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`

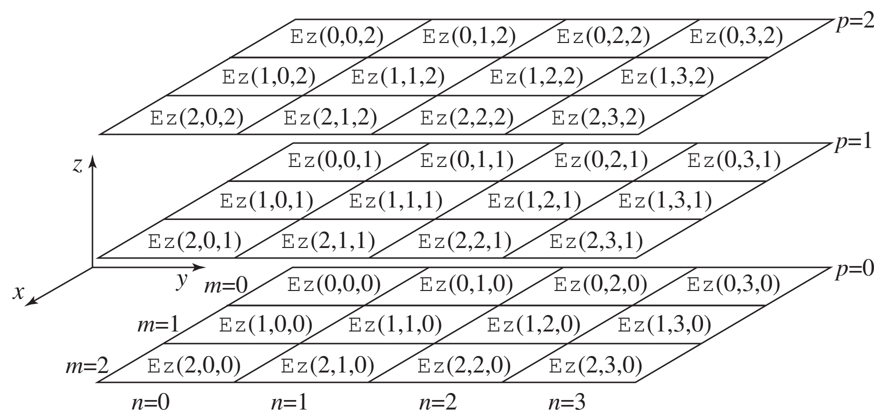


Figure 9.1: Depiction of elements of an array with dimensions $3 \times 4 \times 3$ in the x, y, and z directions, respectively. The indices m, n, and p, are used to specify the x, y, and z locations, respectively. The element at the "origin" has indices $(0, 0, 0)$ and is shown in the upper left corner 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()`.

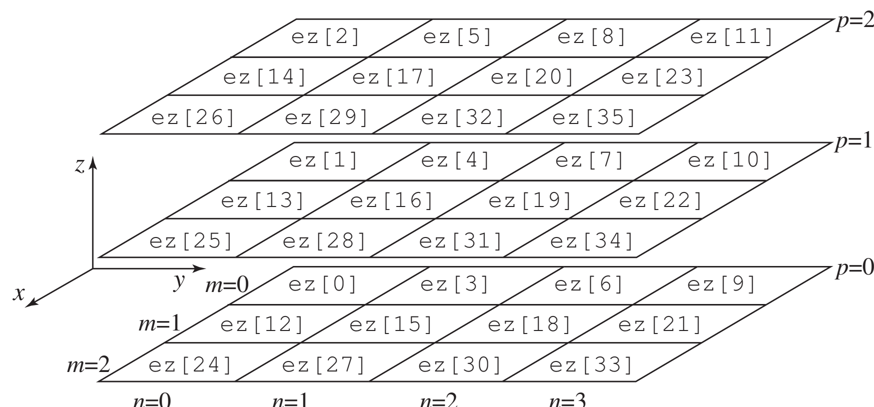


Figure 9.2: The 1D array `ez` is used to store the elements of `Ez`. The three indices for each elements of `Ez` shown in Fig. 9.1 map to the single index shown here.

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:

$$H_x(x,y,z,t) = H_x(m\Delta_x, n\Delta_y, p\Delta_z, q\Delta_t) = H_x^q[m,n,p], \tag{9.3}$$
$$H_y(x,y,z,t) = H_y(m\Delta_x, n\Delta_y, p\Delta_z, q\Delta_t) = H_y^q[m,n,p], \tag{9.4}$$
$$H_z(x,y,z,t) = H_z(m\Delta_x, n\Delta_y, p\Delta_z, q\Delta_t) = H_z^q[m,n,p], \tag{9.5}$$
$$E_x(x,y,z,t) = E_x(m\Delta_x, n\Delta_y, p\Delta_z, q\Delta_t) = E_x^q[m,n,p], \tag{9.6}$$
$$E_y(x,y,z,t) = E_y(m\Delta_x, n\Delta_y, p\Delta_z, q\Delta_t) = E_y^q[m,n,p], \tag{9.7}$$
$$E_z(x,y,z,t) = E_z(m\Delta_x, n\Delta_y, p\Delta_z, q\Delta_t) = E_z^q[m,n,p]. \tag{9.8}$$

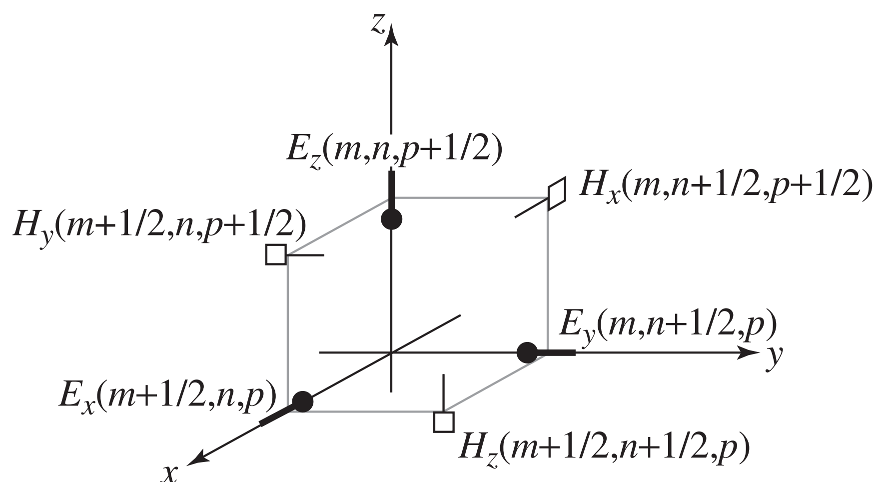


Figure 9.3: Arrangement of nodes in three dimensions. In a computer program all these nodes would have the same m, n, and p indices (the one-halves would be discarded from the equations—the offset would be understood). Electric-field nodes are displaced a half step in the direction in which they point 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)

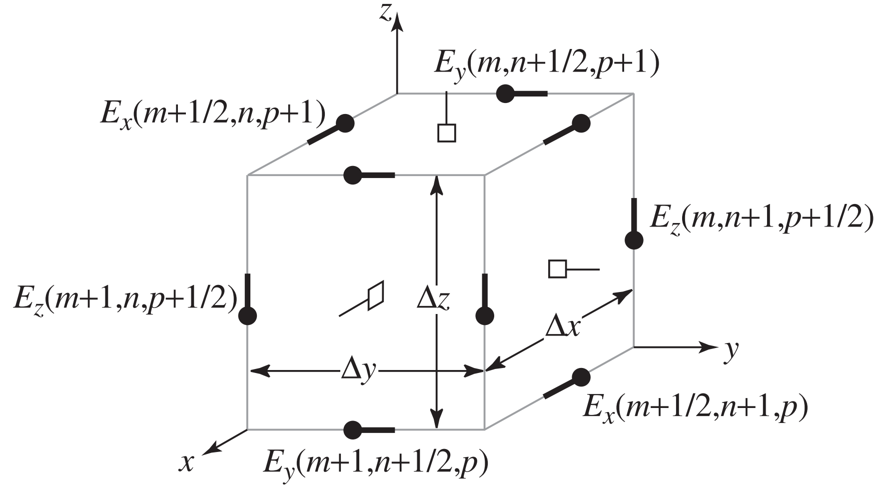


Figure 9.4: The nodes in a 3D FDTD grid are often drawn in the form of a Yee cube or Yee cell. In this depiction the nodes do not all have the same indices. As drawn here the cube would consist of four E_x nodes, four E_y nodes, and four E_z nodes, i.e., the electric fields are along the cube edges. 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

$$-\sigma_m H_x - \mu\frac{\partial H_x}{\partial t} = \left.\frac{\partial E_z}{\partial y} - \frac{\partial E_y}{\partial z}\right|_{x=m\Delta_x,y=(n+1/2)\Delta_y,z=(p+1/2)\Delta_z,t=q\Delta_t}, \tag{9.9}$$

