MEEP C++

Scattering by Cylinders Using MEEP

Technical Report # 51

J. Richie June 6, 2013

Summary

In this work, software that simulates the scattering by a cylinder (two-dimensional case) is investigated using the MIT program MEEP. MEEP is a finite difference-time domain numerical method applied to Maxwell's equations. The algorithm typically uses a "marching-in-time" approach; however, direct solvers, particularly in the frequency domain, can be implemented as well.

The MEEP code can be executed by using a script written in the `libctl` language [1]. `libctl` is a variation of the `Scheme` language. A `python` interface is also available, but has not been investigated. The `libctl` interface has been used and has been found to be convenient when creating video clips of the fields in the problem. Video clips are created by taking snapshots (as GIF files), and using `gifsicle` [2] to concatenate the GIF files into an animation.

The MEEP code also allows for a native C++ interface as well. In this document, the C++ interface is described as part of the purpose for the work. The C++ interface is most convenient because all of the field quantities are readily available within the structure of a C++ program.

The problems solved here all involve scattering from a cylinder. The cases of a perfectly conducting cylinder and a dielectric cylinder are described. The incident fields considered are a plane wave and a monopole line source in the vicinity of the cylinder. In all cases, the electric field is parallel to the axis of the cylinder (the z axis) and hence, TM^z fields are present.

The scattered field is found for plane wave cases by simulating both the problem (i.e., the incident field in the presence of the cylinder) and the incident field with no scatterer present. The incident field is subtracted from the total field during simulation to obtain the scattered field. For monopole line source incident fields, the total field is reported. No incident field only simulation is performed.

The results have been disappointing. No suitable convergence toward the exact far field pattern has been observed at this time. Likely sources of error have been identified but have not been correctable at the present time.

[1] http://ab-initio.mit.edu/wiki/index.php/Libctl

[2] http://www.lcdf.org/gifsicle/

1. Introduction.

This document describes programming MIT MEEP [1]. MIT MEEP is a finite difference-time domain (FD-TD) code that can be executed using scripts (in the `Scheme` language via `libctl`) or as a library that can be called using a C++ interface. Here, the C++ interface is used. The MEEP code is also documented online[3].

The FD-TD method is applied to Maxwell's equations by using a difference approximation for the derivatives of the curl equations. A "marching-in-time" procedure is used where the fields are computed over time. The electric field and the magnetic field are one half time step apart. Thus, the electric field is updated using the electric field values from one time step prior and the magnetic field values from one half time step prior to the present time.

The method has been extensively studied. See [2] for a graduate text introduction to the subject, and [3] for a comprehensive review of the details of the method. The technique has been used for modeling microstrip, antennas [4], and a variety of other electromagnetic scenarios.

A one-dimensional version used in the MU undergraduate Fields II class as a demonstration of waves on a terminated transmission also appears in [5]. In the 1–D case, $\vec{E}$ can be mapped to V and $\vec{H}$ can be mapped to I.

In the case of antenna or scattering problems, the region of interest is typically infinite (or at least extends to the far field). In these situations, there are two issues. The first is that the space can not be simulated to the far field. Therefore, a transformation must be applied on the near fields to infer the far field quantities of interest. Second, the region of interest must be bounded in such a fashion that there is no reflection off the bounding surfaces. In MEEP, a perfectly matched layer (PML) is used to truncate the solution space without causing reflections.

Once a solution can be found that also does not allow for reflections off the solution space boundary, a way to find the far field pattern is needed. Here, the equivalence principle is used. The equivalence principle states that the tangential fields along some fictitious boundary can be used as equivalent sources. This fictitious boundary must enclose the actual sources. Then, the equivalent sources can be "radiated" to the far field using well-known free-space integral relationships.

[3] http://ab-initio.mit.edu/wiki/index.php/Meep

2. Plane Wave Geometry.

The geometry for the plane wave incident field has been set up using a number of parameters that make the geometry adjustable to investigate the effect of geometrical parameters (besides the cylinder radius). The geometry is shown in Fig. 1.

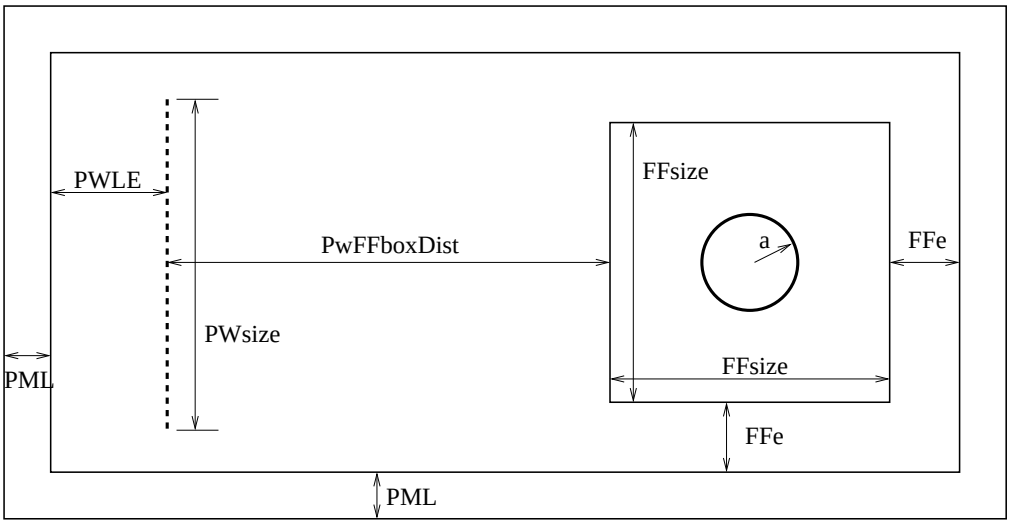

Figure 1. Geometrical parameters for the plane wave incident field problem.

