CHAPTER 3

Building objects in the Yee grid

In Chapter 1 we discussed the derivation of the finite-difference time-domain (FDTD) updating equations based on the staircased grid composed of Yee cells in Cartesian coordinates. We defined material components ε, μ, σ^e, and σ^m associated with the field components such that they represent linear, anisotropic, and nondispersive media. Due to the staircase gridding, the objects in an FDTD problem space can be represented in terms of the size of the cells building the grid. Therefore, the staircased gridding imposes some restrictions on the accurate representation of the objects in the problem space. Some advanced modeling techniques called *local subcell models* are available for defining objects with fine features, and some other FDTD formulations based on nonorthogonal and unstructured grids have been introduced for problems that contain objects that do not conform with the staircased grid [1]. However, the staircased FDTD model can provide sufficiently accurate results for many practical problems. In this chapter, we discuss construction of objects in the staircased FDTD grid through some MATLAB® code examples.

3.1 Definition of objects

Throughout this book, while describing the concepts of the FDTD method we illustrate the construction of an FDTD program as well. After completing this book, the reader should be able to construct an FDTD program that can be used to solve three-dimensional electromagnetics problems of moderate difficulty. In this chapter, we start to provide the blocks of a three-dimensional FDTD program.

We name the main MATLAB program file that runs an FDTD calculation as ***fdtd_solve***. The MATLAB code in Listing 3.1 shows the contents of ***fdtd_solve***, in which the program routines are named by their functions. Usually, it is a good practice to divide a program into functionally separable modules; this will facilitate the readability of the programs and will simplify the debugging process. This file is not in the final form for a complete FDTD program; while we proceed with the chapters we will add more routines with additional functions, will describe the functions of the routines, and will provide partial code sections to illustrate the programming of the respective concepts that have been described in the text. The program structures and codes presented throughout this book are not necessarily samples of the most efficient ways of programming the FDTD method; there

Listing 3.1 fdtd_solve.m

```
% initialize the matlab workspace
clear all; close all; clc;

% define the problem
define_problem_space_parameters;
define_geometry;
define_sources_and_lumped_elements;
define_output_parameters;

% initialize the problem space and parameters
initialize_fdtd_material_grid;
display_problem_space;
display_material_mesh;
if run_simulation
    initialize_fdtd_parameters_and_arrays;
    initialize_sources_and_lumped_elements;
    initialize_updating_coefficients;
    initialize_boundary_conditions;
    initialize_output_parameters;
    initialize_display_parameters;

    % FDTD time marching loop
    run_fdtd_time_marching_loop;

    % display simulation results
    post_process_and_display_results;
end
```

are many ways of translating a concept into a program, and readers can develop program structures or algorithms that they think are more efficient while writing their own code. However, we tried to be as explicit as possible while linking the FDTD concepts with the respective codes.

In the FDTD algorithm, before the time-marching iterations are performed the problem should be set up as the first step as indicated in the concise flow chart in Figure 1.9. Problem setup can be performed in two sets of routines: (1) routines that *define* the problem; and (2) routines that *initialize* the problem space and FDTD parameters. The problem definition process consists of construction of data structures that store the necessary information about the electromagnetic problem to be solved. This information would include the geometries of the objects in the problem space, their material types, electromagnetic properties of these material types, types of sources and waveforms, types of simulation results sought from the FDTD computation, and some other simulation parameters specific to the FDTD algorithm, such as the number of time steps, and types of boundaries of the problem space. The initialization process translates the data structures into an FDTD material grid and constructs and initializes data structures and arrays representing the FDTD updating coefficients, field components, and any other data structures that would be used during and after the FDTD iterations. In other words, it prepares the structural framework for the FDTD computation and postprocessing.

3.1.1 Defining the problem space parameters

Listing 3.1 starts with the initialization of MATLAB workspace. The MATLAB command **clear all** removes all items from the current workspace and frees the memory, **close all** deletes all open figures, and **clc** clears the command window. After the MATLAB workspace initialization, there are four subroutines listed in Listing 3.1 for definition of the problem and other subroutines for initialization of the FDTD procedure. In this chapter, we implement the subroutines ***define_problem_space_parameters***, ***define_geometry***, and ***initialize_fdtd_material_grid***.

The contents of the subroutine ***define_problem_space_parameters*** are given in Listing 3.2. Listing 3.2 starts with the MATLAB command ***disp***, which displays the string in its argument on MATLAB command window. Here the string argument of ***disp*** indicates that the program is

Listing 3.2 define_problem_space_parameters.m

```
disp('defining the problem space parameters');

% maximum number of time steps to run FDTD simulation
number_of_time_steps = 700;

% A factor that determines duration of a time step
% wrt CFL limit
courant_factor = 0.9;

% A factor determining the accuracy limit of FDTD results
number_of_cells_per_wavelength = 20;

% Dimensions of a unit cell in x, y, and z directions (meters)
dx=2.4e-3;
dy=2.0e-3;
dz=2.2e-3;

% ==<boundary conditions>========
% Here we define the boundary conditions parameters
% 'pec' : perfect electric conductor
boundary.type_xp = 'pec';
boundary.air_buffer_number_of_cells_xp = 5;

boundary.type_xn = 'pec';
boundary.air_buffer_number_of_cells_xn = 5;

boundary.type_yp = 'pec';
boundary.air_buffer_number_of_cells_yp = 10;

boundary.type_yn = 'pec';
boundary.air_buffer_number_of_cells_yn = 5;

boundary.type_zp = 'pec';
boundary.air_buffer_number_of_cells_zp = 5;

boundary.type_zn = 'pec';
boundary.air_buffer_number_of_cells_zn = 0;
```

```
% ===<material types>============
% Here we define and initialize the arrays of material types
% eps_r    : relative permittivity
% mu_r     : relative permeability
% sigma_e  : electric conductivity
% sigma_m  : magnetic conductivity

% air
material_types(1).eps_r   = 1;
material_types(1).mu_r    = 1;
material_types(1).sigma_e = 0;
material_types(1).sigma_m = 0;
material_types(1).color   = [1 1 1];

% PEC : perfect electric conductor
material_types(2).eps_r   = 1;
material_types(2).mu_r    = 1;
material_types(2).sigma_e = 1e10;
material_types(2).sigma_m = 0;
material_types(2).color   = [1 0 0];

% PMC : perfect magnetic conductor
material_types(3).eps_r   = 1;
material_types(3).mu_r    = 1;
material_types(3).sigma_e = 0;
material_types(3).sigma_m = 1e10;
material_types(3).color   = [0 1 0];

% a dielectric
material_types(4).eps_r   = 2.2;
material_types(4).mu_r    = 1;
material_types(4).sigma_e = 0;
material_types(4).sigma_m = 0.2;
material_types(4).color   = [0 0 1];

% a dielectric
material_types(5).eps_r   = 3.2;
material_types(5).mu_r    = 1.4;
material_types(5).sigma_e = 0.5;
material_types(5).sigma_m = 0.3;
material_types(5).color   = [1 1 0];

% indices of material types defining air, PEC, and PMC
material_type_index_air = 1;
material_type_index_pec = 2;
material_type_index_pmc = 3;
```

executing the routine in which the problem space parameters are defined. Displaying this kind of informative statement at various stages of the program indicates the progress of the execution as well as helps for debugging any sources of errors if they exist.

The total number of time steps that the FDTD time-marching iterations will run for is defined in line 4 of Listing 3.2 with the parameter **number_of_time_steps**. Line 8 defines a parameter named **courant_factor**: a factor by which the duration of a time step is determined with respect to the CFL stability limit as described in Chapter 2. In line 11 another parameter is defined with the name **number_of_cells_per_wavelength**. This is a parameter by which the highest frequency in the Fourier spectrum of a source waveform is determined for a certain accuracy level. This parameter is described in more detail in Chapter 5, where the source waveforms are discussed. On lines 14, 15, and 16 the dimensions of a unit cell constructing the uniform FDTD problem grid are defined as parameters **dx**, **dy**, and **dz**, respectively.

One of the important concepts in the FDTD technique is the treatment of boundaries of the problem space. The treatment of perfect electric conductor (PEC) boundaries was demonstrated in Chapter 1 through a one-dimensional FDTD code. However, some types of problems may require other types of boundaries. For instance, an antenna problem requires that the boundaries simulate radiated fields propagating outside the finite problem space to infinity. Unlike a PEC boundary, this type of boundary requires advanced algorithms, which is the subject of a separate chapter. However, for now, we limit our discussion to PEC boundaries. A conventional FDTD problem space composed of uniform Yee cells is a rectangular box having six faces. On lines 21–37 of Listing 3.2 the boundary types of these six faces are defined using a data structure called **boundary**. The structure **boundary** has two types of fields: **type** and **air_buffer_number_of_cells**. These field names have extensions indicating the specific face of the domain under consideration. For instance, the extension **_xn** refers to the face of the problem space for which the normal vector is directed in the negative x direction, where **n** stands for *negative direction*. Similarly, the extension **_zp** refers to the face of the problem space for which the normal vector is directed in the positive z direction, where **p** stands for *positive direction*. The other four extensions refer to four other faces. The field **type** defines the type of the boundary under consideration, and in this case all boundaries are defined as PEC by the keyword ***pec***. Other types of boundaries are identified with other keywords.

Some types of boundary conditions require that the boundaries be placed at a distance from the objects. Most of the time this space is filled with air, and the distance in between is given in terms of the number of cells. Hence, the parameter **air_buffer_number_of_cells** defines the distance of the respective boundary faces of the rectangular problem space from the objects placed in the problem space. If the value of this parameter is zero, it implies that the boundary touches the objects.