$$-\sigma_m H_y - \mu\frac{\partial H_y}{\partial t} = \left.\frac{\partial E_x}{\partial z} - \frac{\partial E_z}{\partial x}\right|_{x=(m+1/2)\Delta_x,y=n\Delta_y,z=(p+1/2)\Delta_z,t=q\Delta_t}, \tag{9.10}$$

$$-\sigma_m H_z - \mu\frac{\partial H_z}{\partial t} = \left.\frac{\partial E_y}{\partial x} - \frac{\partial E_x}{\partial y}\right|_{x=(m+1/2)\Delta_x,y=(n+1/2)\Delta_y,z=p\Delta_z,t=q\Delta_t}, \tag{9.11}$$

$$\sigma E_x + \epsilon\frac{\partial E_x}{\partial t} = \left.\frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z}\right|_{x=(m+1/2)\Delta_x,y=n\Delta_y,z=p\Delta_z,t=(q+1/2)\Delta_t}, \tag{9.12}$$

$$\sigma E_y + \epsilon\frac{\partial E_y}{\partial t} = \left.\frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x}\right|_{x=m\Delta_x,y=(n+1/2)\Delta_y,z=p\Delta_z,t=(q+1/2)\Delta_t}, \tag{9.13}$$

$$\sigma E_z + \epsilon\frac{\partial E_z}{\partial t} = \left.\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y}\right|_{x=m\Delta_x,y=n\Delta_y,z=(p+1/2)\Delta_z,t=(q+1/2)\Delta_t}. \tag{9.14}$$

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

$$\begin{aligned} H_x^{q+\frac{1}{2}}\left[m, n+\frac{1}{2}, p+\frac{1}{2}\right] &= \frac{1-\frac{\sigma_m \Delta_t}{2\mu}}{1+\frac{\sigma_m \Delta_t}{2\mu}} H_x^{q-\frac{1}{2}}\left[m, n+\frac{1}{2}, p+\frac{1}{2}\right] \\ &+ \frac{1}{1+\frac{\sigma_m \Delta_t}{2\mu}} \left(\frac{\Delta_t}{\mu \Delta_z} \left\{ E_y^q\left[m, n+\frac{1}{2}, p+1\right] - E_y^q\left[m, n+\frac{1}{2}, p\right] \right\} \right. \\ &\left. - \frac{\Delta_t}{\mu \Delta_y} \left\{ E_z^q\left[m, n+1, p+\frac{1}{2}\right] - E_z^q\left[m, n, p+\frac{1}{2}\right] \right\} \right), \end{aligned} \tag{9.15}$$

$$\begin{aligned} H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}, n, p+\frac{1}{2}\right] &= \frac{1-\frac{\sigma_m \Delta_t}{2\mu}}{1+\frac{\sigma_m \Delta_t}{2\mu}} H_y^{q-\frac{1}{2}}\left[m+\frac{1}{2}, n, p+\frac{1}{2}\right] \\ &+ \frac{1}{1+\frac{\sigma_m \Delta_t}{2\mu}} \left(\frac{\Delta_t}{\mu \Delta_x} \left\{ E_z^q\left[m+1, n, p+\frac{1}{2}\right] - E_z^q\left[m, n, p+\frac{1}{2}\right] \right\} \right. \\ &\left. - \frac{\Delta_t}{\mu \Delta_z} \left\{ E_x^q\left[m+\frac{1}{2}, n, p+1\right] - E_x^q\left[m+\frac{1}{2}, n, p\right] \right\} \right), \end{aligned} \tag{9.16}$$

$$\begin{aligned} H_z^{q+\frac{1}{2}}\left[m+\frac{1}{2}, n+\frac{1}{2}, p\right] &= \frac{1-\frac{\sigma_m \Delta_t}{2\mu}}{1+\frac{\sigma_m \Delta_t}{2\mu}} H_z^{q-\frac{1}{2}}\left[m+\frac{1}{2}, n+\frac{1}{2}, p\right] \\ &+ \frac{1}{1+\frac{\sigma_m \Delta_t}{2\epsilon}} \left(\frac{\Delta_t}{\mu \Delta_y} \left\{ E_x^q\left[m+\frac{1}{2}, n+1, p\right] - E_x^q\left[m+\frac{1}{2}, n, p\right] \right\} \right. \\ &\left. - \frac{\Delta_t}{\epsilon \Delta_x} \left\{ E_y^q\left[m+1, n+\frac{1}{2}, p\right] - E_y^q\left[m, n+\frac{1}{2}, p\right] \right\} \right). \end{aligned} \tag{9.17}$$

$$\begin{aligned} E_x^{q+1}\left[m+\frac{1}{2}, n, p\right] &= \frac{1-\frac{\sigma \Delta_t}{2\epsilon}}{1+\frac{\sigma \Delta_t}{2\epsilon}} E_x^q\left[m+\frac{1}{2}, n, p\right] \\ &+ \frac{1}{1+\frac{\sigma \Delta_t}{2\epsilon}} \left(\frac{\Delta_t}{\epsilon \Delta_y} \left\{ H_z^{q+\frac{1}{2}}\left[m+\frac{1}{2}, n+\frac{1}{2}, p\right] - H_z^{q+\frac{1}{2}}\left[m+\frac{1}{2}, n-\frac{1}{2}, p\right] \right\} \right. \\ &\left. - \frac{\Delta_t}{\epsilon \Delta_z} \left\{ H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}, n, p+\frac{1}{2}\right] - H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2}, n, p-\frac{1}{2}\right] \right\} \right), \end{aligned} \tag{9.18}$$

$$\begin{aligned} E_y^{q+1}\left[m, n+\frac{1}{2}, p\right] &= \frac{1-\frac{\sigma \Delta_t}{2\epsilon}}{1+\frac{\sigma \Delta_t}{2\epsilon}} E_y^q\left[m, n+\frac{1}{2}\right] \\ &+ \frac{1}{1+\frac{\sigma \Delta_t}{2\epsilon}} \left(\frac{\Delta_t}{\epsilon \Delta_z} \left\{ H_x^{q+\frac{1}{2}}\left[m, n+\frac{1}{2}, p+\frac{1}{2}\right] - H_x^{q+\frac{1}{2}}\left[m, n+\frac{1}{2}, p-\frac{1}{2}\right] \right\} \right. \\ &\left. - \frac{\Delta_t}{\epsilon \Delta_x} \left\{ H_z^{q+\frac{1}{2}}\left[m+\frac{1}{2}, n+\frac{1}{2}, p\right] - H_z^{q+\frac{1}{2}}\left[m-\frac{1}{2}, n+\frac{1}{2}, p\right] \right\} \right), \end{aligned} \tag{9.19}$$

$$
\begin{aligned}
E_z^{q+1}\left[m,n,p+\frac{1}{2}\right] &= \frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}} E_z^q\left[m,n,p+\frac{1}{2}\right] \\
&+ \frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\left(\frac{\Delta_t}{\epsilon\Delta_x}\left\{H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n,p+\frac{1}{2}\right] - H_y^{q+\frac{1}{2}}\left[m-\frac{1}{2},n,p+\frac{1}{2}\right]\right\}\right. \\
&\left. - \frac{\Delta_t}{\epsilon\Delta_y}\left\{H_x^{q+\frac{1}{2}}\left[m,n+\frac{1}{2},p+\frac{1}{2}\right] - H_x^{q+\frac{1}{2}}\left[m,n-\frac{1}{2},p+\frac{1}{2}\right]\right\}\right). \qquad (9.20)
\end{aligned}
$$