In Fig. 1, the cylinder is shown with radius a. The surface used by the equivalence principle (denoted the far-field box) is a square of side `FFsize` centered about the cylinder. The edge of the solution space is padded with a region for the perfectly matched layer (`PML`). A region between the far-field box and the `PML` layer has thickness `FFe`. The distance between the far-field box and the plane wave source is `PwFFboxDist`. The padding between the plane wave source and the left edge of the model has thickness `PWLE`. The parameter `PWsize` defines the extent of the plane wave source as a percentage of the full distance. The full distance is the model size less the two `PML` regions. That is, if `PWsize` = 1, then the plane wave extends vertically from `PML` layer to `PML` layer.

3. Monopole Geometry.

The geometry for the monopole line source incident field is shown in Fig. 2. Several of the parameters are the same as the plane wave case. The cylinder radius, the PML layer, the size of the far field box (FFsize), and the far field box buffer (FFe) are all the same.

For the monopole location, a distance, SrcLoc is used. This is the distance from the edge of the cylinder to the line source. The structure consisting of the cylinder and the line source is centered in the far field box. The relation, $SrcLoc + 2a < FFbox$ must be true; the code will abort if FFbox is too small.

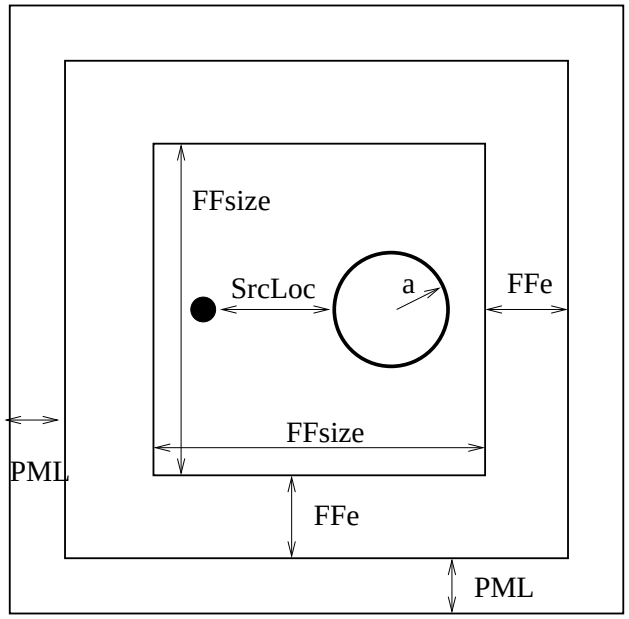

Figure 2. Geometrical parameters for the monopole line source incident field problem.

4. Software.

In this section, the software that is used to simulate the cylinder scattering problems is described. The C++ interface to MEEP is documented insofar as it was used in this work. For reference, the source file, `meep.hpp` is a header file that includes prototypes for all the public class members that can be used to write the code.

The software is written in a sequence of source code files in the subdirectory `src`. The cweb documentation file is in the top directory with figure files in the `fig` subdirectory. There is a makefile in the top level directory and a makefile for the source code in the `src` subdirectory. The top-level makefile calls the `src` makefile to compile the program. The final executable and support scripts/files can be "moved" to a simulation directory (/sims/meepCpp) as well. There is a `results` subdirectory that holds the result data used in this document. Full results are in the `/sims/meepCpp` directory. Finally, there is a `nf2ff` subdirectory that holds the code that computes the near field to far field transformation applied to the analytic near field solution.

The main entry point for the code is defined in the `meepCyl.cc` file. This file declares prototype functions, global variables, and variable types. The global information is repeated (as external) in the header file `meepCyl.h`.

The first function call from `main` is to `input`. The purpose of `input` is to collect the geometrical parameters and problem variables from the user. This is typically done using an input pipe with a file that lists all the input data.

The next function call from `main` is to `ffbox`. The purpose of this function is to use the input parameters to set up the far-field box for the calculations. The far-field box parameters are conveniently stored in the `FFbox` structure. The parameters included are the (x, y) location of each point and the multiplication factor for each point (see the equivalence principle section). An important parameter that is determined in this step is the number of points used in the far-field (equivalence principle) calculation.

The `runMEEP` function call then sets up the solution space and executes the simulation. The software requires that functions be written that can be called to return the value of material constants for any point in the solution space. The function `eps` is used to define the cylinder material. A second function, `air` is used to define the material parameters (free space) for the incident field simulation.

For plane wave simulations, both an incident field simulation and an object-present simulation are performed in parallel. After enough timesteps, the code collects time-domain data at each of the points defined in the `FFbox` structure. This data is stored in a local array of values. For monopole simulations, the total far field pattern is of interest; therefore, only a total field simulation is executed and no subtraction is performed.

Once the simulation is complete, the data collected toward the end of the run is analyzed. The MEEP function `do_harminv` is used to convert each time-series to a phasor magnitude and phase. The equivalent electric and magnetic currents are stored in the FFBox structure.

Finally, the `ffCalc` function is called to perform the far-field integration. this function implements the equivalence principle to compute the (normalized, linear) far field pattern for the field. Plane wave simulations result in the scattered far field; monopole simulations result in the total far field.

