Chapter 7
A Matlab Edge Element Code for Metamaterials

In this chapter, we demonstrate the practical implementation of a mixed finite element method (FEM) for a 2-D Drude metamaterial model (5.1)–(5.4).

Let us recall that the basic procedures of using FEM to solve a partial differential equation (PDE):

1. Discretize the computational domain into finite elements;
2. Rewrite the PDE in a weak formulation, then choose proper finite element spaces and form the finite element scheme from the weak formulation;
3. Calculate those element matrices on each element;
4. Assemble element matrices to form a global linear system;
5. Implement the boundary conditions and solve the linear system;
6. Postprocess the numerical solution.

Compared to many books on finite element programming [21, 62, 158, 240, 252], there are only several books devoted to Maxwell's equations [42, 97, 98, 141, 162, 267]. To our best knowledge, no existing book provides complete source codes for solving time-domain Maxwell's equations using edge elements. Hence, in this chapter, we will present implementation details on using edge elements to solve the Drude metamaterial model (5.1)–(5.4). More specifically, in Sect. 7.1, we present a simple grid-generation algorithm and its implementation. Section 7.2 formulates the finite element scheme for the Drude model (5.1)–(5.4). In Sect. 7.3, we discuss how to calculate those element matrices involved. Then in Sect. 7.4 we discuss the finite element assembly procedure and how to implement the Dirichlet boundary condition. Since edge elements do not yield numerical solutions at mesh nodes automatically, in Sect. 7.5 we present a postprocessing step to retrieve the numerical solutions at element centers. Finally, in Sect. 7.6 we present an example problem to show how our algorithm gets implemented in MATLAB. Detailed MATLAB source codes with many comments are provided. We summarize this chapter in Sect. 7.7.

J. Li and Y. Huang, *Time-Domain Finite Element Methods for Maxwell's Equations in Metamaterials*, Springer Series in Computational Mathematics 43,
DOI 10.1007/978-3-642-33789-5_7, © Springer-Verlag Berlin Heidelberg 2013

7.1 Mesh Generation

For simplicity, we assume that the physical domain is a rectangle $\Omega \equiv [lowx, highx] \times [lowy, highy]$, which is subdivided into $nelex \times neley$ uniform rectangular elements. Here $nelex$ and $neley$ denote the numbers of elements in the x and y directions, respectively. A simple MATLAB code below accomplishes this task, where the x and y coordinates of all nodes are stored in the first and second rows of array $no2xy(1:2, 1:np)$, respectively, where np denotes the total number of points in the mesh.

```
dx=(highx-lowx)/nelex; dy=(highy-lowy)/neley;

nx = nelex+1; ny = neley+1;
np = (nelex+1)*(neley+1);  % total # of grid points
no2xy = zeros(2,np);
for j=1:ny
    for i=1:nx
        ipt=nx*(j-1)+i;
        no2xy(1,ipt)=dx*(i-1);
        no2xy(2,ipt)=dy*(j-1);
     end
end
```

Similar to the classical nodal based finite element method, we need to build up a connectivity matrix $el2no(i, j)$ to describe the relation between local nodes and global nodes. For the lowest-order rectangular edge element, $el2no(i, j)$ denotes the global label of the i-th node of the j-th element, where $i = 1, 2, 3, 4$, $j = 1, \cdots, numel$, and $numel$ denotes the total number of elements. For consistency, the four nodes of each element are ordered counterclockwise. This task is achieved by the following MATLAB code.

```
numel=(nelex)*(neley);  % total number of elements
el2no=zeros(4,numel);

idx=1;
for i=1:neley
  for j=1:nelex
     el2no(:,idx)=[j+(i-1)*nx; j+(i-1)*nx+1; \ldots
                   j+nx*i+1; j+nx*i];
     idx = idx+1;
  end
end
```

Since unknowns in edge element space are associated with edges in the mesh, we need to number the edges and associate an orientation direction with each edge. To do this, we assume that each edge is defined by its start and end points, and each

edge is assigned a global edge number. This task can be done efficiently based on a sorting technique originally proposed by Jin [162, p. 332] and implemented for triangular edge elements in MATLAB [42, p. 125]. Below is our implementation to create an array $el2ed(i, j)$, which stores the i-th edge of the j-th element, where $i = 1, \cdots, 4$, and $j = 1, \cdots, numel$.

```
% the total number of edges including boundary edges
numed=nelex*(ny)+neley*(nx);
for i=1:numel
   for j=1:4
      if (j==1 | j==2 | j==3)
         edges((i-1)*4+j,:)=[el2no(j,i) el2no(j+1,i)];
      else
         edges((i-1)*4+j,:)=[el2no(j,i) el2no(1,i)];
      end
   end
end

edges=sort(edges,2);
[ed2no,trash,el2ed]=unique(edges,'rows');
el2ed=reshape(el2ed,4,numel);
```

The complete MATLAB source code *create_mesh.m* is shown below:

```
function create_mesh

globals2D;

% give the rectangle info
lowx=0;  highx=1.0; lowy=0;  highy=1.0;
nelex=20; neley=20;
dx=(highx-lowx)/nelex; dy=(highy-lowy)/neley;

% generate a rectangular mesh
nx = nelex+1; % number of points in the x direction
ny = neley+1; % number of points in the y direction
np = nx*ny; % total number of grid points
no2xy = zeros(2,np);
for j=1:ny
    for i=1:nx
        ipt=nx*(j-1)+i;
        no2xy(1,ipt)=dx*(i-1);   no2xy(2,ipt)=dy*(j-1);
    end
end

numel=(nelex)*(neley);  % the number of total elements
% 4 nodes (counterclockwise) for each element!
el2no=zeros(4,numel);
idx=1;
for i=1:neley % number of columns to go through
    for j=1:nelex
```

```
        el2no(:,idx)=[j+(i-1)*nx; j+(i-1)*nx+1; ...
                      j+nx*i+1; j+nx*i];
        idx = idx+1;
    end
end

% the total number of edges including boundary edges
numed=nelex*(ny)+neley*(nx);
for i=1:numel
    for j=1:4 % for each element in each column
        if (j==1 | j==2 | j==3)
            edges((i-1)*4+j,:)=[el2no(j,i) el2no(j+1,i)];
        else
            edges((i-1)*4+j,:)=[el2no(j,i) el2no(1,i)];
        end
    end
end
edges=sort(edges,2);
[ed2no,trash,el2ed]=unique(edges,'rows');
el2ed=reshape(el2ed,4,numel);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Indicators: 1 for interior edges; 0 for boundary edges.
ed_id=zeros(numed,1);
for i=1:numed
    v1x = no2xy(1,ed2no(i,1));
    v1y = no2xy(2,ed2no(i,1));
    v2x = no2xy(1,ed2no(i,2));
    v2y = no2xy(2,ed2no(i,2));
    if (v1x==lowx & v2x==lowx) | (v1x==highx & v2x==highx) |...
            (v1y==lowy & v2y==lowy) | (v1y==highy & v2y==highy)
        ed_id(i)=1;
    end
end

eint = find(ed_id == 0); % get labels for all interior edges
iecnt = length(eint);    % total number of interior edges

% compare reference element edge directions vs global
% edge directions to get the orientations for all edges
%        3
%   ----<------
%   |          |
% 4 v          ^  2
%   |          |
%   ----->-----
%         1
edori = ones(numel,4);
for i=1:numel
 %   edori(i,:)=[1 1 -1 -1];
    for j=1:4
        edn = el2ed(j,i);
        n1=ed2no(edn,1); n2=ed2no(edn,2);
        if j < 4
```

```
          m1=el2no(j,i); m2=el2no(j+1,i);
        else
          m1=el2no(j,i); m2=el2no(1,i);
        end
        if m1 > m2
            edori(i,j)=-1;
        end
    end
end
return
```

