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

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

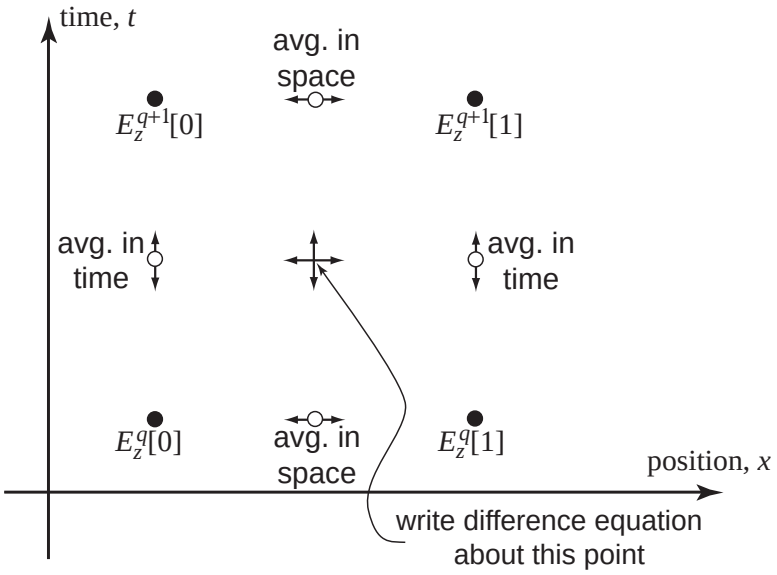

Figure 6.1: Space-time in the neighborhood of the left end of the grid. Only the electric fields are shown. The open circles indicated where an electric field is needed for the advection equation. Since there are none there, averaging is used to approximate the field at those points.

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.

```
/* Function to initialize the Grid structure. */

#include "fdtd3.h"

#define EPSR 9.0

void gridInit(Grid *g) {
  double imp0 = 377.0;
  int mm;

  SizeX = 200;   // size of domain
  MaxTime = 450; // duration of simulation
  Cdtds = 1.0;   // Courant number

  ALLOC_1D(g->ez,   SizeX, double);
  ALLOC_1D(g->ceze, SizeX, double);
  ALLOC_1D(g->cezh, SizeX, double);
  ALLOC_1D(g->hy,   SizeX - 1, double);
  ALLOC_1D(g->chyh, SizeX - 1, double);
  ALLOC_1D(g->chye, SizeX - 1, double);

  /* set electric-field update coefficients */
  for (mm = 0; mm < SizeX; mm++)
    if (mm < 100) {
      Ceze(mm) = 1.0;
      Cezh(mm) = imp0;
    } else {
      Ceze(mm) = 1.0;
      Cezh(mm) = imp0 / EPSR;
    }