Finally, we define different types of materials that the objects in the problem space are made of. Since more than one type of material may be used to construct the objects, an array structure with name **material_types** is used as shown in lines 47–79 of Listing 3.2. An index i in **material_types(i)** refers to the ith material type. Later on, every object will be assigned a material type through the index of the material type. An isotropic and homogeneous material type can be defined by its electromagnetic parameters: relative permittivity ε_r, relative permeability μ_r, electric conductivity σ^e, and magnetic conductivity σ^m; hence, fields **eps_r**, **mu_r**, **sigma_e**, and **sigma_m** are used with parameter **material_types(i)**, respectively, to define the electromagnetic parameters of ith material type. In the given example code, a very high value is assigned to **material_types(2).sigma_e** to construct a material type simulating a PEC, and a very high value is assigned to **material_types(3).sigma_m** to construct a material type simulating a perfect magnetic conductor (PMC). A fifth field **color** in **material_types(i)** is optional and is used to assign a color to each material type, which will help in distinguishing different material types when displaying the objects in the problem space on a MATLAB figure. The field **color** is a vector of three values corresponding to red, green, and blue intensities of the red, green, and blue (RGB) color scheme, respectively, with each color scaled between 0 and 1.

While defining the **material_types** it is a good practice to reserve some indices for some common material types. Some of these material types are air, PEC, and PMC, and these material types are indexed as 1, 2, and 3 in the **material_types** structure array, respectively. Then we can define additional parameters **material_type_index_air**, **material_type_index_pec**, and **material_type_index_pmc**, which store the indices of these materials and can be used to identify them in the program.

3.1.2 Defining the objects in the problem space

In the previous section, we discussed the definition of some problem space parameters including the boundary types and material types. In this section we discuss the implementation of the routine ***define_geometry*** contents, which are given in Listing 3.3. Different types of three-dimensional objects can be placed in a problem geometry. We show example implementations using *prisms* and *spheres* due to their geometrical simplicity, and it is possible to construct more complicated shapes by using a combination of these simple objects. Here we will use the term *brick* to indicate a *prism*. A brick, which has its faces parallel to the Cartesian coordinate axes, can be represented by two corner points: (1) the point with lower x, y, and z coordinates; and (2) the point with upper x, y, and z coordinates as illustrated in Figure 3.1(a). Therefore, a structure array called **bricks** is used in Listing 3.3 together with the fields **min_x**, **min_y**, **min_z**, **max_x**, **max_y**, and **max_z** corresponding to their respective positions in Cartesian coordinates. Each element of the array **bricks** is a brick object indexed by i and is referred to by **bricks(i)**. Another field of the parameter **bricks(i)** is the parameter **material_type**, which refers to the index of the material type of the ith brick. In Listing 3.3 **bricks(1).material_type** is 4, indicating that brick 1 is made of **material_types(4)**, which is defined in Listing 3.2 as a dielectric with $\varepsilon_r = 2.2$ and $\sigma^m = 0.2$. Similarly, **bricks(2).material_type** is 2, indicating that brick 2 is made of **material_types(2)**, which is defined as a PEC.

A sphere can be defined by its center coordinates and radius as illustrated in Figure 3.1(b). Therefore, a structure array **spheres** is used in Listing 3.3 together with the fields **center_x**, **center_y**, **center_z**, and **radius** which correspond to the respective parameters in Cartesian coordinates. Similar to the brick case, the field **material_type** is used together with **spheres(i)** to define the material type of the ith sphere. For example, in Listing 3.3 two spheres are defined with coinciding centers. Sphere 1 is a dielectric of **material_types(5)** with a radius of 20 mm, and sphere 2 is a dielectric of **material_types(1)** with a radius of 15 mm. If these two spheres are created in the problem space in sequence (sphere 1 first and sphere 2 second), their combination is going to form a hollow shell of thickness 5 mm since the material type of the inner sphere **material_types(1)** is air. One should notice that Listing 3.3 starts with two lines defining two arrays, **bricks** and **spheres**, and initializes them by null. Even though we are not going to define an object corresponding to some of these arrays, we still define them and initialize them as empty arrays. Later, when the program executes, these arrays will be accessed by some routines of the program.

Listing 3.3 define_geometry.m

```
disp('defining␣the␣problem␣geometry');

bricks  = [];
spheres = [];

% define a brick with material type 4
bricks(1).min_x = 0;
bricks(1).min_y = 0;
bricks(1).min_z = 0;
bricks(1).max_x = 24e-3;
bricks(1).max_y = 20e-3;
bricks(1).max_z = 11e-3;
bricks(1).material_type = 4;

% define a brick with material type 2
bricks(2).min_x = -20e-3;
bricks(2).min_y = -20e-3;
bricks(2).min_z = -11e-3;
bricks(2).max_x = 0;
bricks(2).max_y = 0;
bricks(2).max_z = 0;
bricks(2).material_type = 2;

% define a sphere with material type 5
spheres(1).radius   = 20e-3;
spheres(1).center_x = 0;
spheres(1).center_y = 0;
spheres(1).center_z = 40e-3;
spheres(1).material_type = 5;

% define a sphere with material type 1
spheres(2).radius   = 15e-3;
spheres(2).center_x = 0;
spheres(2).center_y = 0;
spheres(2).center_z = 40e-3;
spheres(2).material_type = 1;
```

If MATLAB tries to access a parameter that does not exist in its workspace, it is going to fail and stop the execution of the program. To prevent such a runtime error we define these arrays even though they are empty.

So far we have defined an FDTD problem space and defined some objects existing in it. Combining the definitions in Listings 3.2 and 3.3 we obtained a problem space, which is plotted in Figure 3.2. The dimensions of the cells in the grids in Figure 3.2 are the same as the dimensions of the unit cell making the FDTD grid. Therefore, it is evident that the boundaries are away from the objects by the distances defined in Listing 3.2. Furthermore, the objects are placed in the three-dimensional space with the positions and dimensions as defined in Listing 3.3, and their colors are set as their corresponding material type colors.

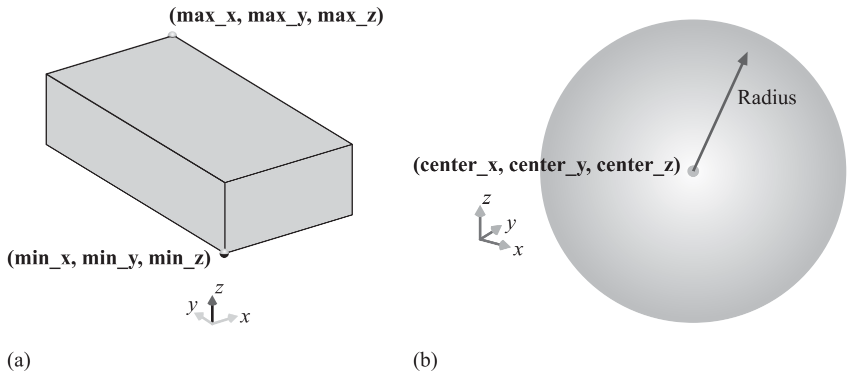


Figure 3.1 Parameters defining a brick and a sphere in Cartesian coordinates: (a) parameters defining a brick and (b) parameters defining a sphere.

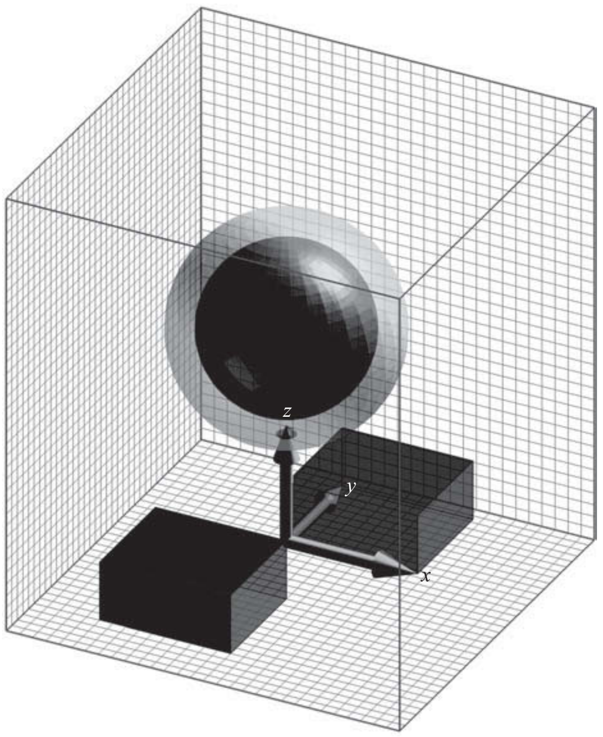


Figure 3.2 An FDTD problem space and the objects defined in it.