7.2 The Finite Element Scheme

For the non-dimensionalized Drude model derived in Sect. 4.4 with added source terms $\mathbf{f}$ and $\mathbf{g}$, we can obtain its weak formulation: For any $t \in (0, T]$, find the solutions $\mathbf{E} \in H_0(\text{curl}; \Omega)$, $\mathbf{J} \in H(\text{curl}; \Omega)$, H and $K \in L^2(\Omega)$ such that

$$(\mathbf{E}_t, \boldsymbol{\phi}) - (\mathbf{H}, \nabla \times \boldsymbol{\phi}) + (\mathbf{J}, \boldsymbol{\phi}) = (\mathbf{f}, \boldsymbol{\phi}), \quad \forall\, \boldsymbol{\phi} \in H_0(\text{curl}; \Omega), \tag{7.1}$$

$$(H_t, \psi) + (\nabla \times \mathbf{E}, \psi) + (K, \psi) = (g, \psi), \quad \forall\, \psi \in L^2(\Omega), \tag{7.2}$$

$$(\mathbf{J}_t, \tilde{\boldsymbol{\phi}}) + \Gamma_e(\mathbf{J}, \tilde{\boldsymbol{\phi}}) - \omega_e^2(\mathbf{E}, \tilde{\boldsymbol{\phi}}) = 0, \quad \forall\, \tilde{\boldsymbol{\phi}} \in H(\text{curl}; \Omega), \tag{7.3}$$

$$(K_t, \tilde{\psi}) + \Gamma_m(K, \tilde{\psi}) - \omega_m^2(H, \tilde{\psi}) = 0, \quad \forall\, \tilde{\psi} \in L^2(\Omega), \tag{7.4}$$

subject to the perfect conducting boundary condition (3.59) and initial conditions (3.60) and (3.61).

To construct a finite element scheme for (7.1)–(7.4), we first discretize the physical domain Ω into rectangular elements $K \in T_h$. On this mesh T_h, we construct the lowest-order Raviart-Thomas-Nédélec finite element spaces:

$$\mathbf{U}_h = \{u_h \in L^2(\Omega) : \ u_h|_K \in Q_{0,0}, \quad \forall\, K \in T_h\}, \tag{7.5}$$

$$\mathbf{V}_h = \{\mathbf{v}_h \in H(\text{curl}; \Omega) : \ \mathbf{v}_h|_K \in Q_{0,1} \times Q_{1,0}, \quad \forall\, K \in T_h\}. \tag{7.6}$$

To take care of the perfect conducting boundary condition (3.59), we introduce a subspace of $\mathbf{V}_h$:

$$\mathbf{V}_h^0 = \{\mathbf{v}_h \in \mathbf{V}_h : \ \mathbf{v}_h \times \mathbf{n} = \mathbf{0} \ \text{ on } \partial\Omega\}.$$

Similar to (5.16)–(5.19), we can formulate a Crank-Nicolson mixed finite element scheme for solving (7.1)–(7.4): For $k \geq 1$, find $\mathbf{E}_h^k \in \mathbf{V}_h^0, \mathbf{J}_h^k \in \mathbf{V}_h, H_h^k, K_h^k \in \mathbf{U}_h$ such that

$$(\delta_\tau \mathbf{E}_h^k, \boldsymbol{\phi}_h) - (\overline{\mathbf{H}}_h^k, \nabla \times \boldsymbol{\phi}_h) + (\overline{\mathbf{J}}_h^k, \boldsymbol{\phi}_h) = (\mathbf{f}^{k-\frac{1}{2}}, \boldsymbol{\phi}_h), \ \forall\, \boldsymbol{\phi}_h \in \mathbf{V}_h^0, \tag{7.7}$$

$$(\delta_\tau H_h^k, \psi_h) + (\nabla \times \overline{\mathbf{E}}_h^k, \psi_h) + (\overline{K}_h^k, \psi_h) = (g^{k-\frac{1}{2}}, \psi_h), \ \forall\, \psi_h \in \mathbf{U}_h, \tag{7.8}$$

$$(\delta_\tau \mathbf{J}_h^k, \tilde{\boldsymbol{\phi}}_h) + \Gamma_e(\overline{\mathbf{J}}_h^k, \tilde{\boldsymbol{\phi}}_h) - \omega_e^2(\overline{\mathbf{E}}_h^k, \tilde{\boldsymbol{\phi}}_h) = 0, \quad \forall\ \tilde{\boldsymbol{\phi}}_h \in \mathbf{V}_h, \tag{7.9}$$

$$(\delta_\tau K_h^k, \tilde{\psi}_h) + \Gamma_m(\overline{K}_h^k, \tilde{\psi}_h) - \omega_m^2(\overline{H}_h^k, \tilde{\psi}_h) = 0, \quad \forall\ \tilde{\psi}_h \in \mathbf{U}_h, \tag{7.10}$$

subject to the initial approximations

$$\mathbf{E}_h^0(\mathbf{x}) = \Pi_h \mathbf{E}_0(\mathbf{x}), \quad H_h^0(\mathbf{x}) = P_h H_0(\mathbf{x}), \tag{7.11}$$

$$\mathbf{J}_h^0(\mathbf{x}) = \Pi_h \mathbf{J}_0(\mathbf{x}), \quad K_h^0(\mathbf{x}) = P_h K_0(\mathbf{x}). \tag{7.12}$$

As usual, we denote P_h for the standard $L^2(\Omega)$-projection operator onto $\mathbf{U}_h$, and Π_h for the Nédélec interpolation operator.

