basename, frame);
  fid = fopen(filename, 'rb');
end

return
```

Appendix C

Rendering and Animating Two-Dimensional Data

The function shown below is Matlab code that can be used to generate a movie from a sequence of binary (raw) files. The files (or frames) are assumed to be named such that they share a common base name then have a dot followed by a frame number. Here the frame number is assumed to start at zero. The function can have one, two, or three arguments. This first argument is the base name, the second is the value which is used to normalize all the data, and the third argument specifies the number of decades of data to display. Here the absolute value of the data is plotted in a color-mapped imaged. Logarithmic (base 10) scaling is used so that the value which is normalized to unity will correspond to zero on the color scale and the smallest normalized value will correspond, on the color scale, to the negative of the number of decades (e.g., if the number of decades were three, the smallest value would correspond to -3). This smallest normalized value actually corresponds to a normalized value of 10^{-d} where d is the number of decades. Thus the (normalized) values shown in the output varying from 10^{-d} to 1. The default normalization and number of decades are 1 and 3, respectively.

Program C.1 `raw2movie.m` Matlab function to generate a movie given a sequence of raw files.

```
function reel = raw2movie(basename, z_norm, decades)
% raw2movie Creates a movie from "raw" files with a common base
%      name.
%
% The absolute value of the data is used together with
% logarithmic scaling.  The user may specify one, two, or
% three arguments.
% raw2movie(basename, z_norm, decades) or
% raw2movie(basename, z_norm) or
% raw2movie(basename):
% basename = common base name for all files
% z_norm   = value used to normalize all frames, typically this
%          would be the maximum value for all the frames.
```

```
% Default value is 1.
% decades  = decades to be used in the display.  The normalized
%         data is assumed to vary between 1.0 and 10^(-decades)
%         so that after taking the log (base 10), the values
%         vary between 0 and -decades.  Default value is 3.
%
% return value:
% reel = movie which can be played with a command such as:
%          movie(reel, 10)
%        pcolor() is used to generate the frames.
%
% raw file format:
% The raw files are assumed to consist of all floats (in
% binary format).  The first two elements specify the horizontal
% and vertical dimensions.  Then the data itself is given in
% English book-reading order, i.e., from the upper left corner
% of the image and then scanned left to right.  The frame number
% is assumed to start at zero.

% set defaults if we have less than three arguments
if nargin < 3, decades = 3; end
if nargin < 2, z_norm = 1.0; end

% open the first frame
frame = 0;
filename = sprintf('%s.%d', basename, frame);
fid = fopen(filename, 'rb');

if fid == -1
  error(['raw2movie: initial frame not found: ', filename])
end

% to work around rendering bug under Mac OS X implementation.
figure; set(gcf, 'Renderer', 'zbuffer');

% provided fid is not -1, there is another file to process
while fid ~= -1
  size_x = fread(fid, 1, 'single');
  size_y = fread(fid, 1, 'single');

  data = flipud(transpose(...
           reshape(...
             fread(fid, size_x * size_y, 'single'), size_x, size_y)...
         ));