3.2 Material approximations

In a three-dimensional FDTD problem space, the field components are defined at discrete locations, and the associated material components are defined at the same locations as illustrated in Figures 1.5 and 1.6. The material components need to be assigned appropriate

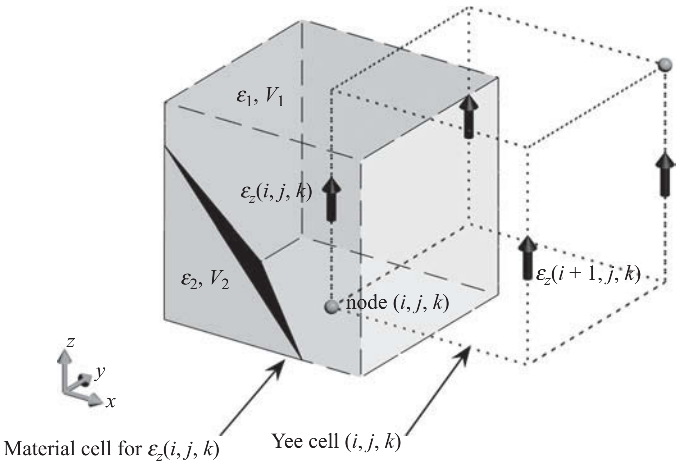


Figure 3.3 A cell around material component $\varepsilon_z(i, j, k)$ partially filled with two media.

values representing the media existing at the respective positions. However, the medium around a material component may not be homogeneous, and some approximation strategies need to be adopted.

One of the strategies is to assume that each material component is residing at the center of a cell. These cells are offset from the Yee cells, and we refer to them as *material cells*. For instance, consider the z-component of permittivity shown in Figure 3.3, which can be imagined as being at the center of a material cell partially filled with two different media denoted by subscripts 1 and 2, with permittivity values ε_1 and ε_2. The simplest approach is to assume that the cell is completely filled with medium 1 since the material component $\varepsilon_z(i, j, k)$ resides in the medium 1. Therefore, $\varepsilon_z(i, j, k)$ can be assigned ε_1; $\varepsilon_z(i, j, k) = \varepsilon_1$.

A better approach would include the effect of ε_2 by employing an averaging scheme. If it is possible to obtain the volume of each medium filling the material cell, a simple weighted volume averaging can be used as employed in [5]. In the example case shown in Figure 3.3, $\varepsilon_z(i, j, k)$ can be calculated as $\varepsilon_z(i, j, k) = (V_1 \times \varepsilon_1 + V_2 \times \varepsilon_2)/(V_1 + V_2)$, where V_1 and V_2 are the volumes of medium 1 and medium 2 in the material cell, respectively.

In many cases, calculation of the volume of each media in a cell may require tedious calculations, whereas it may be much simpler to determine the medium in which a point resides. A crude approximation to the weighted volume averaging scheme is proposed in [6], which may be useful for such cases. In this approach, every cell in the Yee grid is divided into eight subcells, and each material component is located in between eight subcells, as illustrated in Figure 3.4. For every subcell center point the respective medium can be determined and every subcell can be assigned the respective medium material property. Then an effective average of the eight subcells surrounding a material component can be calculated and assigned to the respective material component. For instance, $\varepsilon_z(i, j, k)$ in Figure 3.4, for uniform cell sizes in the x, y, and z directions, can be calculated as

$$\varepsilon_z(i, j, k) = \frac{\varepsilon_1 + \varepsilon_2 + \varepsilon_3 + \varepsilon_4 + \varepsilon_5 + \varepsilon_6 + \varepsilon_7 + \varepsilon_8}{8}. \tag{3.1}$$

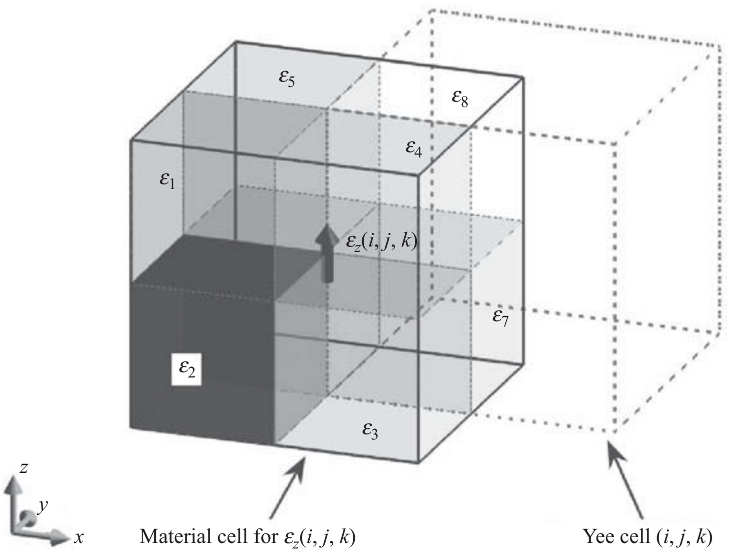


Figure 3.4 A cell around material component $\varepsilon_z(i, j, k)$ divided into eight subcells.

The advantage of this method is that objects may be modeled with eight times the geometrical resolution (two times in each direction) without increasing the size of the staggered grid and memory requirements.

3.3 Subcell averaging schemes for tangential and normal components

The methods discussed thus far do not account for the orientation of the field component associated with the material component under consideration with respect to the boundary interface between two different media. For instance, Figures 3.5 and 3.6 show two such cases where a material cell is partially filled with two different types of media. In the first case the material component is parallel to the boundary of the two media, whereas in the second case the material component is normal to the boundary. Two different averaging schemes can be developed to obtain an equivalent medium type to be assigned to the material component.

For instance, Ampere's law can be expressed in integral form based on the configuration in Figure 3.5 as

$$\oint \vec{H} \cdot d\bar{l} = \frac{d}{dt}\int \vec{D} \cdot d\bar{s} = \frac{d}{dt} D_z \times (A_1 + A_2),$$

where D_z is the electric displacement vector component in the z direction, which is assumed to be uniform in cross-section, and A_1 and A_2 are the cross-section areas of the two media types in the plane normal to D_z. The electric field components in the two media are E_{z1} and E_{z2}. The total electric flux through the cell cross-section can be expressed as

$$D_z \times (A_1 + A_2) = \varepsilon_1 E_{z1} A_1 + \varepsilon_2 E_{z2} A_2 = \varepsilon_{eff} E_z \times (A_1 + A_2),$$

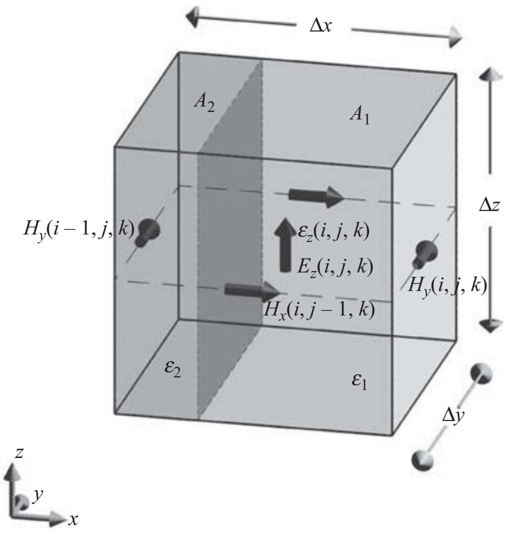


Figure 3.5 Material component $\varepsilon_z(i, j, k)$ parallel to the boundary of two different media partially filling a material cell.

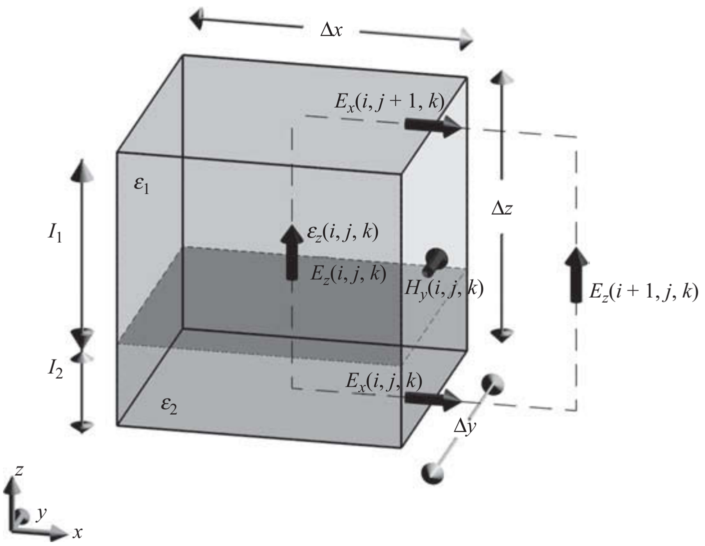


Figure 3.6 Material component $\varepsilon_z(i, j, k)$ normal to the boundary of two different media partially filling a material cell.

where E_z is the electric field component (i, j, k), which is assumed to be uniform in the cross-section of the cell and associated with the permittivity ε_{eff} that is equivalent to $\varepsilon_z(i, j, k)$. On the boundary of the media the tangential components of the electric field are continuous such that $E_{z1} = E_{z2} = E_z$; therefore, one can write

$$\varepsilon_1 A_1 + \varepsilon_2 A_2 = \varepsilon_{eff} \times (A_1 + A_2) \Rightarrow \varepsilon_{eff} = \varepsilon_z(i, j, k) = \frac{\varepsilon_1 A_1 + \varepsilon_2 A_2}{(A_1 + A_2)}. \tag{3.2}$$

Hence, an equivalent permittivity can be calculated for a material component parallel to the interface of two different material types of permittivity ε_1 and ε_2 using (3.2), by which the electric flux conservation is maintained.