5. Makefile.

Listed here is the main makefile for the project. This makefile is used directly to create the documentation for the project. The makefile also calls the `src` makefile (listed in the next section) to compile the executable file.

```
#
#   master makefile for MEEP w/ C++ interface
#
all:                meepCylMake meep-Cpp.dvi

move:                meepCylMake
                cp src/meepCyl ~/sims/meepCpp; \
                cd scripts; cp * ~/sims/meepCpp
meepCylMake:
                cd src;make meepCyl

meep-Cpp.dvi:        meepCyl.w pwGeo.eps boxFlds.eps monoGeo.eps PEC-rmse.eps \
                FFsizeVarFF.eps PW-EPS-ff.eps nf2ffData.eps Pml3.0.eps
                cweave meepCyl.w; \
                tex meepCyl.tex; bibtex meepCyl ; tex meepCyl.tex; tex meepCyl.tex
pwGeo.eps:        fig/pwGeo.fig
                fig2dev -L eps fig/pwGeo.fig ./pwGeo.eps
boxFlds.eps:        fig/boxFlds.fig
                fig2dev -L eps fig/boxFlds.fig ./boxFlds.eps
monoGeo.eps:        fig/monoGeo.fig
                fig2dev -L eps fig/monoGeo.fig ./monoGeo.eps
PEC-rmse.eps:        results/PEC-rmse.plt results/FFsize.data
                cd results; gnuplot PEC-rmse.plt; cd ..
FFsizeVarFF.eps:        results/FFsizeVarFF.plt results/FFsize2.5.sum \
                        results/FFsize5.7.sum
                cd results; gnuplot FFsizeVarFF.plt; cd ..
PW-EPS-ff.eps:        results/PW-EPS-ff.plt results/PwFFboxDist2.0.sum
                cd results; gnuplot PW-EPS-ff.plt; cd ..
nf2ffData.eps:        nf2ff/three.dat nf2ff/nf2ffData.plt
                cd nf2ff; gnuplot nf2ffData.plt; cd ..
nf2ff/three.dat:
                cd nf2ff; nf2ffData.sh; cd ..
Pml3.0.eps:        results/Pml3.0.plt results/Pml3.0.sum
                cd results; gnuplot Pml3.0.plt; cd ..
srcClean:
                cd src; make clean; cd ..

clean:        srcClean
        rm -f *~ *.dvi *.log *.idx *.scn *.tex *.toc meepCyl \
        *.aux *.eps *.h5 *.bbl *.blg fig/*.bak scripts/*~
```

6. Makefile – src.

Here is makefile for source code.

```
MEEPLIBS= -lmeep -lhdf5 -lz -lharminv -lblas -lfftw3 -llapack
LIBS= -lcpp -lm
LOCAL=$(HOME)/bin/local
INCLDIR=$(LOCAL)/include
LIBDIR=$(LOCAL)/lib
CC = g++ -malign-double -Wall
OBJ= meepCyl.o input.o ffbox.o run.o ffcalc.o

all:                meepCyl

meepCyl:        $(OBJ)
                $(CC) $(OBJ) -L$(LIBDIR) -I$(INCLDIR) -o meepCyl $(MEEPLIBS) $(LIBS);

meepCyl.o:        meepCyl.cc
                $(CC) meepCyl.cc -o meepCyl.o -I$(INCLDIR) -c

input.o:        input.cc
                $(CC) input.cc -o input.o -I$(INCLDIR) -c

ffbox.o:        ffbox.cc
                $(CC) ffbox.cc -o ffbox.o $(LIBS) -c

run.o:                run.cc
                $(CC) run.cc -o run.o $(MEEPLIBS) $(LIBS) -c

ffcalc.o:        ffcalc.cc
                $(CC) ffcalc.cc -o ffcalc.o $(LIBS) -c

clean:
                rm -f *.h5 *.png meepCyl *~ *.o
```

7. Source Code.

Here is the source code. The software is modularized to make the code more readable and changeable, witnessed by the development of the code. The plane wave, perfectly conducting cylinder case was created first, subsequent problems (such as the dielectric cylinder and the monopole line source incident field) were added to the code later.

8. Header File: `meepCyl.h`.

The header file, `meepCyl.h` contains the declaration of the global variables (as external) and the definition of the `FFbox` structure that is used to store the information on the box used to compute the far field.

```
#include <math.h>
#include <complex>
  ;
  ;    /*  global variables */
  ;
  extern int NumPts, NumTimeSeries, NumTimeSteps;
  extern double CylRad, epsR, epsRel;
  extern double CylCenterX, CylCenterY;
  extern double resolution, delta, courant;
  extern double FFsize, Pml, FFe, PwFFboxDist, PWle, PWsize, SrcLoc;
  extern double FF[361];
  ;
  ;    /* PWflag is 1 if plane wave, 0 if monopole */
  ;
  extern int PWflag;
  ;
  ;    /* definition of FFbox structure */
  ; struct FFbox { double x;
  double y;
  double f; std::complex < double > Jeq, Meq; } ;
  extern struct FFbox *ffBox;
```

9. Main Entry: meepCyl.cc.

This is the main entry point for the code. All this portion does is set up the global variables, and then calls a set of functions that solves the problem. The functions are:

- input
- ffbox
- runMEEP
- ffCalc.

```
#include <iostream>
#include <math.h>
#include <complex>
#include "std-def.h"
  ;
  ;    /* global variables */
  ;
  int NumPts, NumTimeSeries, NumTimeSteps;
  double CylRad, epsR, epsRel;
  double CylCenterX, CylCenterY;
  double resolution, delta, courant;
  double FFsize, Pml, FFe, PwFFboxDist, PWle, PWsize, SrcLoc;
  double FF[361];
  ;
  ;    /* PWflag is 1 if plane wave, 0 if monopole */
  ;
  int PWflag;
  ;
  ;    /* definition of FFbox structure */
  ; struct FFbox { double x;
  double y;
  double f; std::complex < double > Jeq, Meq; } ;
  ;
  ;    /* definition of FFbox structure */
  ;
  struct FFbox *ffBox;
  ;
  ;    /* other declarations (prototypes) */
  ;
  void input(void);
  void ffbox(void);
  void runMEEP(int argc, char **argv);
  void ffCalc(void);
  ;
  ;    /* Here is main entry point for code */
  ;
  int main(int argc, char **argv)
  {
```

```
  int Iphi;
  ;
  ;    /* splash shot for beginning */
  ;
  COUT << "\n\n+-----------------------------------------------------+\n";
  COUT << "|                                                       |\n";
  COUT << "|     MEEP: 2-D circular cylinder scatterers            |\n";
  COUT << "|                                                       |\n";
  COUT << "+-----------------------------------------------------+\n\n\n";
  input();
  ;
  ffbox();
  ;
  COUT << "\n  Number of points along far field box is " << NumPts << "\n";
  COUT << "\n\n-----------------------------------------------------\n\n";
  COUT << "                              MEEP is running\n\n";
  ;
  runMEEP(argc, argv);
  ;
  COUT << "\n\n  Number of points in time series (for harminv) is " << NumTimeSeries <<
      "\n\n";
  ;
  ffCalc();
  ;
  COUT << "\n\n------------  Far Field Pattern  -------------------\n\n";
  for (Iphi = 0; Iphi < 361; Iphi++) COUT << "  FFP: " << Iphi << "  " << FF[Iphi] << "\n";
  ;
  return 0;
}
```