  % plot the data
  if decades ~= 0
```

```
61      pcolor(log10(abs((data + realmin) / z_norm)))
62      shading flat
63      axis equal
64      axis([1 size_x 1 size_y])
65      caxis([-decades 0])
66      colorbar
67    else
68      pcolor(abs((data + realmin) / z_norm))
69      shading flat
70      axis equal
71      axis([1 size_x 1 size_y])
72      caxis([0 1])
73      colorbar
74    end
75
76    % capture the frame for the movie (Matlab wants index to start
77    % at 1, not zero, hence the addition of one to the frame)
78    reel(frame + 1) = getframe;
79
80    % construct the next file name and try to open it
81    frame = frame + 1;
82    filename = sprintf('%s.%d', basename, frame);
83    fid = fopen(filename,' rb');
84
85  end
```

Appendix D

Notation

c	speed of light *in free space*
N_{freq}	index of spectral component corresponding to a frequency with discretization N_λ
N_λ	number of points per wavelength for a given frequency (the wavelength is the one pertaining to propagation in free space)
N_L	number of points per skin depth (L for loss)
N_P	number of points per wavelength at peak frequency of a Ricker wavelet
N_T	number of time steps in a simulation
S_c	Courant number ($c\Delta_t/\Delta_x$ in one dimension)
s_t	Temporal shift operator
s_x	Spatial shift operator (in the x direction)

Appendix E

PostScript Primer

E.1 Introduction

PostScript was developed by Adobe Systems Incorporated and is both a page-description language and a programming language. Unlike a JPEG or GIF file which says what each pixel in an image should be, PostScript is "vector based" and thus a PostScript file specifies graphic primitives, such as lines and arcs. There primitives are described by various PostScript commands. The quality of the image described by a PostScript file will depend on the output device. For example, a laser printer with 1200 dots per inch will draw a better curved line than would a laser printer with 300 dots per inch (although both will typically produce very good output).

The goal here is to show how one can easily generate PostScript files which convey information about an FDTD simulation. Thus we are more interested in the page-description aspects of PostScript rather than its programming capabilities. (There is a wonderful book and Web site by Bill Casselman that describe PostScript extremely well while illustrating a host of mathematical concepts. The book is entitled *Mathematical Illustrations: A Manual of Geometry and PostScript* which you can find at `www.math.ubc.ca/~cass/graphics/manual/`. It is well worth checking out.)

PostScript is a Forth-like language in that it uses what is known as postfix notation. If you have used an RPN (reverse Polish notation) calculator, you are familiar with postfix notation. You put arguments onto a "stack" and then select an operation which "pops" the arguments from the stack and operates on them. For example, to add 3 and 12 you would enter the following:

```
3
<ENTER>
12
+
```

When 3 is typed on the keypad, it is placed at the top of the stack. It is pushed onto the next stack location by hitting the `ENTER` key. When 12 is typed, it is put at the top of the stack. Hitting the plus sign tells the calculator you want to add the top two numbers on the stack, i.e., the 3 and 12. These numbers are popped (i.e., taken) from the stack, and the result of the addition (15) is placed at the top of the stack.

The PostScript language is much like this. Arguments are given before the operations. Giving arguments before operations facilitates the construction of simple interpreters. PostScript interpreters typically have the task of translating the commands in a PostScript file to some form of viewable graphics. For example, there are PostScript printers which translate (interpret) a PostScript file into a printed page. Most computers have PostScript interpreters which permit the graphics described in a PostScript file to be displayed on the screen. There are free PostScript interpreters available via the Web (you should do a search for GhostScript if you are in need of an interpreter).

E.2 The PostScript File

A file which contains PostScript commands, which we will call a PostScript file, is a plain ASCII file which must start with "`%!PS`". These characters are often referred to as a "magic word." Magic words appear at the start of many computer files and identify the contents of the file. This `%!PS` magic word identifies the contents of the file as PostScript to the interpreter. (The names of PostScript file often end with the suffix `.ps`, but the interpreter does not care what the file name is.) The last command in a PostScript file is typically `showpage`. This command essentially tells the interpreter that all the commands have been given and the page (or screen image or whatever) should be rendered.

What comes between `%!PS` and `showpage` are the commands which specify how the page should appear. Before exploring some of these commands it is important to know that a PostScript interpreter, by default, thinks in terms of units of "points" which are not points in the geometric sense, but rather $1/72$ of an inch. Points are a traditional unit used in the printing industry (thus a "12-point font" is one for which a typical capital letter is $12/72$ of an inch high). A default "page" is 8.5 by 11 inches and thus 612 by 792 points. The origin is the lower left corner of the page.

E.3 PostScript Basic Commands

The PostScript command `moveto` takes two arguments: the x and y coordinates to which the current point should be moved. You can think of the current point as akin to the point where the tip of a pen is moved. To define a line we can give the command `lineto`. `lineto` also takes two arguments: the x and y coordinates of the point to which the line should be drawn. In PostScript, after issuing the `lineto` command we have merely defined the path of the line—we have not actually drawn anything yet. You can think of this as the pen having drawn the line in invisible ink. We have to issue one more command to make the line visible, the `stroke` command.

A complete PostScript file (which we will identify as a "Program") which draws a line from the point $(100, 200)$ to the point $(300, 600)$ is shown in Program E.1.

Program E.1 PostScript commands to draw a single tilted line.