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

$$
\begin{aligned}
C_{hxh}(m,n+1/2,p+1/2) &= \left.\frac{1-\frac{\sigma_m\Delta_t}{2\mu}}{1+\frac{\sigma_m\Delta_t}{2\mu}}\right|_{m\delta,(n+1/2)\delta,(p+1/2)\delta}, \qquad (9.21) \\
C_{hxe}(m,n+1/2,p+1/2) &= \left.\frac{1}{1+\frac{\sigma_m\Delta_t}{2\mu}}\frac{\Delta_t}{\mu\delta}\right|_{m\delta,(n+1/2)\delta,(p+1/2)\delta}, \qquad (9.22) \\
C_{hyh}(m+1/2,n,p+1/2) &= \left.\frac{1-\frac{\sigma_m\Delta_t}{2\mu}}{1+\frac{\sigma_m\Delta_t}{2\mu}}\right|_{(m+1/2)\delta,n\delta,(p+1/2)\delta}, \qquad (9.23) \\
C_{hye}(m+1/2,n,p+1/2) &= \left.\frac{1}{1+\frac{\sigma_m\Delta_t}{2\mu}}\frac{\Delta_t}{\mu\delta}\right|_{(m+1/2)\delta,n\delta,(p+1/2)\delta}, \qquad (9.24) \\
C_{hzh}(m+1/2,n+1/2,p) &= \left.\frac{1-\frac{\sigma_m\Delta_t}{2\mu}}{1+\frac{\sigma_m\Delta_t}{2\mu}}\right|_{(m+1/2)\delta,(n+1/2)\delta,p\delta}, \qquad (9.25) \\
C_{hze}(m+1/2,n+1/2,p) &= \left.\frac{1}{1+\frac{\sigma_m\Delta_t}{2\mu}}\frac{\Delta_t}{\mu\delta}\right|_{(m+1/2)\delta,(n+1/2)\delta,p\delta}. \qquad (9.26)
\end{aligned}
$$

For the electric-field update equations the coefficients are

$$C_{exe}(m+1/2,n,p) = \left.\frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}}\right|_{(m+1/2)\delta,n\delta,p\delta}, \tag{9.27}$$

$$C_{exh}(m+1/2,n,p) = \left.\frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\frac{\Delta_t}{\epsilon\delta}\right|_{(m+1/2)\delta,n\delta,p\delta}, \tag{9.28}$$

$$C_{eye}(m,n+1/2,p) = \left.\frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}}\right|_{m\delta,(n+1/2)\delta,p\delta}, \tag{9.29}$$

$$C_{eyh}(m,n+1/2,p) = \left.\frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\frac{\Delta_t}{\epsilon\delta}\right|_{m\delta,(n+1/2)\delta,p\delta}, \tag{9.30}$$

$$C_{eze}(m,n,p+1/2) = \left.\frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}}\right|_{m\delta,n\delta,(p+1/2)\delta}, \tag{9.31}$$

$$C_{ezh}(m,n,p+1/2) = \left.\frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\frac{\Delta_t}{\epsilon\delta}\right|_{m\delta,n\delta,(p+1/2)\delta}. \tag{9.32}$$

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) +
```

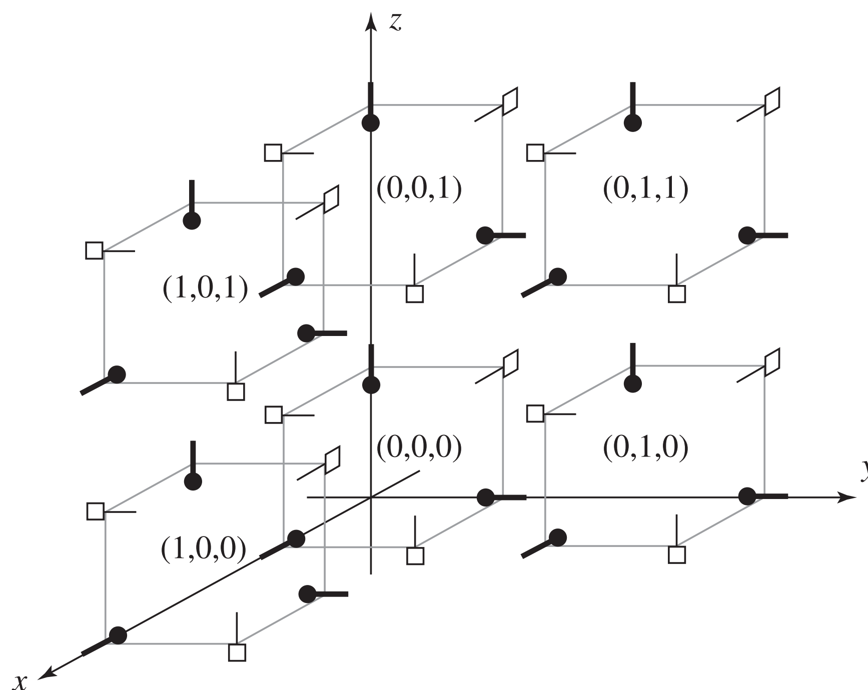


Figure 9.5: Arrangement of six groups of nodes where all of the nodes within the group have the same set of indices. The nodes in a group are joined by gray lines and their indices are shown as an ordered triplet in the center of the group.