In practical implementation, we first solve (7.9) and (7.10) for $\mathbf{J}_h^k$ and K_h^k:

$$\mathbf{J}_h^k = \frac{2-\tau\Gamma_e}{2+\tau\Gamma_e}\mathbf{J}_h^{k-1} + \frac{\tau\omega_e^2}{2+\tau\Gamma_e}(\mathbf{E}_h^k + \mathbf{E}_h^{k-1}), \tag{7.13}$$

$$K_h^k = \frac{2-\tau\Gamma_m}{2+\tau\Gamma_m}K_h^{k-1} + \frac{\tau\omega_m^2}{2+\tau\Gamma_m}(H_h^k + H_h^{k-1}) \tag{7.14}$$

then substituting (7.13) and (7.14) into (7.7) and (7.8), respectively, we obtain

$$\begin{aligned}
(i) \quad & (1 + \frac{\tau^2\omega_e^2}{2(2+\tau\Gamma_e)})(\mathbf{E}_h^k, \boldsymbol{\phi}_h) - \frac{\tau}{2}(H_h^k, \nabla\times\boldsymbol{\phi}_h) \\
&= (1 - \frac{\tau^2\omega_e^2}{2(2+\tau\Gamma_e)})(\mathbf{E}_h^{k-1}, \boldsymbol{\phi}_h) + \frac{\tau}{2}(H_h^{k-1}, \nabla\times\boldsymbol{\phi}_h) \\
&\quad -\frac{2\tau}{2+\tau\Gamma_e}(\mathbf{J}_h^{k-1}, \boldsymbol{\phi}_h) + \tau(\mathbf{f}^{k-\frac{1}{2}}, \boldsymbol{\phi}_h), \\
(ii) \quad & (1 + \frac{\tau^2\omega_m^2}{2(2+\tau\Gamma_m)})(H_h^k, \psi_h) + \frac{\tau}{2}(\nabla\times\mathbf{E}_h^k, \psi_h) \\
&= (1 - \frac{\tau^2\omega_m^2}{2(2+\tau\Gamma_m)})(H_h^{k-1}, \psi_h) - \frac{\tau}{2}(\nabla\times\mathbf{E}_h^{k-1}, \psi_h) \\
&\quad -\frac{2\tau}{2+\tau\Gamma_m}(K_h^{k-1}, \psi_h) + \tau(g^{k-\frac{1}{2}}, \psi_h).
\end{aligned}$$

We can simply rewrite the above system as:

$$A\mathbf{E}^k - B\mathbf{H}^k = \tilde{\mathbf{f}}, \tag{7.15}$$

$$B'\mathbf{E}^k + C\mathbf{H}^k = \tilde{\mathbf{g}}, \tag{7.16}$$

where A, B and C represent the corresponding coefficient matrices. Here B' denote the transpose of B. Solving for $\mathbf{H}^k$ from (7.16), then substituting it into (7.15), we obtain

$$\mathbf{H}^k = C^{-1}(\tilde{\mathbf{g}} - B'\mathbf{E}^k), \quad \mathbf{E}^k = (A + BC^{-1}B')^{-1}(\tilde{\mathbf{f}} + BC^{-1}\tilde{\mathbf{g}}). \tag{7.17}$$

In summary, the algorithm can be implemented as follows: At each time step, we first solve for $\mathbf{E}_h^k$ from (7.17) , then H_h^k; and finally update $\mathbf{J}_h^k$ and K_h^k using (7.13) and (7.14), respectively.

7.3 Calculation of Element Matrices

On a rectangle $K = [x_a, x_b] \times [y_a, y_b]$, we use a scaled edge element basis functions for the space $\mathbf{V}_h$:

$$\hat{\mathbf{N}}_1 = \begin{pmatrix} \frac{y_b - y}{y_b - y_a} \\ 0 \end{pmatrix}, \quad \hat{\mathbf{N}}_2 = \begin{pmatrix} 0 \\ \frac{x - x_a}{x_b - x_a} \end{pmatrix}, \quad \hat{\mathbf{N}}_3 = \begin{pmatrix} \frac{y_a - y}{y_b - y_a} \\ 0 \end{pmatrix}, \quad \hat{\mathbf{N}}_4 = \begin{pmatrix} 0 \\ \frac{x - x_b}{x_b - x_a} \end{pmatrix},$$

where the edges are oriented counterclockwisely, starting from the bottom edge.

In (7.15), the matrix A is really a multiple of a global mass matrix $matM = (\mathbf{N}_j, \mathbf{N}_i)$, while B is a multiple of matrix $matBM = (1, \nabla \times \mathbf{N}_i)$. The matrix C in (7.16) is just a diagonal matrix, whose elements are the areas of all elements. Matrices $matM$ and $matBM$ can be constructed from the corresponding matrices on each element. These element matrices can be obtained directly by the following lemmas.

Lemma 7.1. *The mass matrix* $M^e = (M_{ij}^e) = (\int_{x_a}^{x_b} \int_{y_a}^{y_b} \mathbf{N}_i \cdot \mathbf{N}_j dxdy)$ *is given by*

$$M^e = \frac{(x_b - x_a)(y_b - y_a)}{6} \begin{bmatrix} 2 & 0 & -1 & 0 \\ 0 & 2 & 0 & -1 \\ -1 & 0 & 2 & 0 \\ 0 & -1 & 0 & 2 \end{bmatrix}.$$