For the case where the material component is normal to the media boundary we can employ Faradays's law in integral form based on the configuration in Figure 3.6, which reads

$$\oint \vec{E}\cdot d\bar{l} = -\frac{d}{dt}\int \vec{B}\cdot d\bar{s}$$
$$= -\Delta x E_x(i,j,k) + \Delta x E_x(i,j+1,k) + \Delta z E_z(i,j,k) - \Delta z E_z(i+1,j,k),$$

where E_z is the equivalent electric field vector component in the z direction, which is assumed to be uniform in the boundary cross-section. However, in reality there are two different electric field values, E_{z1} and E_{z2}, due to the different material types. We want to obtain E_z to represent the equivalent effect of E_{z1} and E_{z2} and ε_{eff} to represent the equivalent effect of ε_1 and ε_2. The electric field terms shall satisfy

$$\Delta z E_z(i,j,k) = (l_1+l_2)\times E_z(i,j,k) = l_1E_{z1} + l_2E_{z2}. \tag{3.3}$$

On the boundary of the media the normal components of the electric flux $\vec{D}$ are continuous such that $D_{z1} = D_{z2} = D_z(i,j,k)$; therefore, one can write

$$D_z(i,j,k) = \varepsilon_1E_{z1} = \varepsilon_2E_{z2} = \varepsilon_{eff}E_z(i,j,k), \tag{3.4}$$

which can be expressed in the form

$$E_{z1} = \frac{D_z(i,j,k)}{\varepsilon_1}, \quad E_{z2} = \frac{D_z(i,j,k)}{\varepsilon_2}, \quad E_z(i,j,k) = \frac{D_z(i,j,k)}{\varepsilon_{eff}}. \tag{3.5}$$

Using (3.5) in (3.3) and eliminating the terms $D_z(i,j,k)$ yields

$$\frac{l_1+l_2}{\varepsilon_{eff}} = \frac{l_1}{\varepsilon_1} + \frac{l_2}{\varepsilon_2} \Rightarrow \varepsilon_{eff} = \varepsilon_z(i,j,k) = \frac{\varepsilon_1\varepsilon_2\times(l_1+l_2)}{(l_1\varepsilon_2+l_2\varepsilon_1)}. \tag{3.6}$$

Hence, an expression for the equivalent permittivity has been obtained in (3.6), by which the magnetic flux conservation is maintained.

The averaging schemes given by (3.2) and (3.6) are specific cases of the more general approaches introduced in [7], [8], and [9].

It can be shown that these schemes can be employed for other material parameters as well. For instance, the equivalent permeability μ can be written as

$$\mu_{eff} = \frac{\mu_1A_1+\mu_2A_2}{(A_1+A_2)}, \quad \mu \text{ parallel to media boundary}, \tag{3.7a}$$

$$\mu_{eff} = \frac{\mu_1\mu_2\times(l_1+l_2)}{(l_1\mu_2+l_2\mu_1)}, \quad \mu \text{ normal to media boundary}. \tag{3.7b}$$

These schemes serve as good approximations for low values of σ^e and σ^m as well, where they can be written as

$$\sigma^e_{eff} = \frac{\sigma^e_1 A_1 + \sigma^e_2 A_2}{(A_1 + A_2)}, \quad \sigma^e \text{ parallel to media boundary}, \tag{3.8a}$$

$$\sigma^e_{eff} = \frac{\sigma^e_1 \sigma^e_2 \times (l_1 + l_2)}{(l_1 \sigma^e_2 + l_2 \sigma^e_1)}, \quad \sigma^e \text{ normal to media boundary}, \tag{3.8b}$$

and

$$\sigma^m_{eff} = \frac{\sigma^m_1 A_1 + \sigma^m_2 A_2}{(A_1 + A_2)}, \quad \sigma^m \text{ parallel to media boundary}, \tag{3.9a}$$

$$\sigma^m_{eff} = \frac{\sigma^m_1 \sigma^m_2 \times (l_1 + l_2)}{(l_1 \sigma^m_2 + l_2 \sigma^m_1)}, \quad \sigma^m \text{ normal to media boundary}, \tag{3.9b}$$

3.4 Defining objects snapped to the Yee grid

The averaging schemes provided in the previous section are usually discussed in the context of subcell modeling techniques in which effective material parameter values are introduced while the staircased gridding scheme is maintained. Although there are some other more advanced subcell modeling techniques that have been introduced for modeling fine features in the FDTD method, in this section we discuss modeling of three-dimensional objects assuming that the objects are conforming to the staircased Yee grid. Therefore, each Yee cell is filled with a single material type. Although this assumption gives a crude model compared with the subcell modeling techniques discussed before, we consider this case since it does not require complicated programming effort and since it is sufficient to obtain reliable results for many problems. Furthermore, we still employ the aforementioned subcell averaging schemes for obtaining effective values for the material components lying on the boundaries of cells filled with different material types.

For instance, consider Figure 3.7, where the material component $\varepsilon_z(i, j, k)$ is located in between four Yee cells, each of which is filled with a material type. Every Yee cell is indexed by a respective node. Since every cell is filled with a material type, three-dimensional

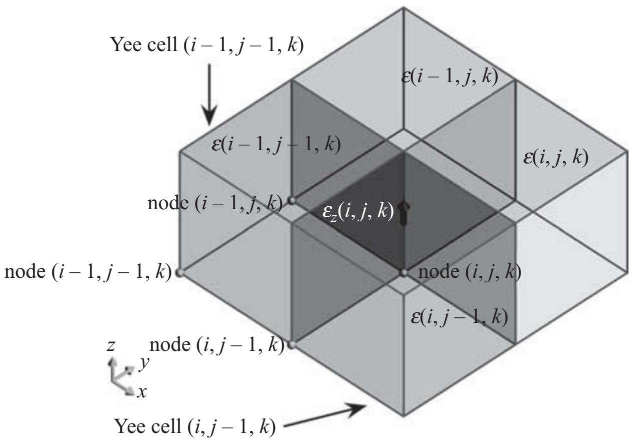


Figure 3.7 Material component $\varepsilon_z(i, j, k)$ located between four Yee cells filled with four different material types.

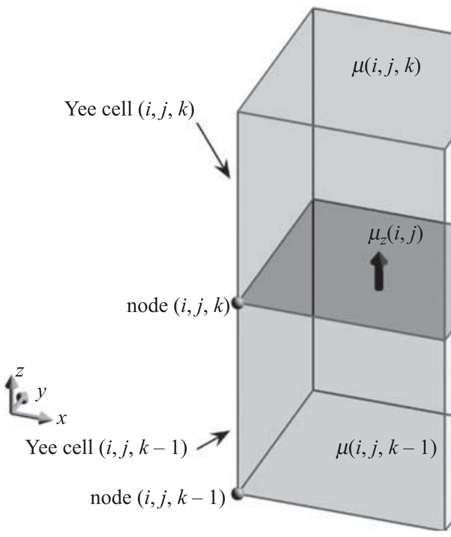


Figure 3.8 Material component $\mu_z(i, j, k)$ located between two Yee cells filled with two different material types.

material arrays can be utilized, each element of which corresponds to the material type filling the respective cell. For instance, $\varepsilon(i-1, j-1, k)$ holds the permittivity value of the material filling the cell $(i-1, j-1, k)$. Consider the permittivity parameters that are indexed as $\varepsilon(i-1, j-1, k)$, $\varepsilon(i-1, j, k)$, $\varepsilon(i, j-1, k)$, and $\varepsilon(i, j, k)$ in Figure 3.7. The material component $\varepsilon_z(i, j, k)$ is tangential to the four surrounding media types; therefore, we can employ (3.2) to get the equivalent permittivity $\varepsilon_z(i, j, k)$ as

$$\varepsilon_z(i, j, k) = \frac{\varepsilon(i, j, k) + \varepsilon(i-1, j, k) + \varepsilon(i, j-1, k) + \varepsilon(i-1, j-1, k)}{4}. \tag{3.10}$$

Since the electric conductivity material components are defined at the same positions as the permittivity components, the same averaging scheme can be employed to get the equivalent electric conductivity $\sigma_z^e(i, j, k)$ as

$$\sigma_z^e(i, j, k) = \frac{\sigma^e(i, j, k) + \sigma^e(i-1, j, k) + \sigma^e(i, j-1, k) + \sigma^e(i-1, j-1, k)}{4}. \tag{3.11}$$

Unlike the permittivity and electric conductivity components, the material components associated with magnetic field components, permeability and magnetic conductivity, are located between two cells and are oriented normal to the cell boundaries as illustrated in Figure 3.8. In this case we can use the relation (3.7b) to get $\mu_z(i, j, k)$ as

$$\mu_z(i, j, k) = \frac{2 \times \mu(i, j, k) \times \mu(i, j, k-1)}{\mu(i, j, k) + \mu(i, j, k-1)}. \tag{3.12}$$

Similarly, we can write $\sigma_z^m(i, j, k)$ as

$$\sigma_z^m(i, j, k) = \frac{2 \times \sigma^m(i, j, k) \times \sigma^m(i, j, k-1)}{\sigma^m(i, j, k) + \sigma^m(i, j, k-1)}. \tag{3.13}$$

3.4.1 Defining zero-thickness PEC objects

In many electromagnetics problems, especially in planar circuits, some very thin objects exist, most of the time in the form of microstrip or stripline structures. In these cases, when constructing an FDTD simulation, it may not be feasible to choose the unit cell size as small as the thickness of the strip, since this will lead to a very large number of cells and, hence to an excessive amount of computer memory that prevents practical simulations with current computer resources. Then if the cell size is larger than the strip thickness, subcell modeling techniques, some of which are discussed in the previous sections, can be employed for accurate modeling of the thin strips in the FDTD method. Usually the subcell modeling of thin strips is studied in the context of *thin-sheet* modeling techniques. Here we discuss a simple modeling technique that can be used to obtain results with a reasonable accuracy for the majority of problems including thin strips where the thin strips are PEC. Here we assume that the thin-strip thickness is zero, which is a reasonable approximation since most of the time the strip thicknesses are much smaller than the cell sizes. Furthermore, we assume that the strips are conforming to the faces of the Yee cells in the problem space.