```
   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

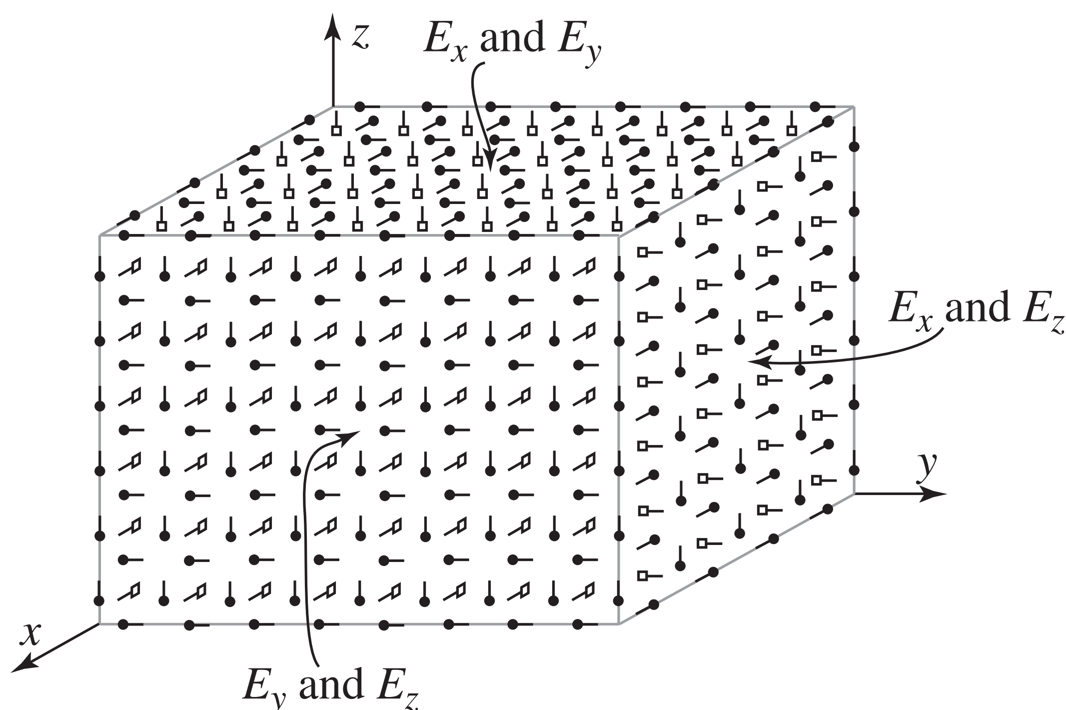


Figure 9.6: Faces of a computational domain which is $5 \times 9 \times 7$ in the x, y, and z directions, respectively. On the constant-x face the tangential fields are E_y and E_z, on the constant-y face they are E_x and E_z, and on the constant-z face they are E_x and E_y. There are also magnetic-field 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.

```
1  /* 3D simulation with dipole source at center of grid. */
2
3  #include "fdtd-alloc.h"
4  #include "fdtd-macro.h"
5  #include "fdtd-proto.h"
6  #include "ezinc.h"
7
8  int main()
9  {
10   Grid *g;
11
12   ALLOC_1D(g, 1, Grid); // allocate memory for grid structure
13   gridInit(g);          // initialize 3D grid
14
15   abcInit(g);           // initialize ABC
16   ezIncInit(g);
17   snapshot3dInit(g);    // initialize snapshots
18
19   /* do time stepping */
20   for (Time = 0; Time < MaxTime; Time++) {
21     updateH(g);         // update magnetic fields
22     updateE(g);         // update electric fields
23     Ex((SizeX - 1) / 2, SizeY / 2, SizeZ / 2) += ezInc(Time, 0.0);
24     abc(g);             // apply ABC
25     snapshot3d(g);      // take a snapshot (if appropriate)
26   } // end of time-stepping
27
```

```
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) +
45 #define ChxeG(G, M, N, P) G->chxe[((M) * (SizeYG(G) - 1) + N) * (SizeZG(G) - 1) +
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)
```

```
117 #define Chyh(M, N, P) ChyhG(g, M, N, P)
118 #define Chye(M, N, P) ChyeG(g, M, N, P)
119
120 #define Hz(M, N, P)   HzG(g, M, N, P)
121 #define Chzh(M, N, P) ChzhG(g, M, N, P)
122 #define Chze(M, N, P) ChzeG(g, M, N, P)
123
124 #define Ex(M, N, P)   ExG(g, M, N, P)
125 #define Cexe(M, N, P) CexeG(g, M, N, P)
126 #define Cexh(M, N, P) CexhG(g, M, N, P)
127
128 #define Ey(M, N, P)   EyG(g, M, N, P)
129 #define Ceye(M, N, P) CeyeG(g, M, N, P)
130 #define Ceyh(M, N, P) CeyhG(g, M, N, P)
131
132 #define Ez(M, N, P)   EzG(g, M, N, P)
133 #define Ceze(M, N, P) CezeG(g, M, N, P)
134 #define Cezh(M, N, P) CezhG(g, M, N, P)
135
136 #define SizeX       SizeXG(g)
137 #define SizeY       SizeYG(g)
138 #define SizeZ       SizeZG(g)
139 #define Time        TimeG(g)
140 #define MaxTime     MaxTimeG(g)
141 #define Cdtds       CdtdsG(g)
142 #define Type        TypeG(g)
143
144 #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.

```
1 #include "fdtd-macro.h"
2 #include <stdio.h>
3
4 /* update magnetic field */
5 void updateH(Grid *g) {
6   int mm, nn, pp;
7
```