10. Input Function: `input.cc`.

This function is used to set up the geometry of the problem and set the related global variables. Input is facilitated using the `lcpp` library [6]. The header file `input.h` (next section) is also needed to declare string constants.

```
#include <iostream>
#include <stdlib.h>
#include "meepCyl.h"
#include "cpp-fcns.h"
#include "std-def.h"
#include "input.h"
  void input(void)
  {
    ;
    ;     /*  input parameters */
    ;
    COUT << "\n\n␣---------------␣Problem␣Parameters␣-----------------␣\n\n";
    epsR = getinputd(s0);
    PWflag = getinputi(sA);
    CylRad = getinputd(s1);
    COUT << "\n\n␣-------------␣Geometrical␣Parameters␣---------------␣\n\n";
    FFsize = getinputd(s2);
    Pml = getinputd(s3);
    FFe = getinputd(s4);
    if (PWflag ≡ 1) {
      PwFFboxDist = getinputd(s5);
      PWsize = getinputd(s7);
      PWle = getinputd(s6);
    }
    if (PWflag ≡ 0) SrcLoc = getinputd(sH);
    ;
    ;     /* pixels per distance */
    ;
    COUT << "\n\n␣-------------␣Miscellaneous␣Parameters␣-------------␣\n\n";
    resolution = getinputd(s8);
    NumTimeSteps = getinputi(s9);
    delta = getinputd(s10);
    courant = getinputd(s11);
    ;
    ;     /* derived parameters */
    ;
    if (PWflag ≡ 1) {
      CylCenterX = Pml + PWle + PwFFboxDist + (0.5 * FFsize);
      CylCenterY = Pml + FFe + (0.5 * FFsize);
    }
    if (PWflag ≡ 0) {
      CylCenterX = Pml + FFe + 0.5 * (FFsize + SrcLoc);
      CylCenterY = Pml + FFe + 0.5 * FFsize;
      if ((2 * CylRad + SrcLoc) > FFsize) {
        COUT << "FFbox␣too␣small\n";
```

```
        exit(1);
      }
    }
  }
```

11. Header file for input: `input.h`.

The header file for `input.cc` contains the string definitions for the input functions.

```
char s0[] = "␣␣Enter␣eps_r␣(negative␣for␣metal)";
char sA[] = "␣␣Enter␣1␣for␣pw,␣0␣for␣monopole";
char s1[] = "␣␣Enter␣the␣cylinder␣radius␣(wavelengths)";
char s2[] = "␣␣Enter␣the␣size␣of␣the␣far␣field␣box␣(w/l)";
char s3[] = "␣␣Enter␣the␣PML␣thickness␣(w/l)";
char s4[] = "␣␣Enter␣the␣far␣field␣buffer␣size␣(FFe,␣w/l)";
char s5[] = "␣␣Enter␣the␣distance,␣PW␣->␣FFbox␣(w/l)";
char s6[] = "␣␣Enter␣left␣side␣PW␣buffer␣dist.␣(PWle,w/l)";
char s7[] = "␣␣Enter␣the␣PWsize␣(0->1)";
char sH[] = "␣␣Enter␣distance␣cyl␣to␣monopole";
char s8[] = "␣␣Enter␣the␣FD-TD␣resolution␣(cells␣per␣w/l)";
char s9[] = "␣␣Enter␣the␣number␣of␣time␣steps";
char s10[] = "␣␣Enter␣delta␣for␣far␣field␣box␣points(w/l)";
char s11[] = "␣␣Enter␣the␣Courant␣number␣(0-1)";
```

12. Far Field Box: `ffbox.cc`.

This function is used to determine and store the points used in the equivalence principle. The points are along the far-field box as shown in Fig. 1. From the input function, the size of the box `FFsize` and the spacing between points `delta` are known.