Consider the PEC plate with zero thickness as shown in Figure 3.9, which conforms to the face of the Yee cell with index (i, j, k). This plate is surrounded by four electric conductivity material components, namely, $\sigma_x^e(i, j, k)$, $\sigma_x^e(i, j+1, k)$, $\sigma_y^e(i, j, k)$, and $\sigma_y^e(i+1, j, k)$. We can simply assign the conductivity of PEC, σ_{pec}^e, to these material components, such that

$$\begin{aligned} &\sigma_x^e(i, j, k) = \sigma_{pec}^e, && \sigma_x^e(i, j+1, k) = \sigma_{pec}^e, \\ &\sigma_y^e(i, j, k) = \sigma_{pec}^e, \quad \text{and} && \sigma_y^e(i+1, j, k) = \sigma_{pec}^e. \end{aligned}$$

Therefore, any electric conductivity components surrounding and coinciding with PEC plates can be assigned the conductivity of PEC to model zero-thickness PEC plates in the FDTD method.

Similarly, if a PEC object with thickness smaller than a cell size in two dimensions, such as a thin conductor wire, needs to be modeled in the FDTD method, it is sufficient to assign the conductivity of PEC to the electric conductivity components coinciding with the object.

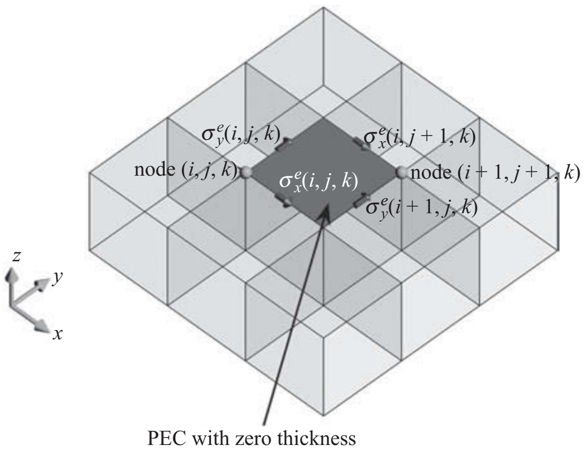


Figure 3.9 A PEC plate with zero thickness surrounded by four σ^e material components.

This approach gives a reasonable approximation to a thin wire in the FDTD method. However, since the field variation around a thin wire is large, the sampling of the fields at the discrete points around the wire may not be accurate. Therefore, it is more appropriate to use more advanced *thin-wire modeling* techniques, which take into account the wire thickness, for better accuracy. A thin-wire modeling technique is discussed in Chapter 10.

3.5 Creation of the material grid

At the beginning of this chapter we discussed how we can define the objects using data structures in MATLAB. We use these structures to initialize the FDTD material grid; we create and initialize three-dimensional arrays representing the components of the material parameters ε, μ, σ^e, and σ^m. Then we assign appropriate values to these arrays using the averaging schemes discussed in Section 3.4. Later on these material arrays will be used to calculate the FDTD updating coefficients.

The routine ***initialize_fdtd_material_grid***, which is used to initialize the material arrays, is given in Listing 3.4. The material array initialization process is composed of multiple stages; therefore, ***initialize_fdtd_material_grid*** is divided into functional subroutines. We describe the implementation of these subroutines and demonstrate how the concepts described in Section 3.4 are coded.

The first subroutine in Listing 3.4 is **calculate_domain_size**, the implementation of which is given in Listing 3.5. In this subroutine the dimensions of the FDTD problem space are determined, including the number of cells (*Nx*, *Ny*, and *Nz*) making the problem space. Once the number of cells in the *x*, *y*, and *z* dimensions are determined, they are assigned to

Listing 3.4 initialize_fdtd_material_grid.m

```
disp('initializing FDTD material grid');

% calculate problem space size based on the object
% locations and boundary conditions
calculate_domain_size;

% Array to store material type indices for every cell
% in the problem space. By default the space is filled
% with air by initializing the array by ones
material_3d_space = ones(nx, ny, nz);

% Create the 3D objects in the problem space by
% assigning indices of material types in the cells
% to material_3d_space

% create spheres
create_spheres;

% create bricks
create_bricks;

% Material component arrays for a problem space
% composed of (nx, ny, nz) cells
eps_r_x     = ones (nx   , nyp1 , nzp1);
eps_r_y     = ones (nxp1 , ny   , nzp1);
eps_r_z     = ones (nxp1 , nyp1 , nz);
```

```
mu_r_x      = ones (nxp1, ny   , nz);
mu_r_y      = ones (nx  , nyp1 , nz);
mu_r_z      = ones (nx  , ny   , nzp1);
sigma_e_x   = zeros(nx  , nyp1 , nzp1);
sigma_e_y   = zeros(nxp1, ny   , nzp1);
sigma_e_z   = zeros(nxp1, nyp1 , nz);
sigma_m_x   = zeros(nxp1, ny   , nz);
sigma_m_y   = zeros(nx  , nyp1 , nz);
sigma_m_z   = zeros(nx  , ny   , nzp1);

% calculate material component values by averaging
calculate_material_component_values;

% create zero thickness PEC plates
create_PEC_plates;
```

Listing 3.5 calculate_domain_size.m

```
disp('calculating_the_number_of_cells_in_the_problem_space');

number_of_spheres = size(spheres,2);
number_of_bricks  = size(bricks,2);

% find the minimum and maximum coordinates of a
% box encapsulating the objects
number_of_objects = 1;
for i=1:number_of_spheres
    min_x(number_of_objects) = spheres(i).center_x - spheres(i).radius;
    min_y(number_of_objects) = spheres(i).center_y - spheres(i).radius;
    min_z(number_of_objects) = spheres(i).center_z - spheres(i).radius;
    max_x(number_of_objects) = spheres(i).center_x + spheres(i).radius;
    max_y(number_of_objects) = spheres(i).center_y + spheres(i).radius;
    max_z(number_of_objects) = spheres(i).center_z + spheres(i).radius;
    number_of_objects = number_of_objects + 1;
end
for i=1:number_of_bricks
    min_x(number_of_objects) = bricks(i).min_x;
    min_y(number_of_objects) = bricks(i).min_y;
    min_z(number_of_objects) = bricks(i).min_z;
    max_x(number_of_objects) = bricks(i).max_x;
    max_y(number_of_objects) = bricks(i).max_y;
    max_z(number_of_objects) = bricks(i).max_z;
    number_of_objects = number_of_objects + 1;
end

fdtd_domain.min_x = min(min_x);
fdtd_domain.min_y = min(min_y);
fdtd_domain.min_z = min(min_z);
fdtd_domain.max_x = max(max_x);
fdtd_domain.max_y = max(max_y);
fdtd_domain.max_z = max(max_z);

```

```
% Determine the problem space boundaries including air buffers
fdtd_domain.min_x = fdtd_domain.min_x ...
    - dx * boundary.air_buffer_number_of_cells_xn;
fdtd_domain.min_y = fdtd_domain.min_y ...
    - dy * boundary.air_buffer_number_of_cells_yn;
fdtd_domain.min_z = fdtd_domain.min_z ...
    - dz * boundary.air_buffer_number_of_cells_zn;
fdtd_domain.max_x = fdtd_domain.max_x ...
    + dx * boundary.air_buffer_number_of_cells_xp;
fdtd_domain.max_y = fdtd_domain.max_y ...
    + dy * boundary.air_buffer_number_of_cells_yp;
fdtd_domain.max_z = fdtd_domain.max_z ...
    + dz * boundary.air_buffer_number_of_cells_zp;

% Determining the problem space size
fdtd_domain.size_x = fdtd_domain.max_x - fdtd_domain.min_x;
fdtd_domain.size_y = fdtd_domain.max_y - fdtd_domain.min_y;
fdtd_domain.size_z = fdtd_domain.max_z - fdtd_domain.min_z;

% number of cells in x, y, and z directions
nx = round(fdtd_domain.size_x/dx);
ny = round(fdtd_domain.size_y/dy);
nz = round(fdtd_domain.size_z/dz);

% adjust domain size by snapping to cells
fdtd_domain.size_x = nx * dx;
fdtd_domain.size_y = ny * dy;
fdtd_domain.size_z = nz * dz;

fdtd_domain.max_x = fdtd_domain.min_x + fdtd_domain.size_x;
fdtd_domain.max_y = fdtd_domain.min_y + fdtd_domain.size_y;
fdtd_domain.max_z = fdtd_domain.min_z + fdtd_domain.size_z;

% some frequently used auxiliary parameters
nxp1 = nx+1;    nyp1 = ny+1;    nzp1 = nz+1;
nxm1 = nx-1;    nxm2 = nx-2;    nym1 = ny-1;
nym2 = ny-2;    nzm1 = nz-1;    nzm2 = nz-2;

% create arrays storing the center coordinates of the cells
fdtd_domain.cell_center_coordinates_x = zeros(nx,ny,nz);
fdtd_domain.cell_center_coordinates_y = zeros(nx,ny,nz);
fdtd_domain.cell_center_coordinates_z = zeros(nx,ny,nz);
for ind = 1:nx
    fdtd_domain.cell_center_coordinates_x(ind,:,:) = ...
        (ind - 0.5) * dx + fdtd_domain.min_x;
end
for ind = 1:ny
    fdtd_domain.cell_center_coordinates_y(:,ind,:) = ...
       (ind - 0.5) * dy + fdtd_domain.min_y;
end
for ind = 1:nz
    fdtd_domain.cell_center_coordinates_z(:,:,ind) = ...
        (ind - 0.5) * dz + fdtd_domain.min_z;
end
```

parameters **nx**, **ny**, and **nz**. Based on the discussion in Section 3.4, we assume that every Yee cell is filled with a material type. Then in line 10 of Listing 3.4 a three-dimensional array **material_3d_space** with size ($Nx \times Ny \times Nz$) is initialized with ones. In this array each element will store the index of the material type filling the corresponding cell in the problem space. Since **material_3d_space** is initialized by the MATLAB function **ones**, all of its elements have the value 1. Referring to Listing 3.2 one can see that **material_types(1)** is reserved for *air*; therefore, the problem space is initially filled with air.