```

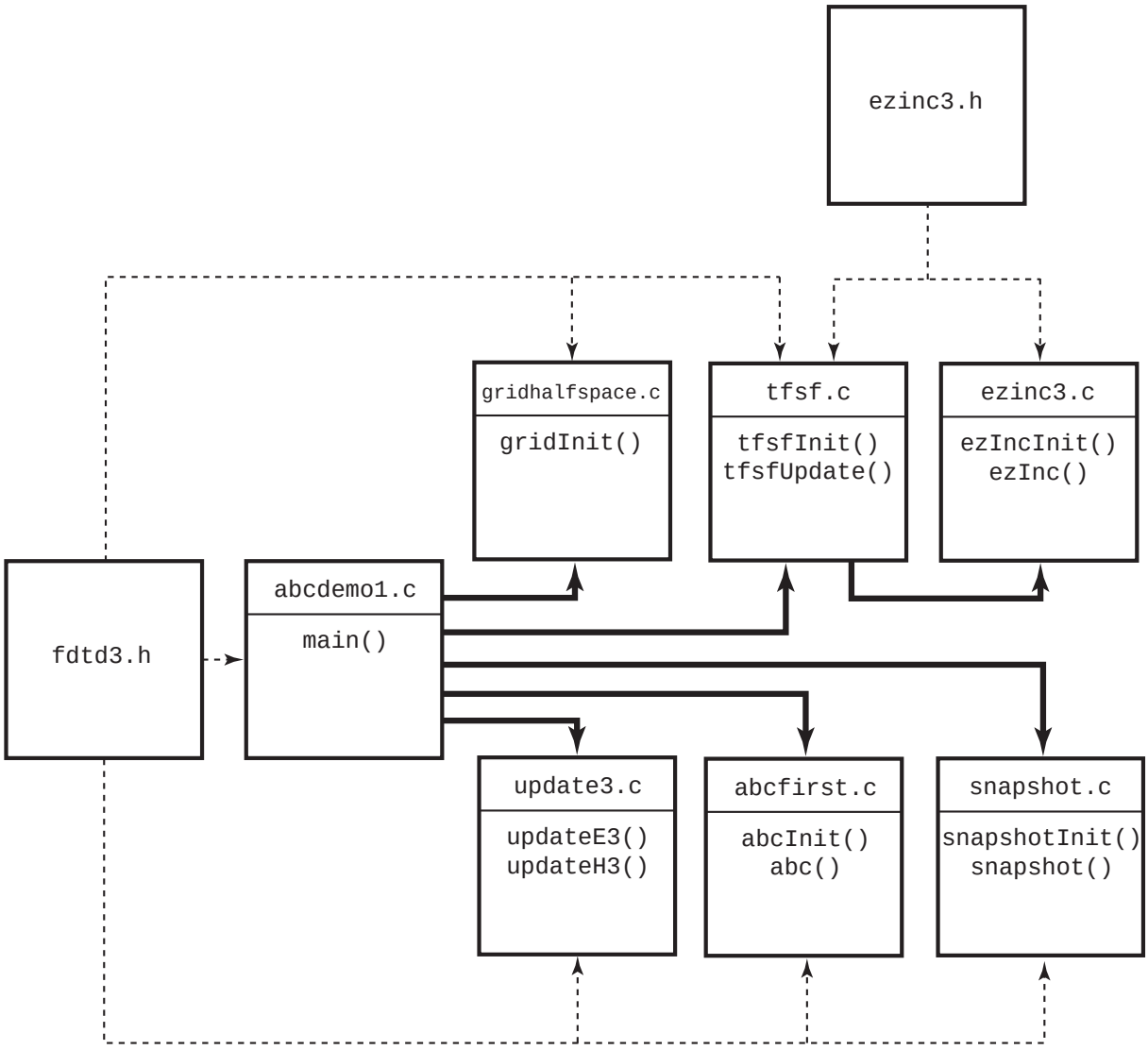

Figure 6.2: Files associated with the modular implementation of a simulation of a dielectric half space. Since a first-order ABC is now being used, some of these files differ from those of Fig. 4.3. Nevertheless, the overall structure of the program is unchanged and all the function names are the same 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
```

```
    updateE3(g);   // update electric field
    abc(g);         // apply ABC -- after E-field update
    snapshot(g);   // take a snapshot (if appropriate)

  } /* end of time-stepping */

  return 0;
}
```

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.

```
/* Function to implement a first-order ABC. */

#include "fdtd3.h"
#include <math.h>

static int initDone = 0;
static double ezOldLeft = 0.0, ezOldRight = 0.0;
static double abcCoefLeft, abcCoefRight;

/* Initialization function for first-order ABC. */
void abcInit(Grid *g) {
  double temp;

  initDone = 1;

  /* calculate coefficient on left end of grid */
  temp = sqrt(Cezh(0) * Chye(0));
  abcCoefLeft = (temp - 1.0) / (temp + 1.0);

  /* calculate coefficient on right end of grid */
  temp = sqrt(Cezh(SizeX - 1) * Chye(SizeX - 2));
  abcCoefRight = (temp - 1.0) / (temp + 1.0);