```
%!PS
100 200 moveto
```

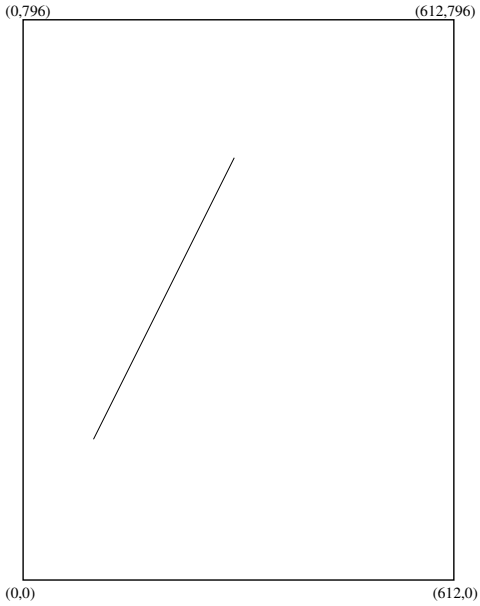


Figure E.1: Simple line rendered by the PostScript commands giving in Program E.1 and E.2. The surrounding box and corner labels have been added for the sake of clarity.

```
300 600 lineto
stroke
showpage
```

The image drawn by these commands is shown in Fig. E.1. The surrounding border and coordinate labels have been added for clarity. The only thing which would actually be rendered is the tilted line shown within the border.

Instead of using the command `lineto` to specify the point to which we want the line to be drawn, the command `rlineto` can be used where now the arguments specify the relative movement from the current point (hence the "r" for relative). The arguments of `rlineto` specify the relative displacement from the current point. In the commands in Program E.1, the line which was drawn went 200 points in the x direction and 400 points in the y direction from the starting point. Thus instead of writing `300 600 lineto`, one could obtain the same result using `200 400 rlineto`. PostScript does not care about whitespace and the percent sign is used to indicate the start of a comment (the interpreter ignores everything from the percent sign to the end of the line). Thus, another file which would also yield the output shown in Fig. E.1 is shown in Program E.2. (The magic word must appear by itself on the first line of the file.)

Program E.2 PostScript commands to draw a single tilted line. Here the `rlineto` command is used. The resulting image is identical to the one produced by Progam E.1 and is shown in Fig. E.1.

```
%!PS
% tilted line using the rlineto command
100 200 moveto 300 600 lineto stroke showpage
```

When creating graphics it is often convenient to redefine the origin of the coordinate system to a point which is more natural to the object being drawn. PostScript allows us to translate the origin to any point on the page using the `translate` command. This command also takes two arguments corresponding to the point in the current coordinate system where the new origin should be located. For example, let us assume we want to think in terms of both positive and negative coordinates. Therefore we wish to place the origin in the middle of the page. This can be accomplished with the command `306 396 translate`. The PostScript commands shown in Program E.3 demonstrate the use of the `translate` command.

Program E.3 PostScript commands which first translate the origin to the center of the page and then draw four lines which "radiate away" from the origin. The corner labels show the corner coordinates after the translation of the origin to the center of the page.

```
%!PS
306 398 translate  % translate origin to center of page
 100  100 moveto  50  50 rlineto stroke
-100  100 moveto -50  50 rlineto stroke
-100 -100 moveto -50 -50 rlineto stroke
 100 -100 moveto  50 -50 rlineto stroke
showpage
```

Program E.3 yields the results shown in Fig. E.2.

As you might imagine, thinking in terms of units of points ($1/72$ of an inch) is not always convenient. PostScript allows us to scale the dimensions by any desired value using the `scale` command. In fact, one can use a different scale factor in both the x and the y directions and thus `scale` takes two arguments. However, we will stick to using equal scaling in both directions.

In the previous example, all the locations were specified in terms of multiples of 50. Therefore it might make sense to scale the dimensions by a factor of 50 (in both the x and y direction). This scaling should be done *after* the translation of the origin. We might anticipate that the commands shown in Program E.4 would render the same output as shown in Fig. E.2.

Program E.4 PostScript file where the units are scaled by a factor of 50 in both the x and y dimensions.

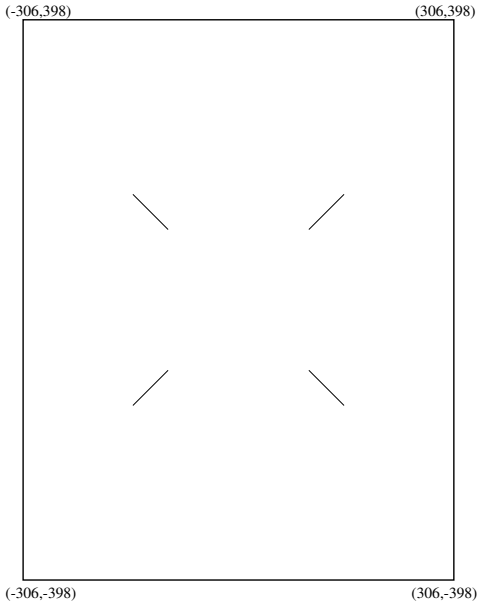


Figure E.2: Output rendered by Program E.3 which translates the origin to the center of the page. This output is also produced by Program E.5 and Program E.6.