```
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)) -
```

```
102                                 (Hy(mm, nn, pp) - Hy(mm, nn, pp - 1)));
103
104     for (mm = 1; mm < SizeX - 1; mm++)
105       for (nn = 0; nn < SizeY - 1; nn++)
106         for (pp = 1; pp < SizeZ - 1; pp++)
107           Ey(mm, nn, pp) = Ceye(mm, nn, pp) * Ey(mm, nn, pp) +
108             Ceyh(mm, nn, pp) * ((Hx(mm, nn, pp) - Hx(mm, nn, pp - 1)) -
109                                 (Hz(mm, nn, pp) - Hz(mm - 1, nn, pp)));
110
111     for (mm = 1; mm < SizeX - 1; mm++)
112       for (nn = 1; nn < SizeY - 1; nn++)
113         for (pp = 0; pp < SizeZ - 1; pp++)
114           Ez(mm, nn, pp) = Ceze(mm, nn, pp) * Ez(mm, nn, pp) +
115             Cezh(mm, nn, pp) * ((Hy(mm, nn, pp) - Hy(mm - 1, nn, pp)) -
116                                 (Hx(mm, nn, pp) - Hx(mm, nn - 1, pp)));
117
118   } else {
119     fprintf(stderr, "updateE: Unknown grid type.  Terminating...\n");
120   }
121
122   return;
123 }  /* 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.

```
1 #include "fdtd-alloc.h"
2 #include "fdtd-macro.h"
3
4 /* Macros to access stored "old" value */
5 #define Eyx0(N, P) eyx0[(N) * (SizeZ) + (P)]
6 #define Ezx0(N, P) ezx0[(N) * (SizeZ - 1) + (P)]
7 #define Eyx1(N, P) eyx1[(N) * (SizeZ) + (P)]
8 #define Ezx1(N, P) ezx1[(N) * (SizeZ - 1) + (P)]
9
```

```
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

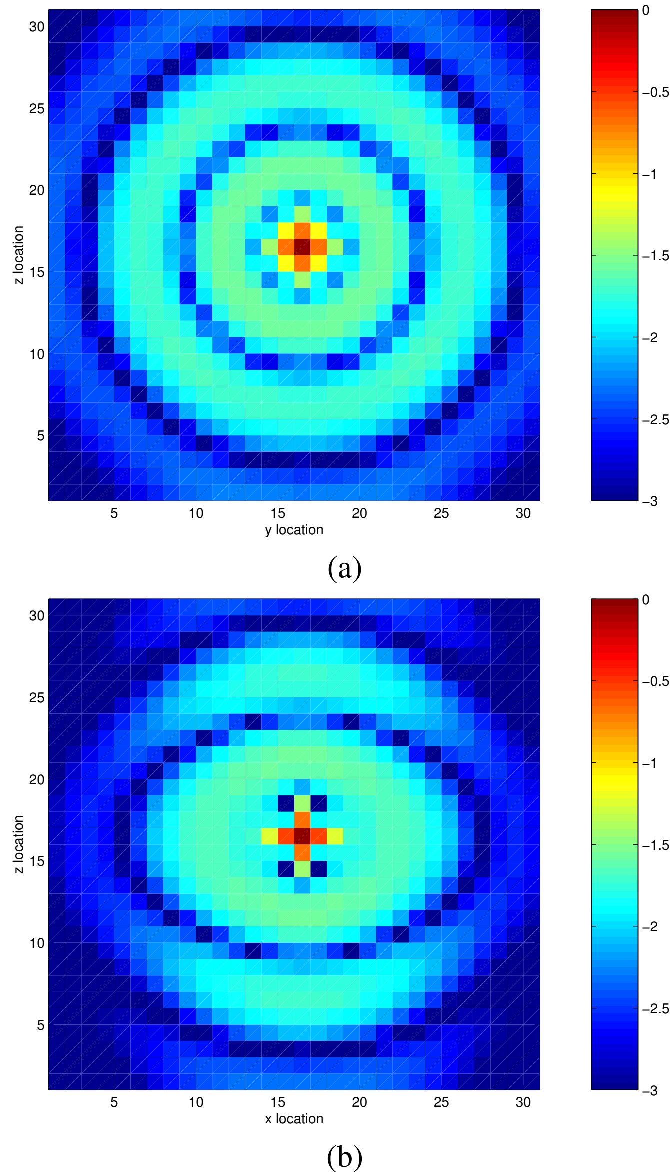


Figure 9.7: Color map of the E_x field at time-step 40. (a) Field over a constant-x plane that passes through the source node. (b) Field over a constant-y plane that passes through the source node. The field has been normalize by 0.3 and three decades of scaling are used. These images were generated 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.

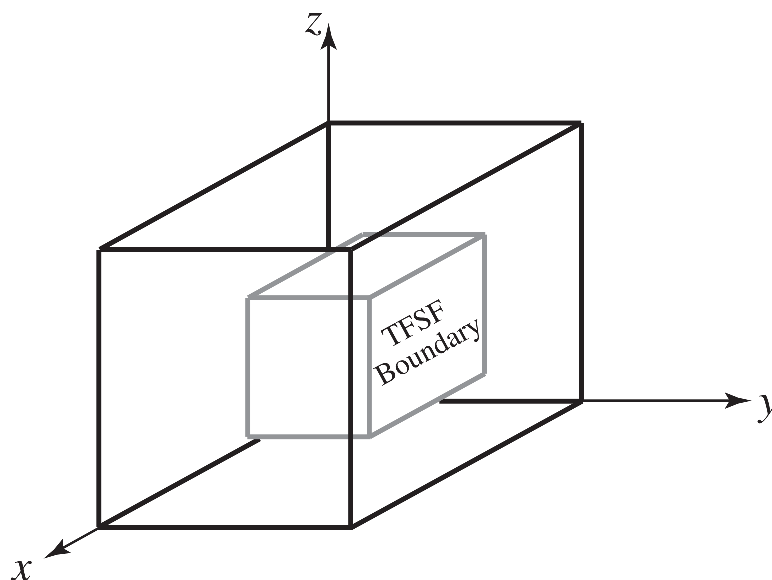


Figure 9.8: Three-dimensional computational domain which contains a TFSF boundary.