After this point we find the cells overlapping with the objects defined in ***define_geometry*** and assign the indices of the material types of the objects to the corresponding elements of the array **material_3d_space**. This procedure is performed in the subroutines ***create_spheres***, which is shown in Listing 3.6, and ***create_bricks***, which is shown in Listing 3.7. With the given implementations, first the spheres are created in the problem space, and then the bricks are created. Furthermore, the objects will be created in the sequence as they are defined in

Listing 3.6 create_spheres.m

```
disp('creating spheres');

cx = fdtd_domain.cell_center_coordinates_x;
cy = fdtd_domain.cell_center_coordinates_y;
cz = fdtd_domain.cell_center_coordinates_z;

for ind=1:number_of_spheres
% distance of the centers of the cells from the center of the sphere
    distance = sqrt((spheres(ind).center_x - cx).^2 ...
             + (spheres(ind).center_y - cy).^2 ...
             + (spheres(ind).center_z - cz).^2);
    I = find(distance<=spheres(ind).radius);
    material_3d_space(I) = spheres(ind).material_type;
end
clear cx cy cz;
```

Listing 3.7 create_bricks.m

```
disp('creating bricks');

for ind = 1:number_of_bricks
    % convert brick end coordinates to node indices
    blx = round((bricks(ind).min_x - fdtd_domain.min_x)/dx) + 1;
    bly = round((bricks(ind).min_y - fdtd_domain.min_y)/dy) + 1;
    blz = round((bricks(ind).min_z - fdtd_domain.min_z)/dz) + 1;

    bux = round((bricks(ind).max_x - fdtd_domain.min_x)/dx)+1;
    buy = round((bricks(ind).max_y - fdtd_domain.min_y)/dy)+1;
    buz = round((bricks(ind).max_z - fdtd_domain.min_z)/dz)+1;

    % assign material type of the brick to the cells
    material_3d_space (blx:bux-1, bly:buy-1, blz:buz-1) ...
        = bricks(ind).material_type;
end
```

define_geometry. This means that if more than one object overlaps in the same cell, the object defined last will overwrite the objects defined before. So one should define the objects accordingly. These two subroutines (**create_spheres** and **create_bricks**) can be replaced by a more advanced algorithm in which the objects can be assigned priorities in case of overlapping, and changing the priorities would be sufficient to have the objects created in the desired sequence in the problem space.

For example, the array **material_3d_space** is filled with the material parameter indices of the objects that were defined in Listing 3.3 and displayed in Figure 3.2. The cells represented by **material_3d_space** are plotted in Figure 3.10, which clearly shows the staircased approximations of the objects as generated by the subroutine ***display_problem_space***, which is called in ***fdtd_solve***.

Then in ***initialize_fdtd_material_grid*** we create three-dimensional arrays representing the material parameter components ε_x, ε_y, ε_z, μ_x, μ_y, μ_z, σ_x^e, σ_y^e, σ_z^e, σ_x^m, σ_y^m, and σ_z^m. Here the parameters **eps_r_x**, **eps_r_y**, **eps_r_z**, **mu_r_x**, **mu_r_y**, and **mu_r_z** are relative permittivity

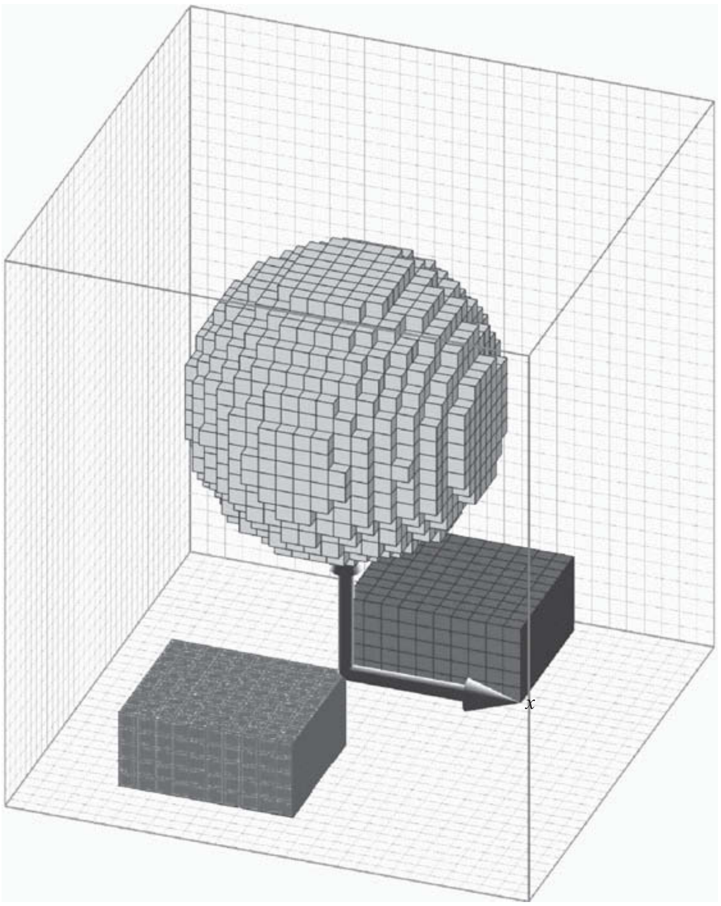


Figure 3.10 An FDTD problem space and the objects approximated by snapping to cells.

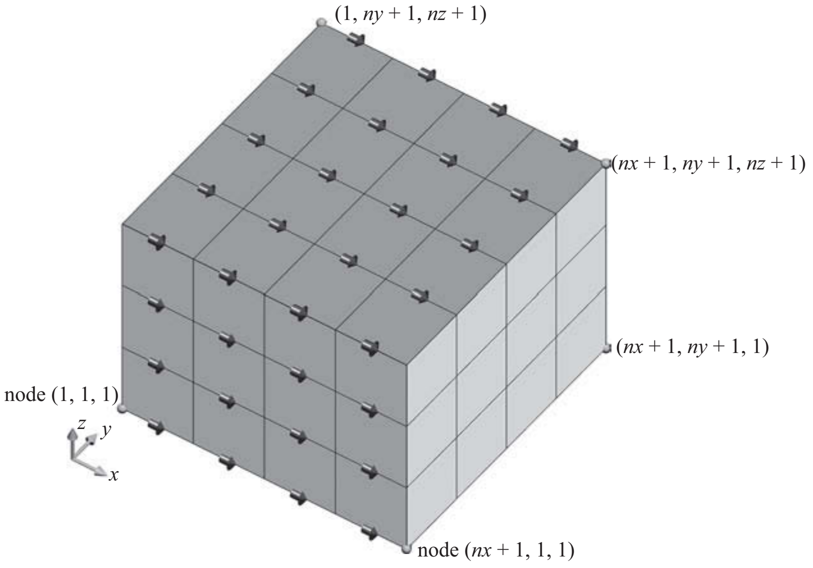


Figure 3.11 Positions of the E_x components on the Yee grid for a problem space composed of $(Nx \times Ny \times Nz)$ cells.

and relative permeability arrays, and they are initialized with 1. The electric and magnetic conductivity arrays are initialized with 0. One can notice that the sizes of these arrays are not the same as they correspond to the size of the associated field component. This is due to the specific positioning scheme of the field components on the Yee cell as shown in Figure 1.5. For instance, the distribution of the x components of the electric field, E_x, in a problem space composed of $(Nx \times Ny \times Nz)$ cells is illustrated in Figure 3.11. It can be observed that there exist $(Nx \times Ny + 1 \times Nz + 1)$ E_x components in the problem space. Therefore, in the program the three-dimensional arrays representing E_x and the material components associated with E_x shall be constructed with size $(Nx \times Ny + 1 \times Nz + 1)$. Hence, in Listing 3.4 the parameters **eps_r_x** and **sigma_e_x** are constructed with size (*nx*, *nyp* 1, *nzp* 1), where *nyp* 1 is $Ny + 1$, and *nzp* 1 is $Nz + 1$. Similarly, it can be observed in Figure 3.12 that there exist $(Nx + 1 \times Ny \times Nz)$ H_x components in the problem space. Therefore, in Listing 3.4 the parameters **mu_r_x** and **sigma_m_x** are constructed with size (*nxp* 1, *ny*, *nz*), where *nxp* 1 is $Nx + 1$.