Proof. It is easy to see that M^e is symmetric, and $M_{12}^e = M_{23}^e = M_{34}^e = 0$. Furthermore, we have

$$M_{11}^e = \int_{x_a}^{x_b} \int_{y_a}^{y_b} \left(\frac{y_b - y}{y_b - y_a} \right)^2 dxdy = \frac{1}{3}(x_b - x_a)(y_b - y_a),$$

$$M_{13}^e = \int_{x_a}^{x_b} \int_{y_a}^{y_b} \frac{(y_b - y)(y_a - y)}{(y_b - y_a)^2} dxdy = \frac{-1}{6}(x_b - x_a)(y_b - y_a),$$

and

$$M_{24}^e = \int_{x_a}^{x_b} \int_{y_a}^{y_b} \frac{(x - x_a)(x - x_b)}{(x_b - x_a)^2} dxdy = \frac{-1}{6}(x_b - x_a)(y_b - y_a),$$

which completes the proof. □

Lemma 7.2. *The corresponding element curl matrix* $B^e = (B_j^e) = (\int_{x_a}^{x_b}\int_{y_a}^{y_b} \nabla \times \boldsymbol{N}_j\, dxdy)$ *is given by*

$$B^e = [x_b - x_a \; y_b - y_a \; x_b - x_a \; y_b - y_a].$$

Proof. Note that

$$\begin{aligned} B_1^e &= \int_{x_a}^{x_b}\int_{y_a}^{y_b} \left(\frac{\partial \mathbf{N}_1^{(2)}}{\partial x} - \frac{\partial \mathbf{N}_1^{(1)}}{\partial y}\right) dxdy \\ &= \int_{x_a}^{x_b}\int_{y_a}^{y_b} \frac{1}{y_b - y_a} dxdy = x_b - x_a. \end{aligned}$$

Similarly, we can prove the other components. □

7.4 Assembly Process and Boundary Conditions

The global mass matrix $matM$ and curl matrix $matBM$ can be formed by assembling the contributions from each element matrix. More specifically, we just need to loop through all the edges of all elements in the mesh to find the global label for each edge, and put the contribution into the right location in the global matrix. This is different from the classical nodal-based finite element method, which needs to loop through all the nodes of all elements in the mesh. We like to remark that during the assembly process, the orientation of each edge (stored as ± 1 in array $edori(1 : numel, 1 : 4)$) needs to be considered before each component is added to the global matrix.

The detailed assembly process for both $matB$ and $matBM$ is realized in the following code.

```
for i=1:numel   % loop through elements
  for j=1:4     % loop through edges
    ed1 = el2ed(j,i);
    matBM(ed1,i) = matBM(ed1,i) + edori(i,j)*Curl(j);

    for k=j:4   % loop through edges
      ed2 = el2ed(k,i);
      matM(ed1,ed2) = matM(ed1,ed2) \ldots
                    + edori(i,j)*edori(i,k)*Mref(j,k);
      matM(ed2,ed1) = matM(ed1,ed2);
    end
  end
end
```

Since our boundary condition $\mathbf{n} \times \mathbf{E} = \mathbf{0}$ is a natural boundary condition, we don't have to impose it explicitly.

After assembly, we have to solve the system (7.17) for the unknown coefficients of electric field $\mathbf{E}$. Since the coefficient matrix is symmetric and well conditioned, we just use the simple direct solver provided by MATLAB. Interested readers can use more advanced solvers, such as the Generalized Minimal Residual (GRMES) method, the Bi-Conjugate Gradient (Bi-CG) method, the Bi-Conjugate Gradient Stabilized (Bi-CGSTAB) method [28], multigrid method and the preconditioner method [129, 136, 146].

The complete MATLAB source code *form_mass_matrix.m*, which accomplishes the construction of the global matrix, is shown below.

```
function [rhsEF,rhsEE,rhsEJ,rhsEH,rhsHH,rhsHK,rhsHG,H0,K0] = ...
          form_mass_matrix(HH,KK,gRHS,f1RHS,f2RHS,Ex,Ey,Jx,Jy)

globals2D;

numel=(nelex)*(neley);
one = ones(1,4);
rhsEF=zeros(numed,1);          % for (f,N_i)
rhsEE=zeros(numed,1);          % for (E0,N_i)
rhsEJ=zeros(numed,1);          % for (J0,N_i)
rhsEH=zeros(numed,1);    % for (H0,curl N_i)
rhsHH=zeros(numel,1);        % for (H0, psi_i)
rhsHK=zeros(numel,1);        % for (K0, psi_i)
rhsHG=zeros(numel,1);        % for (g, psi_i)
% store the initial value at each element center
H0 = zeros(numel,1);
K0 = zeros(numel,1);

matM = sparse(numed,numed);  % zero matrix of numedges x numedges
matBM = sparse(numed,numel);
area = zeros(numel,1);
for i=1:numel
   % coordinates of this element from 1st node & 3rd node
   xae=no2xy(1,el2no(1,i));  xbe=no2xy(1,el2no(3,i));
   yae=no2xy(2,el2no(1,i));  ybe=no2xy(2,el2no(3,i));

   midpt(i,1) = 0.5*(min(no2xy(1,el2no(:,i))) ...
                + max(no2xy(1,el2no(:,i))));
   midpt(i,2) = 0.5*(min(no2xy(2,el2no(:,i))) ...
                + max(no2xy(2,el2no(:,i))));
   H0(i) = HH(midpt(i,1),midpt(i,2),0);  % element center value
   K0(i) = KK(midpt(i,1),midpt(i,2),0);  % element center value
   rhs_g = gRHS(midpt(i,1),midpt(i,2),0.5*dt);

   % the coordinates of the four vertex
   xe(1)=xae; ye(1)=yae;
   xe(2)=xbe; ye(2)=yae;
   xe(3)=xbe; ye(3)=ybe;
   xe(4)=xae; ye(4)=ybe;
   area(i) = (ybe-yae)*(xbe-xae);   % for non-uniform rectangles
```

```
for j=1:4   % loop through edges
   ed1 = el2ed(j,i);

   % evaluate the RHS: \int_O fRHS * N_j
   % we used Gaussian integration: cf. my book p.190!
   rhs_ef=0;  rhs_ee=0;  rhs_ej=0;

   for ii=1:2  % loop over gauss points in eta
      for jj=1:2   % loop over gauss points in psi
        eta = gauss(ii);  psi = gauss(jj);
        % Q1 function: countcockwise starting at left-low corner
        NJ=0.25*(one + psi*psiJ).*(one + eta*etaJ);
        % derivatives of shape functions in reference coordinates
        NJpsi = 0.25*psiJ.*(one + eta*etaJ);    % 1x4 array
        NJeta = 0.25*etaJ.*(one + psi*psiJ);    % 1x4 array
        % derivatives of x and y wrt psi and eta
        xpsi = NJpsi*xe';  ypsi = NJpsi*ye';
        xeta = NJeta*xe';  yeta = NJeta*ye';
        % Jinv = [yeta, -xeta; -ypsi, xpsi];      % 2x2 array
        jcob = xpsi*yeta - xeta*ypsi;

        xhat=dot(xe,NJ);  yhat=dot(ye,NJ);

        if j==1
          bas1=(ybe-yhat)/(ybe-yae);
          rhs_ef = rhs_ef + f1RHS(xhat,yhat,0.5*dt)*bas1*jcob;
          rhs_ee = rhs_ee + Ex(xhat,yhat,0)*bas1*jcob;
          rhs_ej = rhs_ej + Jx(xhat,yhat,0)*bas1*jcob;
        elseif j==2
          bas2=(xhat-xae)/(xbe-xae);
          rhs_ef = rhs_ef + f2RHS(xhat,yhat,0.5*dt)*bas2*jcob;
          rhs_ee = rhs_ee + Ey(xhat,yhat,0)*bas2*jcob;
          rhs_ej = rhs_ej + Jy(xhat,yhat,0)*bas2*jcob;
        elseif j==3
          bas3=-(yhat-yae)/(ybe-yae);
          rhs_ef = rhs_ef + f1RHS(xhat,yhat,0.5*dt)*bas3*jcob;
          rhs_ee = rhs_ee + Ex(xhat,yhat,0)*bas3*jcob;
          rhs_ej = rhs_ej + Jx(xhat,yhat,0)*bas3*jcob;
        else
          bas4=-(xbe-xhat)/(xbe-xae);
          rhs_ef = rhs_ef + f2RHS(xhat,yhat,0.5*dt)*bas4*jcob;
          rhs_ee = rhs_ee + Ey(xhat,yhat,0)*bas4*jcob;
          rhs_ej = rhs_ej + Jy(xhat,yhat,0)*bas4*jcob;
        end
    end
   end

    % assemble the edge contribution into global rhs vector
    rhsEF(ed1)=rhsEF(ed1)+edori(i,j)*rhs_ef;
    rhsEE(ed1)=rhsEE(ed1)+edori(i,j)*rhs_ee;
    rhsEJ(ed1)=rhsEJ(ed1)+edori(i,j)*rhs_ej;
    rhsEH(ed1)=        edori(i,j)*H0(i)*Curl(j);

    matBM(ed1,i) = matBM(ed1,i) + edori(i,j)*Curl(j);
```

```
        for k=j:4
          ed2 = el2ed(k,i);
          matM(ed1,ed2) = matM(ed1,ed2) ...
                          + edori(i,j)*edori(i,k)*Mref(j,k);
          matM(ed2,ed1) = matM(ed1,ed2);
        end
    end   % end of 1st edge loop
    rhsHH(i)=H0(i)*area(i);
    rhsHK(i)=K0(i)*area(i);
    rhsHG(i)=rhs_g*area(i);
end
return
```