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.

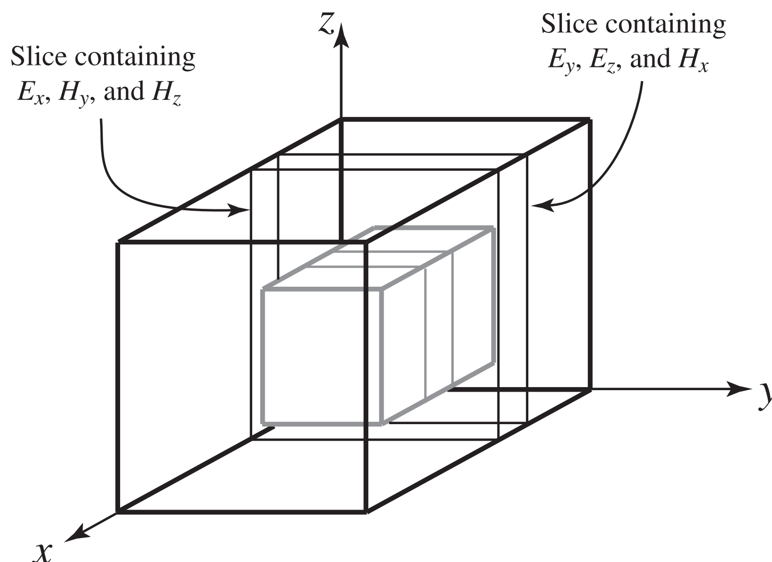


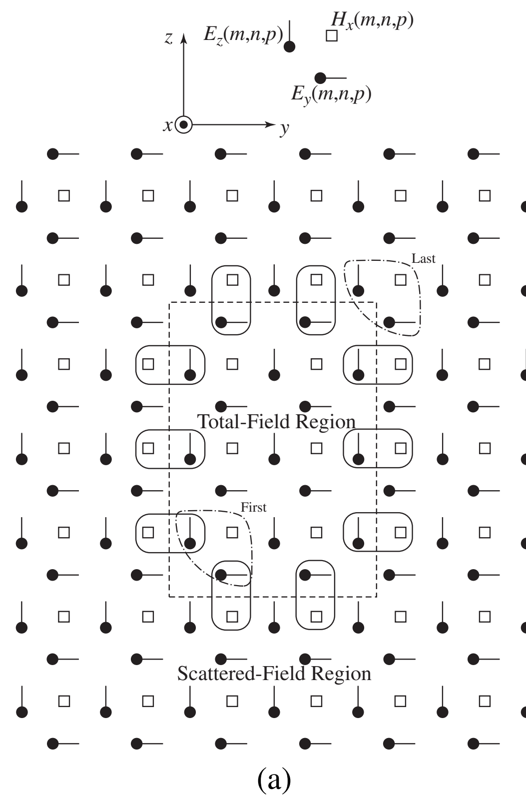
Figure 9.9: Constant-x slices of the computational domain used to illustration the relationship of nodes across the TFSF boundary. The slices are separated by a half spatial-step in the x direction.

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.

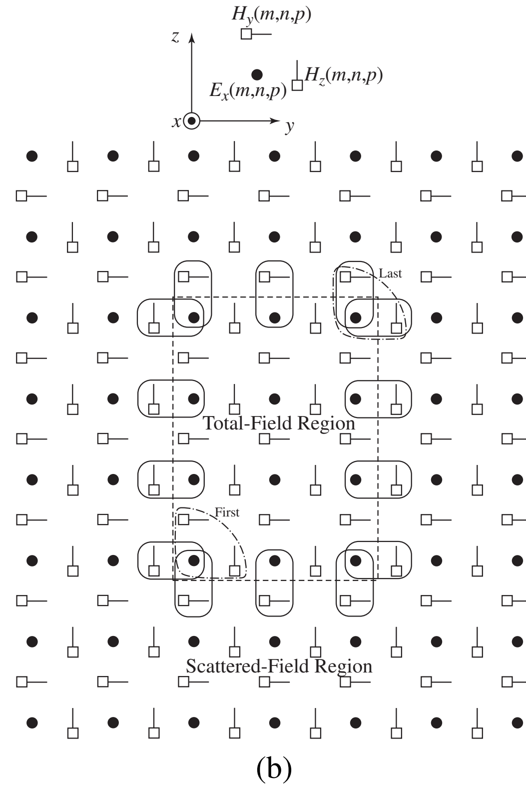
Continuing with the assumption of an incident field whose only non-zero components are E_z and H_y, one sees from Fig. 9.10(a) that H_x nodes on the constant-y faces would have to be corrected since they depend on E_z nodes on the other side of the boundary (these are the nodes along the left and right 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.



(a)



(b)

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.

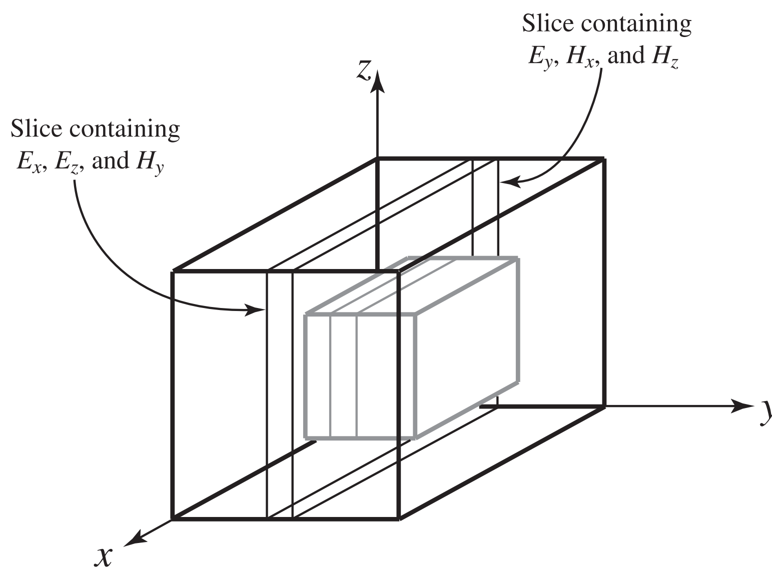


Figure 9.11: Constant-y slices of the computational domain used to illustration the relationship of nodes across the TFSF boundary. The slices are separated by a half spatial-step in the y direction.

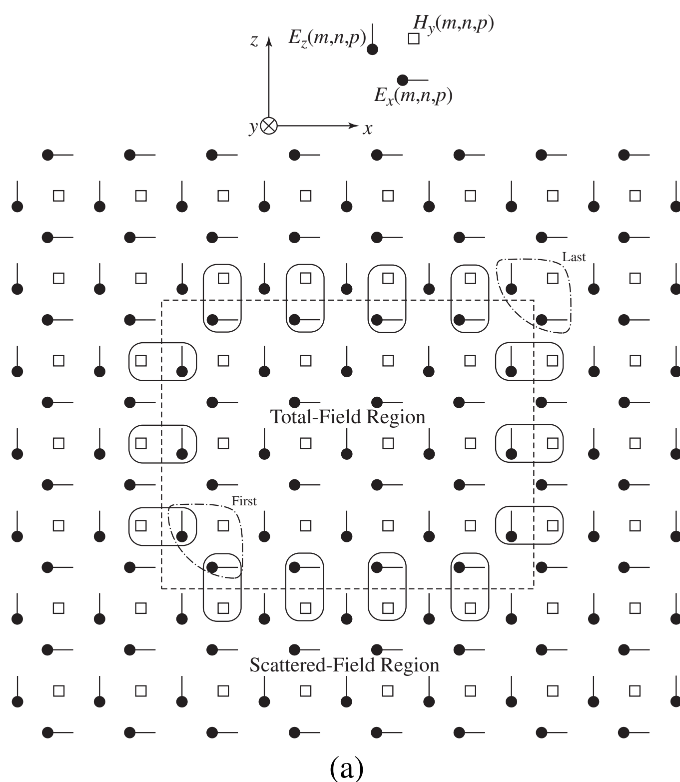
9.6 TFSF Demonstration

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.

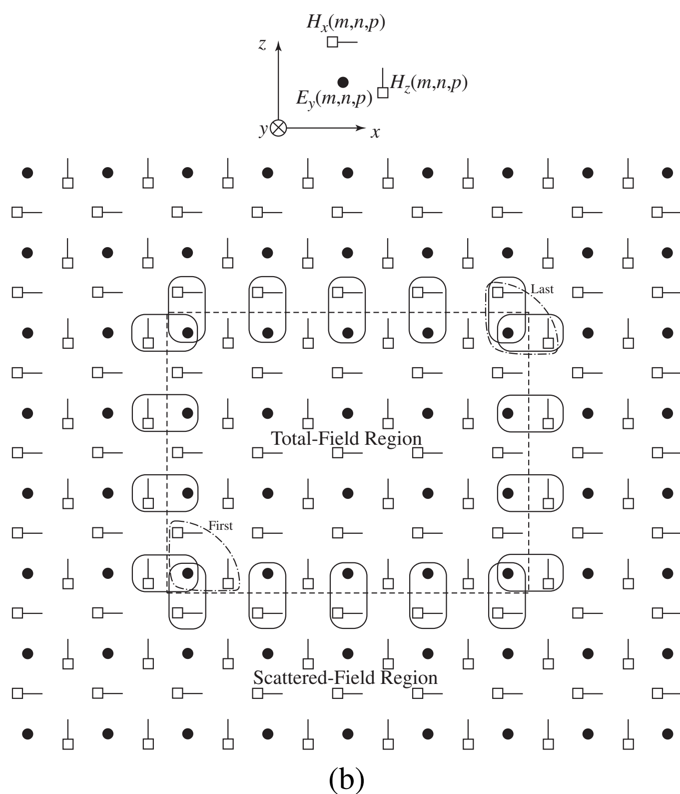
The grid will be $35 \times 35 \times 35$. A Courant number equal to the limit of $1/\sqrt{3}$ will be used. Two simulations will be performed. In one there will be no scatterer and in the other a spherical PEC scatterer will be present. In 3D, the simplest way to model a solid PEC is to test if the 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.

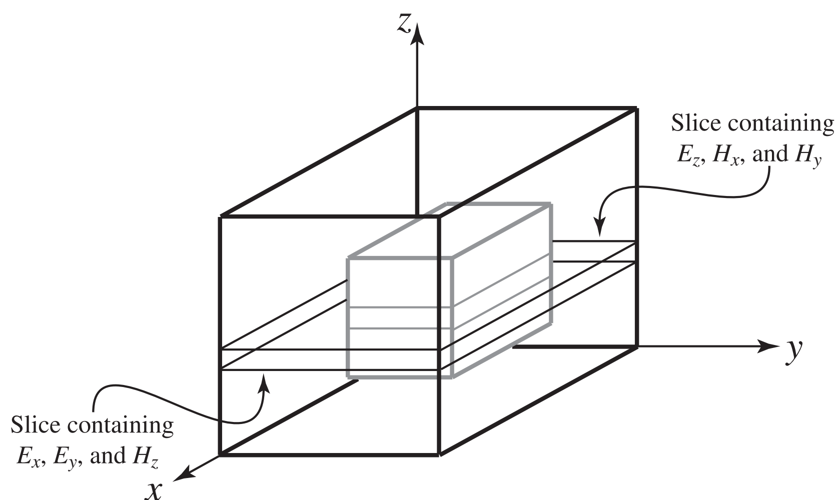
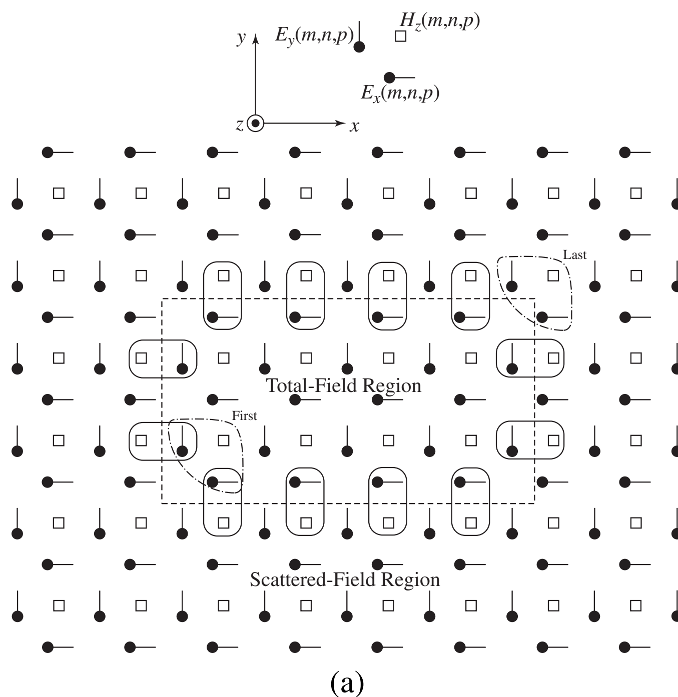
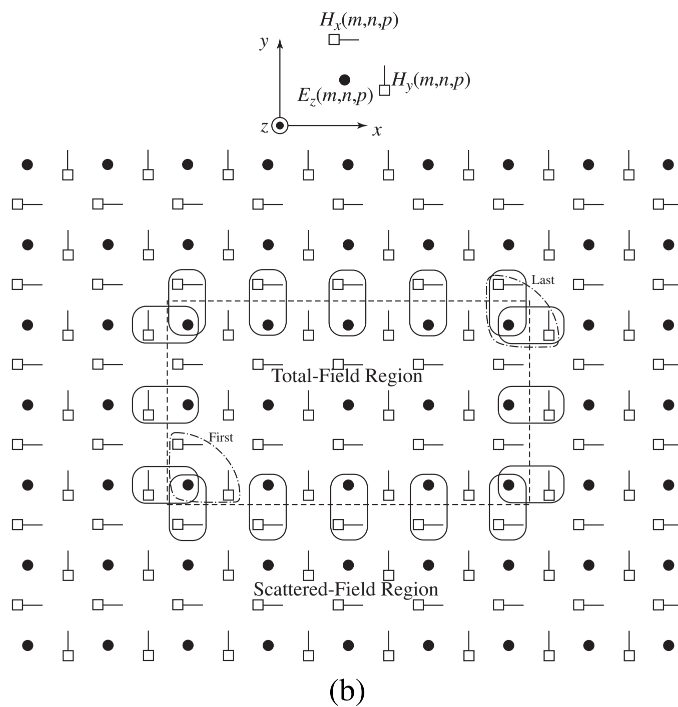


Figure 9.13: Constant-z slices of the computational domain used to illustration the relationship of nodes across the TFSF boundary. The slices are separated by a half spatial-step in the z direction.