Since we have created and filled the array **material_3d_space** as the staircased representation of the FDTD problem space and have initialized material component arrays, we can proceed with assigning appropriate parameter values to the material component array elements based on the averaging schemes discussed in Section 3.4 to form the material grid. This is done in the subroutine ***calculate_material_component_values***; partial implementation of this subroutine is shown in Listing 3.8. In Listing 3.8 calculations for the x components of the material component arrays are illustrated; implementation of other components is left to the reader. Here one can notice that **eps_r_x**, **sigma_e_x**, **mu_r_x**, and **sigma_m_x** are calculated using the forms of equations (3.10), (3.11), (3.12), and (3.13), respectively.

The last step in constructing the material grid is to create the zero-thickness PEC plates in the material grid. The subroutine ***create_PEC_plates*** shown in Listing 3.9 is implemented for

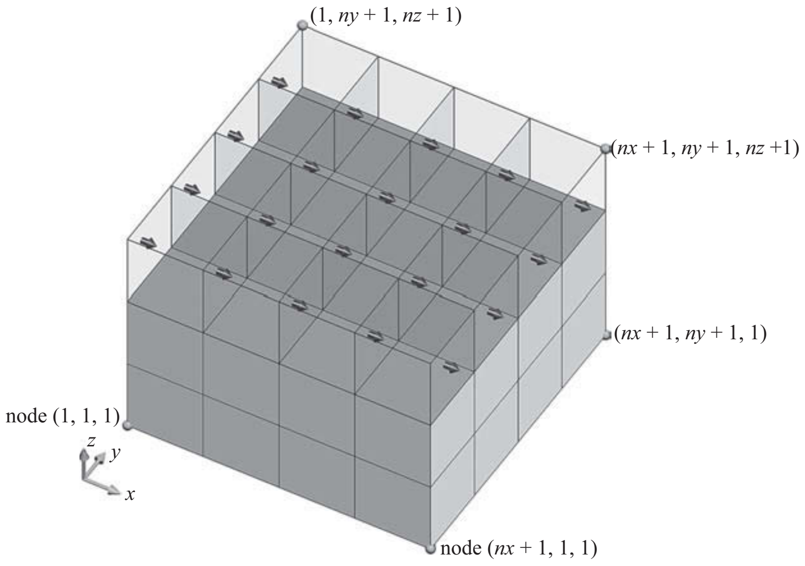


Figure 3.12 Positions of the H_x components on the Yee grid for a problem space composed of $(Nx \times Ny \times Nz)$ cells.

Listing 3.8 calculate_material_component_values.m

```
disp('filling material components arrays');

% creating temporary 1D arrays for storing
% parameter values of material types
for ind = 1:size(material_types,2)
    t_eps_r(ind)    = material_types(ind).eps_r;
    t_mu_r(ind)     = material_types(ind).mu_r;
    t_sigma_e(ind) = material_types(ind).sigma_e;
    t_sigma_m(ind) = material_types(ind).sigma_m;
end

% assign negligibly small values to t_mu_r and t_sigma_m where they are
% zero in order to prevent division by zero error
t_mu_r(find(t_mu_r==0)) = 1e-20;
t_sigma_m(find(t_sigma_m==0)) = 1e-20;

disp('Calculating eps_r_x');
% eps_r_x(i,j,k) is average of four cells
% (i,j,k),(i,j-1,k), (i,j,k-1), (i,j-1,k-1)
eps_r_x(1:nx,2:ny,2:nz) = ...
    0.25 * (t_eps_r(material_3d_space(1:nx,2:ny,2:nz)) ...
    + t_eps_r(material_3d_space(1:nx,1:ny-1,2:nz)) ...
    + t_eps_r(material_3d_space(1:nx,2:ny,1:nz-1)) ...
    + t_eps_r(material_3d_space(1:nx,1:ny-1,1:nz-1)));
disp('Calculating sigma_e_x');
% sigma_e_x(i,j,k) is average of four cells
% (i,j,k),(i,j-1,k), (i,j,k-1), (i,j-1,k-1)
```

```
sigma_e_x(1:nx,2:ny,2:nz) = ...
    0.25 * (t_sigma_e(material_3d_space(1:nx,2:ny,2:nz)) ...
    + t_sigma_e(material_3d_space(1:nx,1:ny-1,2:nz)) ...
    + t_sigma_e(material_3d_space(1:nx,2:ny,1:nz-1)) ...
    + t_sigma_e(material_3d_space(1:nx,1:ny-1,1:nz-1)));
disp('Calculating mu_r_x');
% mu_r_x(i,j,k) is average of two cells (i,j,k),(i-1,j,k)
mu_r_x(2:nx,1:ny,1:nz) = ...
    2 * (t_mu_r(material_3d_space(2:nx,1:ny,1:nz)) ...
    .* t_mu_r(material_3d_space(1:nx-1,1:ny,1:nz))) ...
    ./(t_mu_r(material_3d_space(2:nx,1:ny,1:nz)) ...
    + t_mu_r(material_3d_space(1:nx-1,1:ny,1:nz)));
disp('Calculating sigma_m_x');
% sigma_m_x(i,j,k) is average of two cells (i,j,k),(i-1,j,k)
sigma_m_x(2:nx,1:ny,1:nz) = ...
    2 * (t_sigma_m(material_3d_space(2:nx,1:ny,1:nz)) ...
    .* t_sigma_m(material_3d_space(1:nx-1,1:ny,1:nz))) ...
    ./(t_sigma_m(material_3d_space(2:nx,1:ny,1:nz)) ...
    + t_sigma_m(material_3d_space(1:nx-1,1:ny,1:nz)));
```

Listing 3.9 create_PEC_plates.m

```
disp('creating PEC plates on the material grid');

for ind = 1:number_of_bricks

    mtype = bricks(ind).material_type;
    sigma_pec = material_types(mtype).sigma_e;

    % convert coordinates to node indices on the FDTD grid
    blx = round((bricks(ind).min_x - fdtd_domain.min_x)/dx)+1;
    bly = round((bricks(ind).min_y - fdtd_domain.min_y)/dy)+1;
    blz = round((bricks(ind).min_z - fdtd_domain.min_z)/dz)+1;

    bux = round((bricks(ind).max_x - fdtd_domain.min_x)/dx)+1;
    buy = round((bricks(ind).max_y - fdtd_domain.min_y)/dy)+1;
    buz = round((bricks(ind).max_z - fdtd_domain.min_z)/dz)+1;

    % find the zero thickness bricks
    if (blx == bux)
        sigma_e_y(blx, bly:buy-1,blz:buz) = sigma_pec;
        sigma_e_z(blx, bly:buy,blz:buz-1) = sigma_pec;
    end
    if (bly == buy)
        sigma_e_z(blx:bux, bly, blz:buz-1) = sigma_pec;
        sigma_e_x(blx:bux-1, bly, blz:buz) = sigma_pec;
    end
    if (blz == buz)
        sigma_e_x(blx:bux-1,bly:buy, blz) = sigma_pec;
        sigma_e_y(blx:bux,bly:buy-1,blz)  = sigma_pec;
    end
end
```

this purpose. The code checks all of the defined bricks for zero thickness and assigns their material's conductivity values to the electric conductivity components overlapping with them.

Using the code sections described in this section, the material grid for the material cell distribution in Figure 3.10 is obtained and plotted on three plane cuts for relative permittivity distribution in Figure 3.13(a) and for relative permeability distribution in Figure 3.13(b). The effects of averaging can be clearly observed on the object boundaries.

3.6 Improved eight-subcell averaging

In Section 3.2 we described how any cell can be divided into eight subcells and how every material parameter component is located in between eight surrounding cells as illustrated in Figure 3.4. A simple averaging scheme as (3.1) was used to find an equivalent value for a given material component. We can combine the averaging schemes discussed in Section 3.3 with the eight-subcell modeling scheme to obtain an improved approach for modeling the material components in the FDTD grid.

Consider the eight subcells in Figure 3.4, each of which is filled with a material type. We can assume an equivalent permittivity ε_p for the four subcells with permittivities ε_1, ε_4, ε_5, and ε_8, which can be calculated as $\varepsilon_p = 0.25 \times (\varepsilon_1 + \varepsilon_4 + \varepsilon_5 + \varepsilon_8)$. Similarly, we can assume an equivalent permittivity ε_n for the four subcells with permittivities ε_2, ε_3, ε_6, and ε_7 that can be calculated as $\varepsilon_n = 0.25 \times (\varepsilon_2 + \varepsilon_3 + \varepsilon_6 + \varepsilon_7)$. The permittivity component $\varepsilon_z(i, j, k)$ is located between the half-cells filled with materials of permittivities ε_p and ε_n, and it is oriented normal to the boundaries of these half-cells. Therefore, using (3.6) we can write

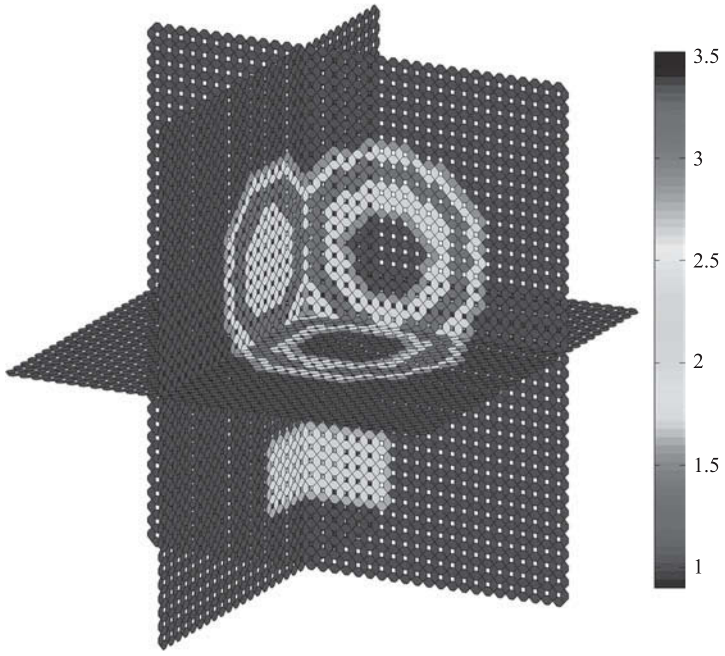
$$\varepsilon_z(i, j, k) = \frac{2 \times \varepsilon_p \times \varepsilon_n}{\varepsilon_p + \varepsilon_n}. \tag{3.14}$$

The same approach can be used for parameter components permeability and for electric and magnetic conductivities as well. Furthermore, it should be noted that the improved eight-subcell averaging scheme presented here is based on the application of the more general approach for eight subcells [8].