By similar techniques, we have to assemble the right hand side vector in each time step. This task is realized in the driver function *Drude_cn.m* shown in Sect. 7.6.

7.5 Postprocessing

Once we obtain the unknown coefficients of electric field $\mathbf{E}$, we can use them to construct the numerical electric field $\mathbf{E}_h$ at any point, which can be used to compare with the analytic electric field $\mathbf{E}$ for error estimates. This reconstruction part can be realized in the following code, where the numerical electric field $\mathbf{E}$ is calculated at each element center.

```
solvec = zeros(numed,1);
% extract the coefficients of E field
solvec(eint)=znew(1:iecnt);
for i=1:numel
    % coordinates of this element from 1st & 3rd nodes
    xae=no2xy(1,el2no(1,i));  xbe=no2xy(1,el2no(3,i));
    yae=no2xy(2,el2no(1,i));  ybe=no2xy(2,el2no(3,i));
    % basis functions
    bas1=(ybe-midpt(i,2))/(ybe-yae);
    bas3=-(midpt(i,2)-yae)/(ybe-yae);
    bas2=(midpt(i,1)-xae)/(xbe-xae);
    bas4=-(xbe-midpt(i,1))/(xbe-xae);

    %construct the numerical E fields
    Ex_num(i)=edori(i,1)*solvec(el2ed(1,i))*bas1 + \ldots
              edori(i,3)*solvec(el2ed(3,i))*bas3;
    Ey_num(i)=edori(i,2)*solvec(el2ed(2,i))*bas2 + \ldots
              edori(i,4)*solvec(el2ed(4,i))*bas4;
 end
```

Considering that H is a piecewise constant, the numerical magnetic field H can be directly obtained by the following code.

```
for i=1:numel
    HH_num(i) = znew(iecnt+i);
end
```

Once we have the numerical solutions, we can postprocess the solutions in various ways. For example, we can plot the electric field $\mathbf{E}$ by simple commands as follows:

```
figure(1);clf(1);
quiver(midpt(:,1)',midpt(:,2)',Ex_num,Ey_num),
titstr=strcat('Numerical E field at midpoints');
title(titstr),
axis([lowx highx lowy highy]);
```

Similarly, we can do a surface plot for the scale magnetic field H as shown below:

```
figure(4);clf(4);
for j=1:neley
    for i=1:nelex
        % change 1-D vector into 2-D array
        U2d(i,j)=HH_num(nelex*(j-1)+i);
    end
end

surf(1:nelex, 1:neley, U2d');
titstr=strcat('Numerical H field');
title(titstr);
xlabel('X'); ylabel('Y');
```