The location of the four corners of the box are computed. The number of points is counted using four (one for each side) for loops. The number of points is saved as `NumPts`, a global variable. Memory is allocated for the `ffBox` structure. Then, the for loops are executed again to fill in `ffBox` with x, y locations and a factor f that is used in the far field calculation.

```
#include "meepCyl.h"
#include <math.h>
#include <complex>
  void ffbox(void)
  {
    int count;
    double xL, xH, yL, yH, x, y;
    ;
    ;     /*find corners of the box and count number of points to use */
    ;
    xL = (CylCenterX - 0.5 * FFsize);
    xH = (CylCenterX + 0.5 * FFsize);
    yL = (CylCenterY - 0.5 * FFsize);
    yH = (CylCenterY + 0.5 * FFsize);
    ;
    count = 0;
    ;
    for (x = xL; x < xH + delta/2.; x = x + delta) count++;
    for (y = yL; y < yH + delta/2.; y = y + delta) count++;
    for (x = xH; x > xL - delta/2.; x = x - delta) count++;
    for (y = yH; y > yL - delta/2.; y = y - delta) count++;
    ;
    ;     /* Set number of points and allocate memory for ffBox */
    ;
    NumPts = count;
    ffBox = new struct FFbox[NumPts];
    ;
    ;     /* fill in FFbox->x,y,f */
    ;
    count = 0;
    y = yL;
    for (x = xL; x < xH + delta/2.; x = x + delta) {
      (ffBox + count)->x = x;
      (ffBox + count)->y = y;
      (ffBox + count)->f = 1.0;
      if ((x < xL + delta * 0.7) ∨ (x > xH - delta * 0.7)) (ffBox + count)->f = 0.5;
      count++;
    }
    x = xH;
    for (y = yL; y < yH + delta/2.; y = y + delta) {
```

```
    (ffBox + count)→x = x;
    (ffBox + count)→y = y;
    (ffBox + count)→f = 1.0;
    if ((y < yL + 0.7 * delta) ∨ (y > yH − delta * 0.7)) (ffBox + count)→f = 0.5;
    count++;
  }
  y = yH;
  for (x = xH; x > xL − delta/2.; x = x − delta) {
    (ffBox + count)→x = x;
    (ffBox + count)→y = y;
    (ffBox + count)→f = 1.0;
    if ((x > xH − delta * 0.7) ∨ (x < xL + delta * 0.7)) (ffBox + count)→f = 0.5;
    count++;
  }
  x = xL;
  for (y = yH; y > yL − delta/2.; y = y − delta) {
    (ffBox + count)→x = x;
    (ffBox + count)→y = y;
    (ffBox + count)→f = 1.0;
    if ((y > yH − delta * 0.7) ∨ (y < yL + delta * 0.7)) (ffBox + count)→f = 0.5;
    count++;
  }
}
```

13. runMEEP: `run.cc`.

This is the function that executes the MEEP code and also does the post-processing to obtain the equivalent electric and magnetic currents used in the equivalence principle around the far field box. Upon exit, the Jeq and Meq values for the `ffBox` structure have been computed and stored.

To compile the software including the MEEP code, one must `#include meep.hpp`, declare ‘‘`using namespace meep`’’ and also initialize the solver by executing `initializempi(argc,argv)` (this appears later in the `runMEEP` function).

The units for the solution space have been set so that one unit (or “block”) is one wavelength. In MEEP, all constants such as c, ϵ_o, and μ_o are one. Thus, the impedance, or ratio of electric field to magnetic field is also one.

The MEEP code requires that functions be used to return the material properties given a vector at any (non-PML) location in the solution space. The first of these is `eps`, which defines the cylinder. Note that the center of the cylinder is already known and can be used in `eps`. Also, metal is a special case of dielectric constant in MEEP. For metal, one uses $\epsilon_r = -\infty$. A second function is also used in the plane wave case because the incident field is also needed. The function `air` is used to simulate the incident field without the cylinder.

First, the grid volume is declared and then two structures are defined, one for the cylinder problem, and one for the incident field problem (if necessary). Two instances of the fields class are constructed, one for each problem.

The incident field is created at a frequency of one since $f\lambda = c$. The width of the source is set at twice the MEEP resolution. The width is the time for the source to ramp up to the final amplitude. This is done to avoid any dispersion formed by a fast turn on of the source.

The simulation is run for 201 time units. One time unit is actually a number of time steps; the number of time steps in one time unit is twice the declared resolution. After the 201 time units, the simulation is stepped further; however, at each time step, the amplitude of the electric and magnetic fields along the far-field box are stored for later processing. This is accomplished using `get_field`. To avoid the small phase difference between $\vec{E}$ and $\vec{H}$ (due to the half time-step difference), the fields are synchronized before they are stored. The fields are then restored to their previous state before the next time step is computed.

Once the time-stepping has ended, the data collected along the far-field box is processed. Each set of data is a time series at a single frequency. What is needed for the far field calculation is the amplitude and phase of each time series. Previously, this was crudely estimated using the maximum and minimum values along with the first upward zero crossing.