```
%!PS
306 398 translate
50 50 scale % scale units in x and y direction by 50
 2  2 moveto  1  1 rlineto stroke
-2  2 moveto -1  1 rlineto stroke
-2 -2 moveto -1 -1 rlineto stroke
 2 -2 moveto  1 -1 rlineto stroke
showpage
```

However this file yields the output shown in Fig. E.3. Note that the lines which radiate from origin are now much thicker. In fact, they are 50 times thicker than they were in the previous image. By default, a line in PostScript has a thickness of unity i.e., one point. The `scale` command scaled the line thickness along with all the other dimensions so that now the line thickness is 50 points.

Although we have only given integer dimensions so far, as far as PostScript is concerned all values are actually real numbers (i.e., floating-point numbers). We can control the line thickness with the `setlinewidth` command which takes a single argument. If we want the line thickness still to be one point, the line thickness should be set to the inverse of the scale factor, i.e., $1/50 = 0.02$. Also, it is worth noting that the stroke command does not have to be given after each drawing command. We just have to ensure that it is given before the end of the file (or before the line style changes to something else). Thus, a PostScript file which scales the dimensions by 50 and produces the same output as shown in Fig. E.2 is shown in Program E.4

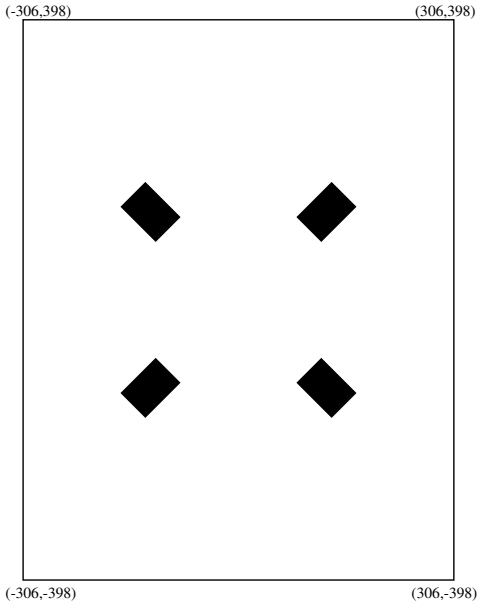


Figure E.3: Output rendered by Program E.4 which scales the units by 50.

Program E.5 PostScript file where the units are scaled by a factor of 50 and the line thickness is corrected to account for this scaling. Note that a single `stroke` command is given.

```
%!PS
306 398 translate
50 50 scale
0.02 setlinewidth  % correct line thickness to account for scaling
 2  2 moveto  1  1 rlineto
-2  2 moveto -1  1 rlineto
-2 -2 moveto -1 -1 rlineto
 2 -2 moveto  1 -1 rlineto stroke
showpage
```

PostScript permits the use of named variables and, as we shall see, named procedures. This is accomplished using the `def` command which takes, essentially, two arguments: the first being the literal string which is the variable or procedure name and the second being the value or procedure (where the procedure would be enclosed in braces). A literal string is a backslash character followed by the string. For example, the following sets the variable `scalefactor` to 50:

```
/scalefactor 50 def
```

After issuing this command, we can use `scalefactor` in place of 50 everywhere in the file.

The PostScript language includes a number of mathematical functions. One can add using `add`, subtract using `sub`, multiply using `mul`, and divide using `div`. Each of these functions

takes two arguments consistent with an RPN calculator. To calculate the inverse of 50, one could issue the following command:

```
1 50 div
```

This places 1 on the stack, then 50, and then divides the two. The result, 0.02, remains at the top of the stack.

The program shown in Program E.6 uses the `def` and `div` commands and is arguably a bit cleaner and better self-documenting than the one shown in Program E.5. Program E.6 also produces the output shown in Fig. E.2.

Program E.6 PostScript file which uses the `def` command to define a scale-factor which is set to 50. The inverse of the scale-factor is obtained by using the `div` command to divide 1 by the scale-factor.

```
%!PS
306 398 translate
% define "scalefactor" to be 50
/scalefactor 50 def
% scale x and y directions by the scale factor
scalefactor scalefactor scale
% set line width to inverse of the scale factor
1 scalefactor div setlinewidth
 2  2 moveto  1  1 rlineto
-2  2 moveto -1  1 rlineto
-2 -2 moveto -1 -1 rlineto
 2 -2 moveto  1 -1 rlineto stroke
showpage
```

The `arc` command takes five arguments: the x and y location of the center of the arc, the radius of the arc, and the angles (in degrees) at which the arc starts and stops. For example, the following command would draw a complete circle of radius 0.5 about the point $(2, 2)$:

```
2 2 0.5 0 360 arc stroke
```

Let us assume we wish to draw several circles, each of radius 0.5. We only wish to change the center of the circle. Rather than specifying the `arc` command each time with all its five arguments, we can use the `def` command to make the program more compact. Consider the program shown in Program E.7. Here the `def` command is used to say that the literal `circle` is equivalent to `0.5 0 360 arc stroke`, i.e., three of the arguments are given to the `arc` command—one just has to provide the two missing arguments which are the x and y location of the center of the circle. The output produced by this program is shown in Fig. E.4.

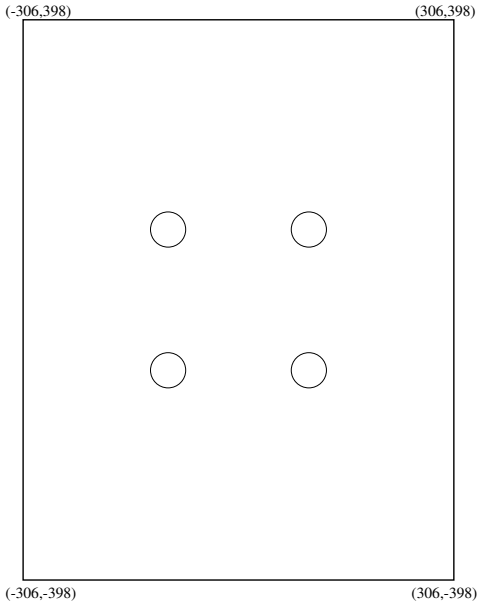


Figure E.4: Output rendered by Program E.7.

Program E.7 PostScript file which renders the output shown in Fig. E.4.

```
%!PS
306 398 translate
/scalefactor 50 def
scalefactor scalefactor scale
1 scalefactor div setlinewidth
/circle {0.5 0 360 arc stroke} def
 2  2 circle
-2  2 circle
-2 -2 circle
 2 -2 circle
showpage
```

In addition to `stroke`-ing a path, PostScript allows paths to be `fill`-ed using the fill command. So, instead of drawing a line around the perimeter of the circles shown in Fig. E.4, one can obtain filled circles by issuing the `fill` command instead of the `stroke` command. Program E.8 and the corresponding output shown in Fig. E.5 illustrate this.

Program E.8 PostScript file which defines a `stroke`-ed and `fill`-ed circle. The corresponding output is shown in Fig. E.5.

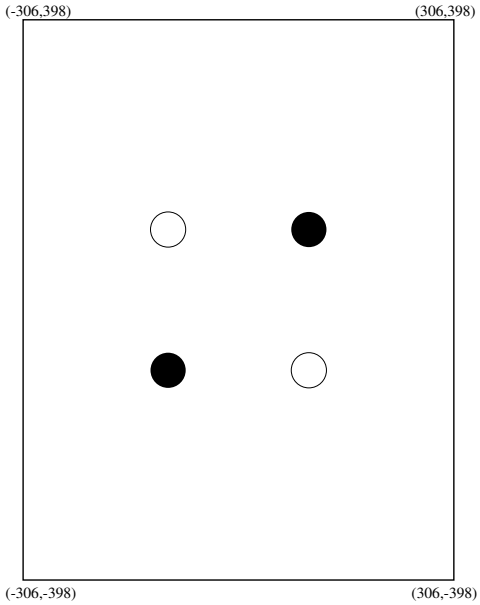


Figure E.5: Output rendered by Program E.8.