A sample MATLAB code *postprocessing.m* demonstrating our postprocessing implementation is given below:

```
function postprocessing(HH,Ex,Ey,znew,tt,numel)

globals2D;

%plot the numerical field
solvec = zeros(numed,1);
solvec(eint)=znew(1:iecnt);   %coefficients of E field
 for i=1:numel
    % coordinates of this element from 1st node & 3rd node
    xae=no2xy(1,el2no(1,i));  xbe=no2xy(1,el2no(3,i));
    yae=no2xy(2,el2no(1,i));  ybe=no2xy(2,el2no(3,i));
    % basis functions: cf p111
    bas1=(ybe-midpt(i,2))/(ybe-yae);
    bas3=-(midpt(i,2)-yae)/(ybe-yae);
    bas2=(midpt(i,1)-xae)/(xbe-xae);
```

```
    bas4=-(xbe-midpt(i,1))/(xbe-xae);

    %calculate the numerical and exact E fields
    Ex_num(i)=edori(i,1)*solvec(el2ed(1,i))*bas1 + ...
              edori(i,3)*solvec(el2ed(3,i))*bas3;
    Ey_num(i)=edori(i,2)*solvec(el2ed(2,i))*bas2 + ...
              edori(i,4)*solvec(el2ed(4,i))*bas4;

    Ex_ex(i) = Ex(midpt(i,1),midpt(i,2),tt);
    Ey_ex(i) = Ey(midpt(i,1),midpt(i,2),tt);

    % calculate the numerical and exact H fields (a scalar)
    HH_ex(i) = HH(midpt(i,1),midpt(i,2),tt);
    HH_num(i) = znew(iecnt+i);
 end

 figure(1);clf(1);
 quiver(midpt(:,1)',midpt(:,2)',Ex_num,Ey_num),
 titstr=strcat('Numerical E field at midpoints');
 title(titstr),
 axis([lowx highx lowy highy]);

figure(2);clf(2);
quiver(midpt(:,1)',midpt(:,2)',Ex_ex, Ey_ex),
titstr=strcat('Analytical E field at midpoints');
title(titstr),
axis([lowx highx lowy highy]);

% plot Hz at the last time step
      timestep=int2str(nt);
      figure(3);clf(3);
      pcolor((reshape(HH_num(1:numel),nelex,neley))');
      hold on;
   %%   line(boxlinex,boxliney,'Color','w');
   %   hold off
      shading flat;
    %  caxis([-1.0 1.0]);
    %%  axis([1 ie 1 je]);
      colorbar;
      axis image;
     % axis off;
      titstr=strcat('Numerical H field');
      title(titstr);
      xlabel('X'); ylabel('Y');

      figure(4);clf(4);
      for j=1:neley
        for i=1:nelex
          % change 1-D vector into 2-D array
          U2d(i,j)=HH_num(nelex*(j-1)+i);
          H2d(i,j)=HH_ex(nelex*(j-1)+i)-U2d(i,j);
          Ex2d(i,j)=Ex_ex(nelex*(j-1)+i)-Ex_num(nelex*(j-1)+i);
        end
      end
```

```
      surf(1:nelex, 1:neley, U2d');
      title(titstr);
      xlabel('X'); ylabel('Y');

      figure(5);clf(5);
      surf(1:nelex, 1:neley, H2d');
      title('H field pointwise error');
      xlabel('X'); ylabel('Y');

      figure(6);clf(6);
      surf(1:nelex, 1:neley, Ex2d');
      title('Electric component Ex pointwise error');
      xlabel('X'); ylabel('Y');

%Debug: check the last 4 element solutions
for i=numel-4:numel
    display(' H exact, numer ='),HH_ex(i),HH_num(i)
    display('Ex exact, numer ='),Ex_ex(i),Ex_num(i)
end

display('Number of interior edges, numel, DOF = '), ...
                size(eint),numel,size(znew)

% calculate the max pointwise error
err_Ex=max(abs(Ex_num-Ex_ex)),
err_Ey=max(abs(Ey_num-Ey_ex)),
err_H=max(abs(HH_num-HH_ex)),
```

7.6 Numerical Results

In this section, we use an example to demonstrate our implementation of the scheme (7.1)–(7.4). To check the convergence rate, we construct the following exact solutions for the 2-D transverse electrical model (assuming that $\Gamma_m = \Gamma_e, \omega_m = \omega_e$) on the domain $\Omega = (0, 1)^2$:

$$\mathbf{E} \equiv \begin{pmatrix} E_x \\ E_y \end{pmatrix} = \begin{pmatrix} \sin \pi y \\ \sin \pi x \end{pmatrix} e^{-\Gamma_e t},$$

$$H = \frac{1}{\pi}(\cos \pi x - \cos \pi y) e^{-\Gamma_e t} (\omega_e^2 t - \Gamma_e).$$

The corresponding electric and magnetic currents are

$$\mathbf{J} \equiv \begin{pmatrix} J_x \\ J_y \end{pmatrix} = \begin{pmatrix} \sin \pi y \\ \sin \pi x \end{pmatrix} \omega_e^2 t e^{-\Gamma_e t},$$

and

$$K = \frac{1}{\pi}(\cos \pi x - \cos \pi y)e^{-\Gamma_e t}\omega_e^2(\frac{1}{2}\omega_e^2 t^2 - \Gamma_e t),$$

respectively. The corresponding source term $\mathbf{f} \equiv 0$, while g is given by

$$g = \frac{1}{\pi}(\cos \pi x - \cos \pi y)e^{-\Gamma_e t}(-2\Gamma_e\omega_e^2 t + \Gamma_e^2 + \omega_e^2 + \pi^2 + \frac{1}{2}\omega_e^4 t^2).$$

Notice that our solution $\mathbf{E}$ satisfies the conditions

$$\mathbf{n} \times \mathbf{E} = \mathbf{0} \quad \text{on } \partial\Omega, \qquad \nabla \cdot \mathbf{E} = 0 \quad \text{in } \Omega.$$

Our complete codes for solving this problem are composed of five MATLAB functions:

1. *Drude_cn.m*: the driver function;
2. *globals2D.m*: define all the global variables and constants;
3. *create_mesh.m, form_mass_matrix.m, postprocessing.m*: the other supporting functions explained above.