The function, `do_harminv` is ideally suited to extract the amplitude and phase of each time series. `do_harminv` reads in the time series and finds the complex frequencies within a given band. In the time series, the center frequency should be the inverse of twice the resolution and only one component should be present. The complex amplitudes are determined and stored in the `ffBox` structure.

```
  ;
  ;     /* We assume one block is one wavelength in this simulation */
  ;
  ;
#include <meep.hpp>
  using namespace meep;
#include "meepCyl.h"
```

```
#include <stdlib.h>
  ;
  ;    /* function to define the dielectric/air regions one for cyl (eps) and one for
       noCyl (air) */
  ;
  double eps(const vec&p)
  {
    double xc = p.x( ) − CylCenterX, yc = p.y( ) − CylCenterY;
    if (sqrt(xc ∗ xc + yc ∗ yc) < CylRad) return epsRel;
    else return 1.0;
  }
  ;
  double SigmaCyl(const vec&p)
  {
    double xc = p.x( ) − CylCenterX, yc = p.y( ) − CylCenterY;
    if (sqrt(xc ∗ xc + yc ∗ yc) < CylRad) return 0.;
    else return 0.1;
  }
  ;
  double air(const vec&p)
  {
    return 1.0;
  }
  ;
  double SigmaAir(const vec&p)
  {
    return 0.1;
  }
  ;
  ; void runMEEP(int argc, char ∗∗argv){ int count;
      double sx, sy;
      double rtmpx1, rtmpy1, rtmpy2; complex < double > freq;
      double width, x, y;
      double fLo, fHi;
      int i;
      int num, j; complex < double > fEz[NumPts][(int) resolution ∗ 2], fHp[NumPts][(int)
          resolution ∗ 2]; complex < double > ezTmp[(int) resolution ∗ 2], hpTmp[(int) resolution ∗ 2];
      initializempi(argc, argv);      /* do this even for non-MPI Meep */
      ;
      ;      /* set relative epsilon value */
      ;
      if (epsR < 0) epsRel = −1. ∗ infinity;
      if (epsR > 0) epsRel = epsR;
      ;
      ;      /* set problem geometry size */
      ;
      if (PWflag ≡ 1) {
        sx = 2. ∗ Pml + PWle + PwFFboxDist + FFe + FFsize;
        sy = FFsize + 2. ∗ (Pml + FFe);
```

```
}
if (PWflag ≡ 0) {
  sx = 2. * Pml + 2. * FFe + FFsize;
  sy = 2. * Pml + 2. * FFe + FFsize;
}
;
;
  /* set up space and field array for problem with cylinder and without cylinder */
;
grid_volume v = vol2d(sx, sy, resolution);
structure s(v, eps, pml(Pml), meep::identity(), 0, courant);
  /* s.set_conductivity(Dz, SigmaCyl); */
fields f(&s);      /* f.use_real_fields(); */
;
;
structure s0(v, air, pml(Pml), meep::identity(), 0, courant);
  /* s0.set_conductivity(Dz, SigmaAir); */
fields f0(&s0);      /* f0.use_real_fields(); */
;
;    /* print eps function for graphics */
;    /* f.output_hdf5(Dielectric, v.surroundings()); */
;
;    /* set up the incident field */
;
freq = 1.;
width = 2. * resolution;
continuous_src_time src(freq, width);
;
;    /* set up plane wave source */
;
if (PWflag ≡ 1) {
  rtmpx1 = Pml + PWle;      /* x location */
  rtmpy1 = sy - Pml - PWsize * (2. * FFe + FFsize);      /* y bottom */
  rtmpy2 = Pml + PWsize * (2. * FFe + FFsize);      /* y top */
  ;
  ;     /* traditional volume source */
  ;
  volume src_volume(vec(rtmpx1, rtmpy1), vec(rtmpx1, rtmpy2));
  f.add_volume_source(Ez, src, src_volume);
  f0.add_volume_source(Ez, src, src_volume);
}
if (PWflag ≡ 0) {
  rtmpx1 = CylCenterX - CylRad - SrcLoc;
  rtmpy1 = sy/2.;
  f.add_point_source(Ez, src, vec(rtmpx1, rtmpy1));
}
;
;    /* implement plane wave as set of point sources ; dy = (rtmpy2 - rtmpy1)/2000.;
    for (y = rtmpy1; y ≤ rtmpy2; y += dy) { f.add_point_source(Ez, src, vec(rtmpx1, y));
    f0.add_point_source(Ez, src, vec(rtmpx1, y)); } */
;
;    /* Now, run the code and collect scattered data */
```

```
;
while (f.time( ) < NumTimeSteps) {
  f.step( );
  if (PWflag ≡ 1) f0.step( );
}
f.output_hdf5(Ez, v.surroundings( ));
count = 0;
while (f.time( ) < NumTimeSteps + 1) {
  f.step( );
  if (PWflag ≡ 1) f0.step( );
  for (i = 0; i < NumPts; i++) {
    x = (ffBox + i)→x;
    y = (ffBox + i)→y;
    if (PWflag ≡ 1) fEz[i][count] = f.get_field(Ez, vec(x, y)) − f0.get_field(Ez, vec(x, y));
    if (PWflag ≡ 0) fEz[i][count] = f.get_field(Ez, vec(x, y));
    if ((i < NumPts/4) ∨ ((i ≥ 2 ∗ NumPts/4) ∧ (i < 3 ∗ NumPts/4))) {
      if (PWflag ≡ 1) {
        f.synchronize_magnetic_fields( );
        f0.synchronize_magnetic_fields( );
        fHp[i][count] = f.get_field(Hx, vec(x, y)) − f0.get_field(Hx, vec(x, y));
        f.restore_magnetic_fields( );
        f0.restore_magnetic_fields( );
      }
      if (PWflag ≡ 0) {
        f.synchronize_magnetic_fields( );
        fHp[i][count] = f.get_field(Hx, vec(x, y));
        f.restore_magnetic_fields( );
      }
    }
    else {
      if (PWflag ≡ 1) {
        f.synchronize_magnetic_fields( );
        f0.synchronize_magnetic_fields( );
        fHp[i][count] = f.get_field(Hy, vec(x, y)) − f0.get_field(Hy, vec(x, y));
        f.restore_magnetic_fields( );
        f0.restore_magnetic_fields( );
      }
      if (PWflag ≡ 0) {
        f.synchronize_magnetic_fields( );
        fHp[i][count] = f.get_field(Hy, vec(x, y));
        f.restore_magnetic_fields( );
      }
    }
  }
  count++;
}
;
;    /* set the number of time steps variable */
;
if (count ≠ (int) resolution ∗ 2) {
  master_printf("Number␣of␣points␣in␣time␣series␣incorrect\n");
  exit(1);
```

```
}
NumTimeSeries = count;
;
;     /* do the post processing with harminv */
;
int maxBands = 1; complex < double > *amps = new complex < double > [maxBands];
double *freq_re = new double[maxBands];
double *freq_im = new double[maxBands];
;
;
fLo = 0.75/(2. * resolution);
fHi = 1.25/(2. * resolution);
;
for (i = 0; i < NumPts; i++) {
  for (j = 0; j < NumTimeSeries; j++) {
    ezTmp[j] = fEz[i][j];
    hpTmp[j] = fHp[i][j];
      /* if (i ≡ 0) master_printf("fEz,fHp, %f %f\n", real(ezTmp[j]), real(hpTmp[j])); */
  }
  ;
  ;     /* determine Jeq + check for stray frequencies */
  ;
  num = do_harminv(ezTmp, NumTimeSeries, 1, fLo, fHi, maxBands, amps, freq_re, freq_im);
  (ffBox + i)->Jeq = amps[0];
  if (num ≠ 1) master_printf("issue in harminv at NumPts = %d\n", i);
  ;
  ;     /* determine Meq + check for stray frequencies */
  ;
  num = do_harminv(hpTmp, NumTimeSeries, 1, fLo, fHi, maxBands, amps, freq_re, freq_im);
  (ffBox + i)->Meq = amps[0];
  if (num ≠ 1) master_printf("issue in harminv at NumPts = %d\n", i);
}
}
```