  return;
}

/* First-order ABC. */
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

$$IE_z^{q+1}[m] = E_z^{q+1}[m]\,, \tag{6.16}$$

$$s_x^1 E_z^{q+1}[m] = E_z^{q+1}[m+1]\,, \tag{6.17}$$

$$s_t^{-1} E_z^{q+1}[m] = E_z^{q}[m]\,. \tag{6.18}$$

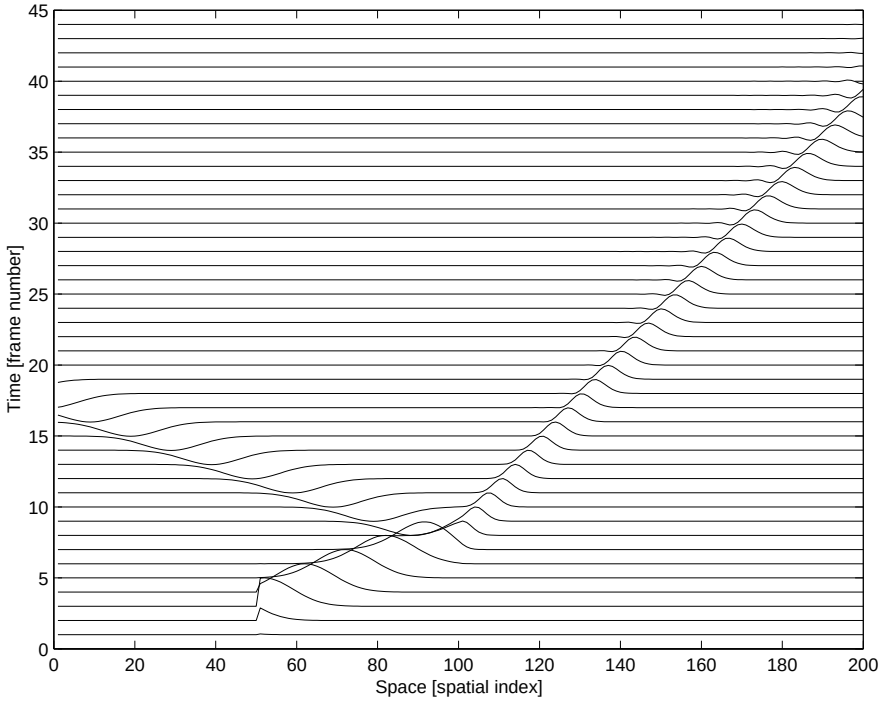

Figure 6.3: Waterfall plot of the snapshots generated by Program 6.2. Comparing this to Fig. 3.13 one sees they are the same until the transmitted field encounters the right end of the grid. At that point 3.13 shows there is a reflected field. However, because of the ABC used here, no reflected field 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.

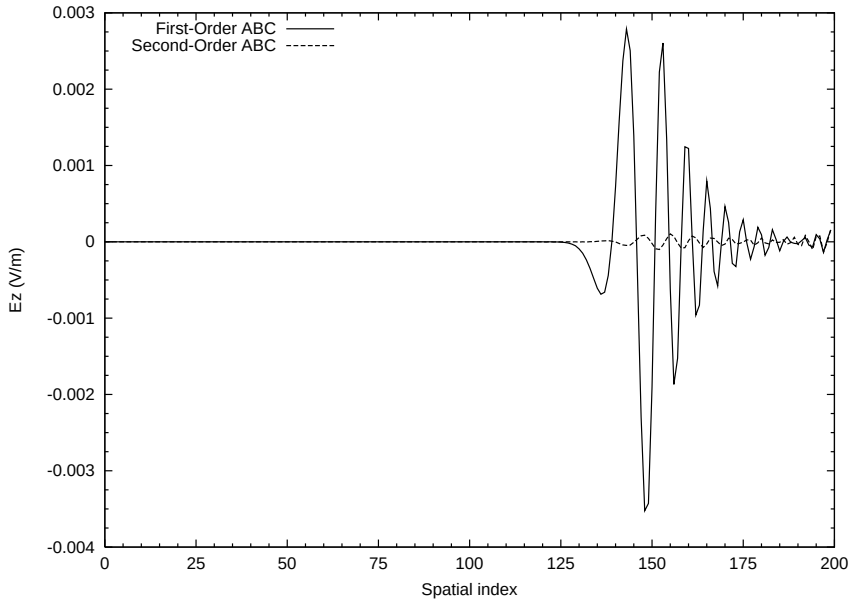

Figure 6.4: Plot of the fields at time step 550 when the grid is terminated with either a first- or second-order ABC. This snapshot is taken after the transmitted pulse has encountered the right edge of the grid. Hence this shows the field reflected by the ABC. Ideally the reflected fields should be 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 /* Initization 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.