```
/* 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

```



(a)



(b)

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

$$\begin{aligned} \delta = \Delta_x \quad &\Rightarrow \quad \Delta_x = \delta, \\ r_y = \frac{\Delta_y}{\Delta_x} \quad &\Rightarrow \quad \Delta_y = r_y \delta, \\ r_z = \frac{\Delta_z}{\Delta_x} \quad &\Rightarrow \quad \Delta_z = r_z \delta. \end{aligned}$$

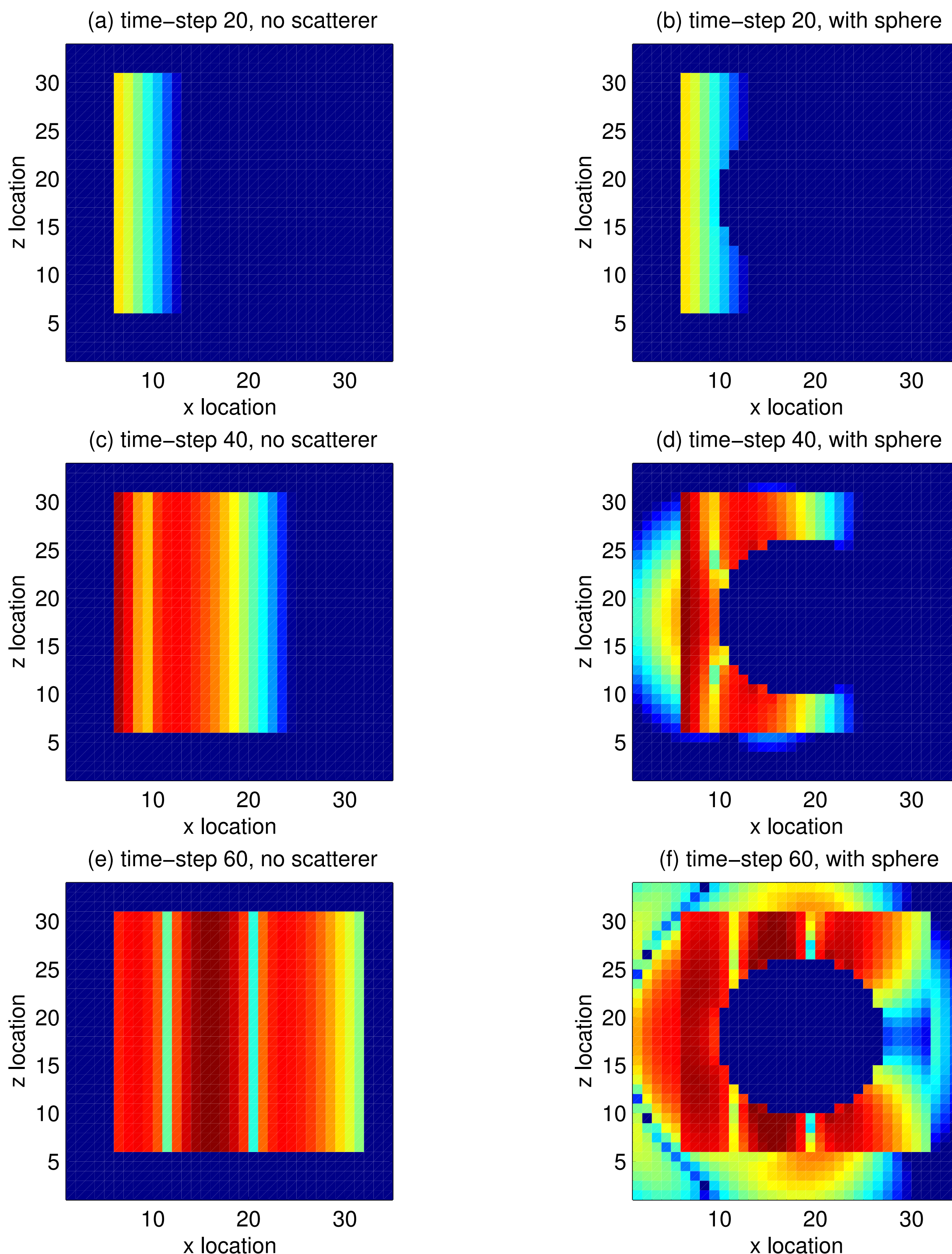


Figure 9.15: The E_z field over a constant-y plane taken from a 3D simulation in which the incident electric field is z-polarized and propagation is in the x direction. In the figures on the left no scatterer is present. Figures (a), (c), and (e) are taken at time-steps, 20, 40, and 60, respectively. 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)

$$\begin{aligned} \frac{1}{\Delta_x}\Delta_x E_x^{q+1}\left[m+\frac{1}{2},n,p\right] &= \frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}}\frac{1}{\Delta_x}\Delta_x E_x^{q}\left[m+\frac{1}{2},n,p\right] \\ &+ \frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\left(\frac{\Delta_t}{\epsilon\Delta_y\Delta_z}\left\{\Delta_z H_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n+\frac{1}{2},p\right]-\Delta_z H_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n-\frac{1}{2},p\right]\right\}\right. \\ &\left. - \frac{\Delta_t}{\epsilon\Delta_y\Delta_z}\left\{\Delta_y H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n,p+\frac{1}{2}\right]-\Delta_y H_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n,p-\frac{1}{2}\right]\right\}\right). \end{aligned} \tag{9.39}$$

Multiplying through by Δ_x and employing the definitions given above, this update equation becomes

$$\begin{aligned} e_x^{q+1}\left[m+\frac{1}{2},n,p\right] &= \frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}} e_x^{q}\left[m+\frac{1}{2},n,p\right] \\ &+ \frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}}\frac{\Delta_t}{\epsilon\delta}\frac{1}{r_y r_z}\left(\left\{h_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n+\frac{1}{2},p\right]-h_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n-\frac{1}{2},p\right]\right\}\right. \\ &\left. - \left\{h_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n,p+\frac{1}{2}\right]-h_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n,p-\frac{1}{2}\right]\right\}\right). \end{aligned} \tag{9.40}$$

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:

$$\begin{aligned} h_x^{q+\frac{1}{2}}\left[m,n+\frac{1}{2},p+\frac{1}{2}\right] &= \frac{1-\frac{\sigma_m\Delta_t}{2\mu}}{1+\frac{\sigma_m\Delta_t}{2\mu}} h_x^{q-\frac{1}{2}}\left[m,n+\frac{1}{2},p+\frac{1}{2}\right] \\ &+ \frac{1}{1+\frac{\sigma_m\Delta_t}{2\mu}}\frac{\Delta_t}{\mu\delta}\frac{1}{r_y r_z}\left(\left\{e_y^{q}\left[m,n+\frac{1}{2},p+1\right]-e_y^{q}\left[m,n+\frac{1}{2},p\right]\right\}\right. \\ &\left. - \left\{e_z^{q}\left[m,n+1,p+\frac{1}{2}\right]-e_z^{q}\left[m,n,p+\frac{1}{2}\right]\right\}\right), \end{aligned} \tag{9.41}$$

$$
\begin{aligned}
h_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n,p+\frac{1}{2}\right] = {} & \frac{1-\frac{\sigma_m\Delta_t}{2\mu}}{1+\frac{\sigma_m\Delta_t}{2\mu}} h_y^{q-\frac{1}{2}}\left[m+\frac{1}{2},n,p+\frac{1}{2}\right] \\
& + \frac{1}{1+\frac{\sigma_m\Delta_t}{2\mu}} \frac{\Delta_t}{\mu\delta} \frac{r_y}{r_z} \left(\left\{ e_z^q\left[m+1,n,p+\frac{1}{2}\right] - e_z^q\left[m,n,p+\frac{1}{2}\right] \right\} \right. \\
& \left. - \left\{ e_x^q\left[m+\frac{1}{2},n,p+1\right] - e_x^q\left[m+\frac{1}{2},n,p\right] \right\} \right), \qquad (9.42)
\end{aligned}
$$

$$
\begin{aligned}
h_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n+\frac{1}{2},p\right] = {} & \frac{1-\frac{\sigma_m\Delta_t}{2\mu}}{1+\frac{\sigma_m\Delta_t}{2\mu}} h_z^{q-\frac{1}{2}}\left[m+\frac{1}{2},n+\frac{1}{2},p\right] \\