14. Far Field Calculation: `ffcalc.c` (rnb:03/06/2013).

This function uses the equivalent currents from the MEEP code along with the locations and factor values (all stored in `struct ffBox`) to compute the linear far field pattern (normalized). The basis for this calculation is the equivalence principle. The equivalence principle states that the tangential fields along a closed surface can be used to find the fields outside that closed surface provided there are no sources outside.

Typically, a second step is to enforce either a perfect electric conducting or perfect magnetic conducting condition within the closed surface. This is done in diffraction where a metal screen with an aperture is present. However, in the case studied here, one can not enforce such a condition. Therefore, both the tangential electric and tangential magnetic fields are needed to find the far field pattern.

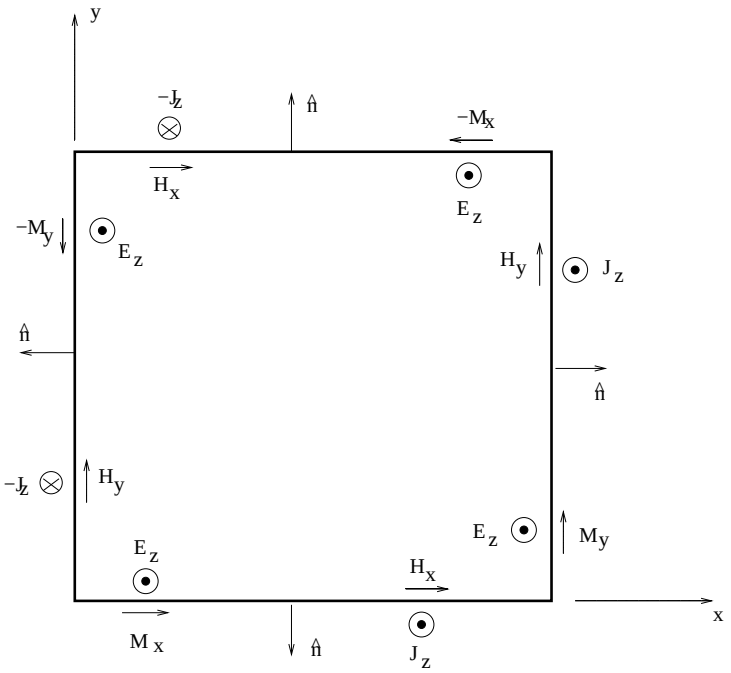

Figure 3. Geometry for Meq and Jeq around the far-field box.

Consider the electric field, $\vec{E} = E_z(x,y)\hat{e}_z$. The equivalent magnetic current is given by

$$\vec{M} = \vec{E} \times \hat{n} \tag{1}$$

where $\hat{n}$ is the unit normal vector, as shown in Fig. 3. The vector electric potential can be computed using the magnetic current. The result can be reduced to two dimensions by integrating over z':

$$\vec{F} = \frac{\epsilon}{4\pi} \int_v \vec{M}(\vec{r}\,') \frac{e^{-jk|\vec{r}-\vec{r}\,'|}}{|\,\vec{r}-\vec{r}\,'\,|} dv' \longrightarrow -j\frac{\epsilon}{4} \int_{x',y'} \vec{M}(\vec{\rho}\,') H_o^{(2)}(k\,|\,\vec{\rho}-\vec{\rho}\,'\,|) d\ell' \tag{2}$$

The far field relationship between $\vec{F}$ and $\vec{E}$ is given by:

$$(E_F)_\theta \rightarrow -j\omega\eta F_\phi$$

as $r \to \infty$. Since we need F_ϕ, we substitute and take the dot product with $\hat{e}_\phi$:

$$F_\phi(\vec{\rho}) = -j\frac{\epsilon}{4}\int\limits_{x',y'} [M_x(\vec{\rho}')\hat{e}_x + M_y(\vec{\rho}')\hat{e}_y] \cdot \hat{e}_\phi H_o^{(2)}(k \mid \vec{\rho} - \vec{\rho}' \mid)\ell' \tag{3}$$

Using the identities:

$$\begin{aligned} \hat{e}_x \cdot \hat{e}_\phi &= -\sin\phi \\ \hat{e}_y \cdot \hat{e}_\phi &= \cos\phi \end{aligned} \tag{4}$$

the following rigorous expression is obtained:

$$F_\phi(\vec{r}) = j\frac{\epsilon}{4}\int\limits_{x',y'} [M_x(\vec{\rho}')\sin\phi - M_y(\vec{\rho}')\cos\phi]\, H_o^{(2)}(k \mid \vec{\rho} - \vec{\rho}' \mid)d\ell' \tag{5}$$

To obtain the far field pattern, the ρ dependence is examined. As $\rho \to \infty$,

$$H_o^{(2)}(\mid \vec{\rho} - \vec{\rho}' \mid) \longrightarrow \sqrt{\frac{2}{\pi k\rho}}\; e^{-jk\rho} e^{jk\rho' \cos(\phi - \phi')} e^{j\pi/4} \tag{6}$$

Finally, after some algebraic manipulations, our final result is obtained:

$$E_z^M = -e^{j\pi/4}\sqrt{\frac{k}{8\pi\rho}}\; e^{-jk\rho}\int\limits_{x',y'} [M_x(\vec{\rho}')\sin\phi - M_y(\vec{\rho}')\cos\phi]\, e^{jk\rho' \cos(\phi - \phi')} d\ell' \tag{7}$$

where the superscript M is used to denote that the contribution to the electric field is due to the magnetic current.