In the driver function *Drude_cn.m*, we assign the time step size, the total number of time steps of the simulation, and define the exact solutions used to calculate the error estimates. Below is our implementation of *Drude_cn.m*:

```
%-------------------------------------------------------------
% Author: Prof. Jichun Li
%
% Solve Drude metamaterial model using Crank-Nicolson type
% mixed FEM by rectangular edge element.
%
% Drive function (this one): Drude_cn.m
% Other supporting functions:
%    1. globals2D.m: define global variables and constants
%    2. create_mesh.m:  generate rectangular mesh
%    3. form_mass_matrix.m: create the global mass matrix
%                      and prepare for time marching
%    4. postprocessing.m: compare numerical and analytical
%            solutions, calculate errors and do plottings
%-------------------------------------------------------------
clear all,
globals2D;   % all global variables and constants
format long;

%%%%%%%%%% set up the exact solutions  %%%%%%%%%%%%%
gama=1.0e0; wpem=1.0e0;
% exact electric field and electric polarization
Ex = @(x,y,t)sin(pi*y).*exp(-gama*t);
Ey = @(x,y,t)sin(pi*x).*exp(-gama*t);
Jx = @(x,y,t)sin(pi*y).*exp(-gama*t)*wpem^2*t;
Jy = @(x,y,t)sin(pi*x).*exp(-gama*t)*wpem^2*t;
% exact magnetic field and magnetic polarization
```

```
HH = @(x,y,t)(cos(pi*x)-cos(pi*y))...
       /pi.*exp(-gama*t)*(wpem^2*t-gama);
KK = @(x,y,t)(cos(pi*x)-cos(pi*y))/pi.*exp(-gama*t)...
       *wpem^2*(0.5*(wpem*t)^2-gama*t);

% exact RHS
f1RHS = @(x,y,t)0.0;
f2RHS = @(x,y,t)0.0;
gRHS  = @(x,y,t)(cos(pi*x)-cos(pi*y))/pi.*exp(-gama*t)...
        *(-2*gama*wpem^2*t + gama^2 + wpem^2 ...
        + pi^2 + 0.5*wpem^2*(wpem*t)^2);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
dt=1.e-8, nt=10;
id_bc = 1;    %Indicator: 1 for Dirichlet BC; 0 otherwise.

% create a rectangular mesh on [lowx,highx]x[lowy,highy]
create_mesh;
dim = iecnt + numel;   % total number of unknowns

% local mass matrix
Mref = (dx*dy/6)*[2 0 -1 0;0 2 0 -1;-1 0 2 0;0 -1 0 2];
Curl = [dx;dy;dx;dy];      % (1, curl N_i)

matM = sparse(numed,numed);
matBM = sparse(numed,numel);
area = zeros(numel,1);

% form matrix matM and matrix matBM
[rhsEF,rhsEE,rhsEJ,rhsEH,rhsHH,rhsHK,rhsHG,H0,K0] = ...
    form_mass_matrix(HH,KK,gRHS,f1RHS,f2RHS,Ex,Ey,Jx,Jy);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if id_bc == 1 % For Dirichlet BC, use internal edge values
    matM = matM(eint,eint);
    matBM=matBM(eint,:);
    rhsEF=rhsEF(eint);
    rhsEE=rhsEE(eint);
    rhsEJ=rhsEJ(eint);
    rhsEH=rhsEH(eint);
end

E0=inv(matM)*rhsEE;   % get initial values of E and J
J0=inv(matM)*rhsEJ;

cst1 = 1+(dt*wpem)^2/(2*(2+dt*gama));
cst2 = 0.5*dt; cst3 = 1-(dt*wpem)^2/(2*(2+dt*gama));
cst4 = 2*dt/(2+dt*gama);
cst5=(2-dt*gama)/(2+dt*gama);
cst6=dt*wpem^2/(2+dt*gama);

 rhs_glb=[cst3*rhsEE + cst2*rhsEH - cst4*rhsEJ + dt*rhsEF; ...
        cst3*rhsHH - cst2*matBM'*E0 - cst4*rhsHK + dt*rhsHG];

 % the global coefficient matrix (A -B; B' D)
```

```
 % Ae  - Bh = f_1
 % B'e + Dh = f_2
 % solve the system: h = D^{-1}(f_2-B'e);
 %                  e = (A+BD^{-1}B')^{-1}(f_1+BD^{-1}f_2)
 % coefficient matrix for unknown E is: A+B*inv(D)*B'

 %mat4E = cst1*matM + matBM*matBM'./area*(cst2*cst2)/cst3;
 for k=1:numel
    tmp(:,k) = (matBM(:,k)/area(k));
 end
 mat4E = cst1*matM + matBM*tmp'*(cst2*cst2)/cst3;

% rhs4E = rhs_glb(1:iecnt) ...
%         + matBM./area*rhs_glb(iecnt+1:dim)*cst2/cst3;
 rhs4E = rhs_glb(1:iecnt) ...
          +tmp(:,1:numel)*rhs_glb(iecnt+1:dim)*cst2/cst3;

 %%%%%%%%%%%%%% begin time marching %%%%%%%%%%%%%%%%%%
 one = ones(1,4);
 tic   % start to measure the elapsed time
 for n=1:nt
   %first solve the system for E_h^k, then for H_h^k
   znew(1:iecnt,1) = mat4E\rhs4E;

   sum1 = zeros(numel,1);
   for ii=1:numel
       sum1(ii)=sum1(ii)+matBM(1:iecnt,ii)'*znew(1:iecnt);
   end

   znew(iecnt+1:dim,1) = (rhs_glb(iecnt+1:dim,1) ...
        -cst2*sum1(1:numel,1))./(cst3*area(1:numel,1));
   %znew(iecnt+1:dim)=(rhs_glb(iecnt+1:dim)...
   %                     -dt*matBM'*znew(1:iecnt))/a1;

   % for safety, re-assign to zero
   rhsEF=zeros(numed,1);          % for (f,N_i)
   rhsEH=zeros(numed,1);        % for (H0,curl N_i)
   rhsHH=zeros(numel,1);        % for (H0, psi_i)
   rhsHK=zeros(numel,1);        % for (K0, psi_i)
   rhsHG=zeros(numel,1);        % for (g, psi_i)

   if n <= (nt-1)
     % update all degrees of freedom
     tt = (n+0.5)*dt;
     En = znew(1:iecnt);
     Hn = znew(iecnt+1:dim);
     Jn = cst6*(En + E0) + cst5*J0;
     Kn = cst6*(Hn + H0) + cst5*K0;

     for i=1:numel
       xae=no2xy(1,el2no(1,i));  xbe=no2xy(1,el2no(3,i));
       yae=no2xy(2,el2no(1,i));  ybe=no2xy(2,el2no(3,i));

       % the coordinates of the four vertex
```

```
      xe(1)=xae; ye(1)=yae;
      xe(2)=xbe; ye(2)=yae;
      xe(3)=xbe; ye(3)=ybe;
      xe(4)=xae; ye(4)=ybe;

      for j=1:4   % loop through edges
       ed1 = el2ed(j,i);
       rhs_ef=0;

       for ii=1:2  % loop over gauss points in eta
          for jj=1:2   % loop over gauss points in psi
            eta = gauss(ii);  psi = gauss(jj);
            % Q1 function:
            NJ=0.25*(one + psi*psiJ).*(one + eta*etaJ);
            % derivatives of shape functions
            NJpsi=0.25*psiJ.*(one + eta*etaJ);  % 1x4 array
            NJeta=0.25*etaJ.*(one + psi*psiJ);  % 1x4 array
            % derivatives of x and y wrt psi and eta
            xpsi = NJpsi*xe';  ypsi = NJpsi*ye';
            xeta = NJeta*xe';  yeta = NJeta*ye';
          % Jinv = [yeta, -xeta; -ypsi, xpsi];  % 2x2 array
            jcob = xpsi*yeta - xeta*ypsi;

            xhat=dot(xe,NJ); yhat=dot(ye,NJ);

           if j==1
             bas1=(ybe-yhat)/(ybe-yae);
             rhs_ef=rhs_ef + f1RHS(xhat,yhat,tt)*bas1*jcob;
           elseif j==2
             bas2=(xhat-xae)/(xbe-xae);
             rhs_ef=rhs_ef + f2RHS(xhat,yhat,tt)*bas2*jcob;
           elseif j==3
             bas3=-(yhat-yae)/(ybe-yae);
             rhs_ef=rhs_ef + f1RHS(xhat,yhat,tt)*bas3*jcob;
           else
             bas4=-(xbe-xhat)/(xbe-xae);
             rhs_ef=rhs_ef + f2RHS(xhat,yhat,tt)*bas4*jcob;
           end
         end
       end
      % assemble the edge contribution into global rhs vector
      rhsEF(ed1)=rhsEF(ed1)+edori(i,j)*rhs_ef;
      rhsEH(ed1)=         edori(i,j)*Hn(i)*Curl(j);
    end   % end of 1st edge loop
    rhsHH(i)=Hn(i)*area(i);
    rhsHK(i)=Kn(i)*area(i);
    rhsHG(i)=gRHS(midpt(i,1),midpt(i,2),tt)*area(i);
  end  % end of element loop

     if id_bc == 1  % Dirichlet BC
       rhsEF=rhsEF(eint);
       rhsEH=rhsEH(eint);
     end
   % form new RHS
```