& + \frac{1}{1+\frac{\sigma_m\Delta_t}{2\epsilon}} \frac{\Delta_t}{\mu\delta} \frac{r_z}{r_y} \left(\left\{ e_x^q\left[m+\frac{1}{2},n+1,p\right] - e_x^q\left[m+\frac{1}{2},n,p\right] \right\} \right. \\
& \left. - \left\{ e_y^q\left[m+1,n+\frac{1}{2},p\right] - e_y^q\left[m,n+\frac{1}{2},p\right] \right\} \right). \qquad (9.43)
\end{aligned}
$$

$$
\begin{aligned}
e_x^{q+1}\left[m+\frac{1}{2},n,p\right] = {} & \frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}} e_x^q\left[m+\frac{1}{2},n,p\right] \\
& + \frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}} \frac{\Delta_t}{\epsilon\delta} \frac{1}{r_y r_z} \left(\left\{ h_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n+\frac{1}{2},p\right] - h_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n-\frac{1}{2},p\right] \right\} \right. \\
& \left. - \left\{ h_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n,p+\frac{1}{2}\right] - h_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n,p-\frac{1}{2}\right] \right\} \right). \qquad (9.44)
\end{aligned}
$$

$$
\begin{aligned}
e_y^{q+1}\left[m,n+\frac{1}{2},p\right] = {} & \frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}} e_y^q\left[m,n+\frac{1}{2}\right] \\
& + \frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}} \frac{\Delta_t}{\epsilon\delta} \frac{r_y}{r_z} \left(\left\{ h_x^{q+\frac{1}{2}}\left[m,n+\frac{1}{2},p-\frac{1}{2}\right] - h_x^{q+\frac{1}{2}}\left[m,n+\frac{1}{2},p-\frac{1}{2}\right] \right\} \right. \\
& \left. - \left\{ h_z^{q+\frac{1}{2}}\left[m+\frac{1}{2},n+\frac{1}{2},p\right] - h_z^{q+\frac{1}{2}}\left[m-\frac{1}{2},n+\frac{1}{2},p\right] \right\} \right), \qquad (9.45)
\end{aligned}
$$

$$
\begin{aligned}
e_z^{q+1}\left[m,n,p+\frac{1}{2}\right] = {} & \frac{1-\frac{\sigma\Delta_t}{2\epsilon}}{1+\frac{\sigma\Delta_t}{2\epsilon}} e_z^q\left[m,n,p+\frac{1}{2}\right] \\
& + \frac{1}{1+\frac{\sigma\Delta_t}{2\epsilon}} \frac{\Delta_t}{\epsilon\delta} \frac{r_z}{r_y} \left(\left\{ h_y^{q+\frac{1}{2}}\left[m+\frac{1}{2},n,p+\frac{1}{2}\right] - h_y^{q+\frac{1}{2}}\left[m-\frac{1}{2},n,p+\frac{1}{2}\right] \right\} \right. \\
& \left. - \left\{ h_x^{q+\frac{1}{2}}\left[m,n+\frac{1}{2},p+\frac{1}{2}\right] - h_x^{q+\frac{1}{2}}\left[m,n-\frac{1}{2},p+\frac{1}{2}\right] \right\} \right). \qquad (9.46)
\end{aligned}
$$

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_{hzh}, 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_{hze} 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

$$C_{hxe}(m, n+1/2, p+1/2) = \left. \frac{1}{1+\frac{\sigma_m \Delta_t}{2\mu}} \frac{\Delta_t}{\mu\delta} \frac{1}{r_y r_z} \right|_{m\Delta_x,(n+1/2)\Delta_y,(p+1/2)\Delta_z}, \tag{9.47}$$

$$C_{hye}(m+1/2, n, p+1/2) = \left. \frac{1}{1+\frac{\sigma_m \Delta_t}{2\mu}} \frac{\Delta_t}{\mu\delta} \frac{r_y}{r_z} \right|_{(m+1/2)\Delta_x,n\Delta_y,(p+1/2)\Delta_z}, \tag{9.48}$$

$$C_{hze}(m+1/2, n+1/2, p) = \left. \frac{1}{1+\frac{\sigma_m \Delta_t}{2\mu}} \frac{\Delta_t}{\mu\delta} \frac{r_z}{r_y} \right|_{(m+1/2)\Delta_x,(n+1/2)\Delta_y,p\Delta_z}, \tag{9.49}$$

$$C_{exh}(m+1/2, n, p) = \left. \frac{1}{1+\frac{\sigma \Delta_t}{2\epsilon}} \frac{\Delta_t}{\epsilon\delta} \frac{1}{r_y r_z} \right|_{(m+1/2)\Delta_x,n\Delta_y,p\Delta_z}, \tag{9.50}$$

$$C_{eyh}(m, n+1/2, p) = \left. \frac{1}{1+\frac{\sigma \Delta_t}{2\epsilon}} \frac{\Delta_t}{\epsilon\delta} \frac{r_y}{r_z} \right|_{m\Delta_x,(n+1/2)\Delta_y,p\Delta_z}, \tag{9.51}$$

$$C_{ezh}(m, n, p+1/2) = \left. \frac{1}{1+\frac{\sigma \Delta_t}{2\epsilon}} \frac{\Delta_t}{\epsilon\delta} \frac{r_z}{r_y} \right|_{m\Delta_x,n\Delta_y,(p+1/2)\Delta_z}. \tag{9.52}$$

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.