```
%!PS
306 398 translate
/scalefactor 50 def
scalefactor scalefactor scale
1 scalefactor div setlinewidth
/circle {0.5 0 360 arc stroke} def
/circlef {0.5 0 360 arc fill} def
 2  2 circlef
-2  2 circle
-2 -2 circlef
 2 -2 circle
showpage
```

The PostScript commands we have considered are shown in Table E.1.

Instead of using an editor to write a PostScript file directly, we can use another program to generate the PostScript file for us. Specifically, let us consider a C program which generates a PostScript file. This program is supposed to demonstrate how one could use PostScript to display a particular aspect of an FDTD grid. For example, let us assume we are using a TM^z grid which is 21 cells by 21 cells. There is a PEC cylinder with a radius of 5 which is centered in the grid. We know that the E_z nodes which fall within the cylinder should be set to zero and this zeroing operation would be done with a for-loop. However, precisely which nodes are being set to zero? The code shown in Program E.9 could be incorporated into an FDTD program. This code produces the PostScript output file `grid.ps` which renders the image shown in Fig. E.6. The first several lines of the file `grid.ps` are shown in Fragment E.10.

Command	Description
$x\ y$ `moveto`	move current point to (x, y)
$x\ y$ `lineto`	draw a line from current point to (x, y)
$\delta_x\ \delta_y$ `rlineto`	from current point draw a line over δ_x and up δ_y
$x\ y$ `translate`	translate the origin to the point (x, y)
$s_x\ s_y$ `scale`	scale x and y coordinates by s_x and s_y
`stroke`	"apply ink" to a previously defined path
`fill`	fill the interior of a previously defined path
w `setlinewidth`	set the line width to w
$d_1\ d_2$ `div`	calculate d_1/d_2; result is placed at top of stack
$x_c\ y_c\ r\ a_1\ a_2$ `arc`	draw an arc of radius r centered at (x_c, y_c) starting at angle a_1 and ending at angle a_2 (degrees)
/literal {*definition*} `def`	define the *literal* string to have the given *definition*; braces are needed if the definition contains any whitespace

Table E.1: An assortment of PostScript commands and their arguments.

Program E.9 C program which generates a PostScript file. The file draws either a cross or a filled circle depending on whether a node is outside or inside a circular boundary, respectively. The rendered image is shown in Fig. E.6.

```
/* C program to generate a PostScript file which draws a cross
 * if a point is outside of a circular boundary and draws a
 * filled circle if the point is inside the boundary.
 */

#include <stdio.h>
#include <math.h>

int is_inside_pec(double x, double y);

int main() {
  int m, n;

  FILE *out;

  out = fopen("grid.ps","w");  // output file is "grid.ps"

  /* header material for PostScript file */
  fprintf(out,"%%!PS\n"
          "306 396 translate\n"
          "/scalefactor 20 def\n"
          "scalefactor scalefactor scale\n"
          "1 scalefactor div setlinewidth\n"
```

```
24            "/cross {moveto\n"
25            "         -.2 0 rmoveto .4 0 rlineto\n"
26            "         -.2 -.2 rmoveto 0 .4 rlineto stroke} def\n"
27            "/circle {.2 0 360 arc fill} def\n"
28            );
29
30    for (m=-10; m<=10; m++)
31      for (n=-10; n<=10; n++)
32        if (is_inside_pec(m,n)) {
33          fprintf(out,"%d %d circle\n",m,n);
34        } else {
35          fprintf(out,"%d %d cross\n",m,n);
36        }
37
38    fprintf(out,"showpage\n");
39
40    return 0;
41  }
42
43  /* Function returns 1 if point (x,y) is inside a circle (or on
44   * the perimeter of circle) and returns 0 otherwise.
45   */
46  int is_inside_pec(double x, double y) {
47    double radius = 5.0;
48
49    return x*x + y*y <= radius*radius;
50  }
```

Fragment E.10 First several lines of the file `grid.ps` which is produced by Program E.9.