Table 7.1 The pointwise errors at element centers with $\Gamma_e = \omega_e = 1, \tau = 10^{-8}$ after 1 time step

Meshes	E_x errors	H_z errors
10×10	4.10388426568e−003	2.55501841905e−010
20×20	1.02758690447e−003	6.44169162455e−011
40×40	2.57051528841e−004	1.61380908636e−011
80×80	6.43804719980e−005	4.19719814459e−012
160×160	1.63183158637e−005	1.66422431391e−012

Table 7.2 The pointwise errors at element centers with $\Gamma_e = \omega_e = 1, \tau = 10^{-8}$ after 100 time step

Meshes	E_x errors	H_z errors	DOFs	CPU time (sec)
10×10	4.10387149e−03	2.55493626e−10	280	5.49
20×20	1.02756605e−03	6.44204689e−11	1,160	22.17
40×40	2.57014726e−04	1.61460844e−11	4,720	99.71
80×80	6.43118232e−05	4.11581879e−12	19,040	604.19
160×160	1.61859982e−05	1.33271171e−12	76,480	4479.43

```
rhs_glb=[cst3*matM*E0+cst2*rhsEH-cst4*matM*J0+dt*rhsEF;...
         cst3*rhsHH-cst2*matBM'*E0-cst4*rhsHK+dt*rhsHG];
rhs4E = rhs_glb(1:iecnt) ...
        + tmp(:,1:numel)*rhs_glb(iecnt+1:dim)*cst2/cst3;

   % update all dof
   E0 = En; J0 = Jn; H0 = Hn; K0 = Kn;
 end    % end of the BIG 'if' loop
 display('step n='),n
end    % end of time marching
toc    % end measurement of elapsed time
 %%%%%%%%%%%%%% end of time marching %%%%%%%%%%%%%%%
% compare the analytic and numerical solutions at T
postprocessing(HH,Ex,Ey,znew,nt*dt,numel);
return
```

Exemplary results are shown in Tables 7.1 and 7.2 (where DOFs denote the total number of degrees of freedom) and in Fig. 7.1. Tables 7.1 and 7.2 clearly show the pointwise convergence rate $O(h^2)$ at element centers, where h is the mesh size. Note that $O(h^2)$ is better than the theoretical approximation result, i.e., superconvergence happens at the rectangular element centers as we proved in Chap. 5.

7.7 Bibliographical Remarks

The number of books covering finite element programming for Maxwell's equations is quite limited. For example, the classic books by Jin [162] and by Silverster and Ferrari [267] describe the basic finite element techniques for Maxwell's equations. [267] even provides all the source code in Fortran. The recent book by Hesthaven and Warburton [141] introduced the nodal discontinuous Galerkin (DG) method for

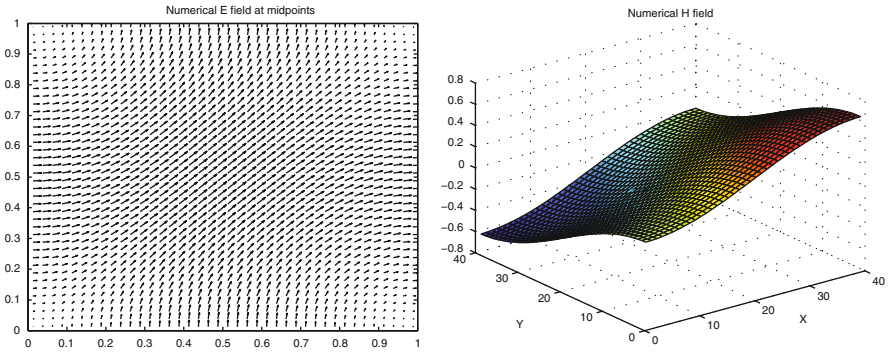

Fig. 7.1 Numerical solution obtained on 20×20 mesh with $\Gamma_e = \omega_e = 1, \tau = 10^{-10}$ after 100 time steps: (*Left*) The electric field; (*Right*) The magnetic field

conservation laws and Maxwell's equations. This book provides a very nice package for readers to experience the DG method for solving various problems, including time-domain Maxwell's equations in free space. Other recent contributions to this area are the books by Demkowicz et al. [97, 98], in which they detailed the hp-finite element method for solving elliptic problems and time-harmonic Maxwell's equations. A self-contained 2-D package (covering grid generation, solver, and visualisation) is included in [97]. Readers can find a few other hp Maxwell packages mentioned in the Foreword of [97].