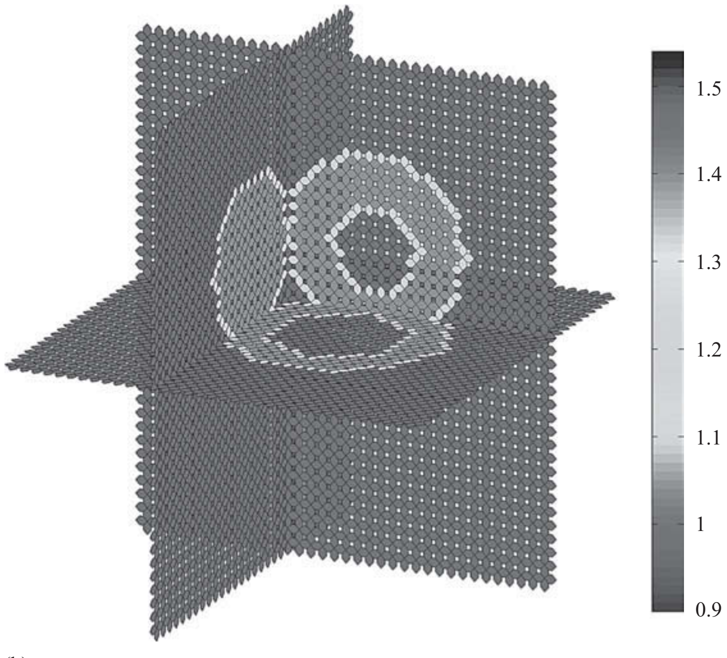
3.7 Exercises

3.1 The file **fdtd_solve** is the main program used to run FDTD simulations. Open this subroutine and examine its contents. It calls several subroutines that are not implemented yet. In the code disable the lines 7, 8, and 14–27 by "commenting" these lines. Simply insert "%" sign in front of these lines. Now the code is ready to run with its current functionality.

Open the subroutine ***define_problem_space-parameters***, and define the problem space parameters. Set the cell size as 1 mm on a side, boundaries as "***pec***," and air gap between the objects and boundaries as five cells on all sides. Define a material type with index 4 having the relative permittivity as 4, relative permeability as 2, electric conductivity as 0.2, and magnetic conductivity as 0.1. Define another material type with index 5 having relative permittivity as 6, relative permeability as 3, electric conductivity as 0.4, and magnetic conductivity as 0.2. Open the subroutine ***define_geometry***, and define the problem geometry. Define a brick with size



(a)



(b)

Figure 3.13 Material grid on three plane cuts: (a) relative permittivity components and (b) relative permeability components.

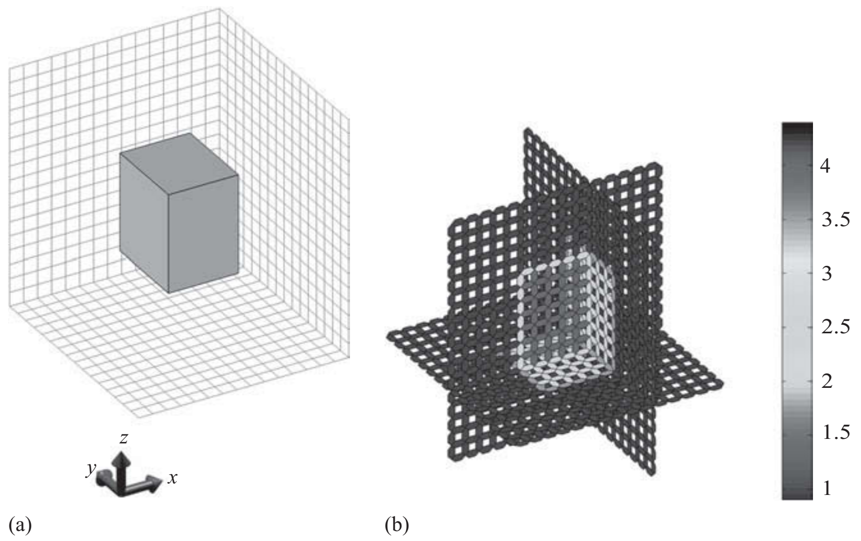


Figure 3.14 The FDTD problem space of Exercise 3.1: (a) geometry of the problem and (b) relative permittivity distribution.

5 mm × 6 mm × 7 mm, and assign material type index 4 as its material type. Figure 3.14(a) shows the geometry of the problem in consideration.

In the directory including your code files there are some additional files prepared as plotting routines that you can use together with your code. These routines are called in ***fdtd_solve*** after the definition routines. Insert the command "show_problem_space =true;" in line 9 of fdtd_solve temporarily to enable three-dimensional geometry display. Run the program ***fdtd_solve***. The program will open a figure including a three-dimensional view of the geometry you have defined and another program named as "Display Material Mesh" that helps you examine the material mesh created for the defined geometry. For instance, Figure 3.14(b) shows the relative permittivity distribution of the FDTD problem space in three plane cuts generated by the program "Display Material Mesh." Examine the geometry and the material mesh, and verify that the material parameter values are averaged on the boundaries as explained in Chapter 3.

3.2 Consider the problem you constructed in Exercise 3.1. Now define another brick with size 3 mm × 4 mm × 5 mm, assign material type index 5 as its material type, and place it at the center of the first brick. Run ***fdtd_solve***, and then examine the geometry and the material mesh and verify that the material parameter values are averaged on the boundaries as explained in Chapter 3.

Now change the definition sequence of these two bricks in ***define_geometry***; define the second brick first with index 1 and the first brick as second with its index 2. Run ***fdtd_solve***, and examine the geometry and the material mesh. Verify that the smaller brick has disappeared in the material mesh. With the given implementation of the program, the sequence of defining the objects determines the way the FDTD material mesh is created.

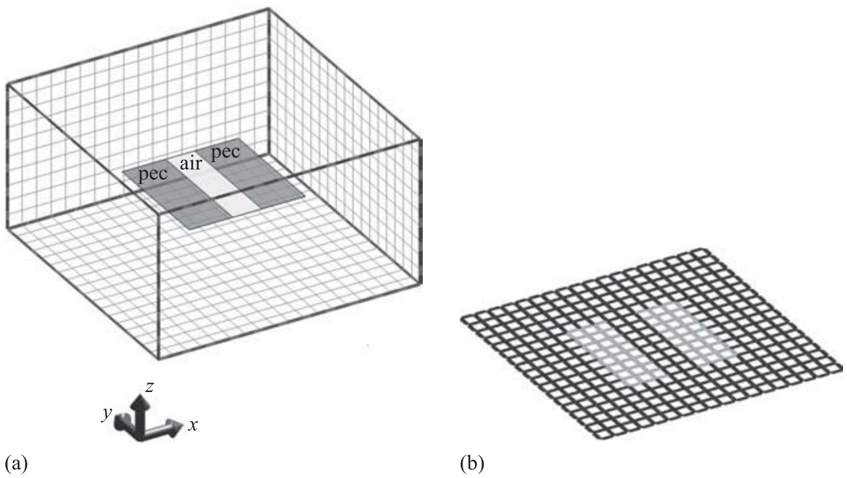


Figure 3.15 The FDTD problem space of Exercise 3.3: (a) geometry of the problem and (b) electric conductivity distribution.

3.3 Construct a problem space with the cell size as 1 mm on a side, boundaries as "***pec***," and air gap between the objects and boundaries as five cells on all sides. Define a brick as a PEC plate with zero thickness placed between the coordinates given in millimeters as (0, 0, 0) and (8, 8, 0). Define another brick as a plate with zero thickness, material type air, and placed between the coordinates (3, 0, 0) and (5, 8, 0) as illustrated in Figure 3.15(a), which shows the geometry of the problem in consideration. Run ***fdtd_solve***, and then examine the geometry and the material mesh. Examine the electric conductivity distribution, and verify that on the boundary between the plates the material components are air as shown in Figure 3.15(b).

3.4 Construct a problem space with the cell size as 1 mm on a side, boundaries as "***pec***," and air gap between the objects and boundaries as five cells on all sides. Define a brick as a PEC plate with zero thickness placed between the coordinates given in millimeters as (0, 0, 0) and (3, 8, 0). Define another brick as PEC plate and placed between the coordinates (5, 0, 0) and (8, 8, 0). Run ***fdtd_solve***, and then examine the geometry and the material mesh. Examine the electric conductivity distribution, and verify that on the boundaries of the plates facing each other the material components are PEC.

Although the geometry constructed in this example is physically the same as the one in Exercise 3.3, the material meshes are different since the ways the geometries are defined are different.