```
%!PS
306 396 translate
/scalefactor 20 def
scalefactor scalefactor scale
1 scalefactor div setlinewidth
/cross {moveto
         -.2 0 rmoveto .4 0 rlineto
         -.2 -.2 rmoveto 0 .4 rlineto stroke} def
/circle {.2 0 360 arc fill} def
-10 -10 cross
-10 -9 cross
-10 -8 cross
-10 -7 cross
```

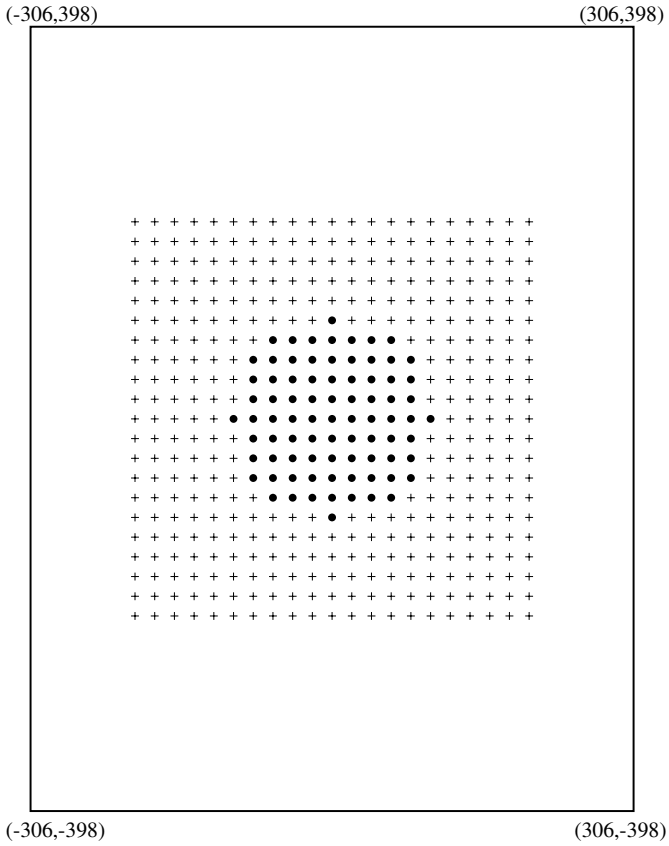


Figure E.6: Grid depiction rendered by the file `grid.ps` which is produced by Program E.9. Crosses corresponds to nodes which are outside a circular boundary of radius 5. Filled circles correspond to nodes inside the boundary (or identically on the perimeter of the boundary).

```
-10 -6 cross
     .
     .
     .
```

Bibliography

[1] D. Sullivan and J. L. Young. Far-field time-domain calculation from aperture radiators using the FDTD method. *IEEE Transactions on Antennas and Propagation*, 49(3):464–469, March 2001.

[2] S. Dey and R. Mittra. A locally conformal finite-difference time-domain (FDTD) algorithm for modeling three-dimensional perfectly conducting objects. *IEEE Microwave Guided Wave Letters*, 7(9):273–275, September 1997.

[3] S. Benkler, N. Chavannes, and N. Kuster. A new 3-D conformal PEC FDTD scheme with user-defined geometric precision and derived stability criterion. *IEEE Transactions on Antennas and Propagation*, 54(6):1843–1849, June 2006.

[4] C. A. Balanis. *Advanced Engineering Electromagnetics*. John Wiley & Sons, New York, 1989.

[5] X. Li, A. Taflove, and V. Backman. Modified FDTD near-to-far-field transformation for improved backscattering calculation of strongly forward-scattering objects. *IEEE Antennas and Wireless Propagation Letters*, 4:35–38, 2005.

[6] A. Taflove and S. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method, 3 ed.* Artech House, Boston, MA, 2005.

[7] A. Ishimaru. *Electromagnetic Wave Propagation, Radiation, and Scattering*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.