A similar procedure can be used to obtain the contribution from the tangential magnetic field. The equivalent electric current, J, is given by:

$$J = \hat{n} \times \vec{H}$$

where, in the far field,

$$\vec{E} = -j\omega\vec{A}$$

The vector magnetic potential is found using

$$\vec{A}(\vec{r}) = -j\frac{\mu}{4\pi}\int\limits_{v'} J(\vec{r}')\frac{e^{-jk|\vec{r} - \vec{r}'|}}{\mid \vec{r} - \vec{r}' \mid} dv' \longrightarrow -j\frac{\mu}{4}\int\limits_{x',y'} J(\vec{\rho}')H_o^{(2)}(k \mid \vec{\rho} - \vec{\rho}' \mid)ds' \tag{8}$$

where the second expression is specialized to the two-dimensional case, as before. Using the same procedure, the far field expression for the electric field due to an electric current is obtained:

$$E_z^J = -\eta e^{j\pi/4}\sqrt{\frac{k}{8\pi\rho}}\; e^{-jk\rho}\int\limits_{x',y'} J_z(\vec{\rho}') e^{jk\rho' \cos(\phi - \phi')} d\ell' \tag{9}$$

where the superscript J implies that the electric field contribution is due to the equivalent electric current. The total far-field contribution is given by:

$$E_z^{FF} = E_z^J + E_z^M \tag{10}$$

The data that is collected from MEEP includes the equivalent electric current at every point on the far-field box. the equivalent magnetic current is stored as either M_x or M_y depending on which side of the box the point resides. Note that each corner of the box is used twice: once for the vertical side and once for the horizontal side. In addition, the "factor" element in structure `ffBox` is either 1 or -1, depending again on the side of the box. However, because the integral goes around the box in a direction, all factors are actually 1. Also, each corner only represents half a Riemann box, so the factor for the corner contributions is 0.5.

To compute the far field from this data, the integrations are approximated using a simple Riemann sum. This can be modified to more elaborate numerical integration methods by adjusting the factor element of `ffBox`. Once the integrations are completed, the values are saved and normalized. Finally, the far field pattern is printed.

```
#include <iostream>
#include <math.h>
#include <complex>
#define pi  M_PI
  ;
#include "meepCyl.h"
  ;
  ; void ffCalc(void){ ; std::complex < double > ctmp, sum, ij(0,1);
      double rho, phiP, angle, phi, ffMax;
      int Iphi, i;
      double factor, eta, x, y;
      ;
      ;
      ffMax = 0.;
      eta = 1.;
      ;
      for (Iphi = 0; Iphi < 361; Iphi++) {
        phi = (double) Iphi * pi/180.;
        ;
        sum = 0.;
        ;
        for (i = 0; i < NumPts; i++) {
          x = (ffBox + i)->x;
          y = (ffBox + i)->y;
          factor = (ffBox + i)->f;
          rho = sqrt(x * x + y * y);
          phiP = atan2(y, x);
          angle = 2. * pi * rho * cos(phi - phiP);
          ctmp = exp(ij * angle) * (ffBox + i)->Jeq * factor * eta;
          ;
          sum += ctmp;
          ;
          ctmp = exp(ij * angle) * (ffBox + i)->Meq * factor;
          if ((i < NumPts/4) ∨ ((i ≥ 2 * NumPts/4) ∧ (i < 3 * NumPts/4))) ctmp *= sin(phi);
          else ctmp *= cos(phi) * (−1.);
          ;
          sum += ctmp;
        }
        FF[Iphi] = abs(sum);
```

```
  if (FF[Iphi] > ffMax) ffMax = FF[Iphi];
}
;
;     /* Normalize the pattern */
;
for (Iphi = 0; Iphi < 361; Iphi++) FF[Iphi] = FF[Iphi]/ffMax;
}
```

15. Scripts.

There are a number of script files used in the simulations. These are listed here. The script files include bash shell scripts, awk code, and a number of exact pattern data files.

doPW.sh

This script executes the code and performs a simple calculation to determine the RMSE for the pattern compared to the analytic (or GMT) solution for the plane wave case.

```
#
# script to run meep on cylinder
#
#  there are 10 possible parameters:
#
#  cylinder radius:  CylRad
#  FFsize         :  FFsize
#  Pml            :  Pml
#  FFe            :  FFe
#  PwFFboxDist    :  PwFFboxDist
#  PWle           :  PWle
#  PWsize         :  PWsize
#  resolution     :  RES
#  NumTimeSteps   :  NumSteps
#  delta          :  delta
#
# To use this script, you need to:
#   A.  comment out the two variables that are being swept in the
#       code that sets the default values
#
#   B.  set up the default value to be $2 and/or $3 ($1 is output file name)

epsR=-5
Pw1Mono0=1
CylRad=0.5
FFsize=3
Pml=0.25
FFe=0.75
PwFFboxDist=0.5
PWle=0.25
PWsize=1.0
RES=40
NumSteps=200
delta=0.025
courant=0.5
#
# name of analytic ff data file
#
Exact=FFgmt.jrd
# Exact=rp5e2p56gmt.jrd
```

```
#
#  here, set the actual input values
#
#  default values have RMSE=32.2%
#
# Pml=0.175 min RMSE at 28.8% with all others at default
# FFsize no clear min RMSE (oscillations)
#

cat inFile.jrd |
sed s/epsR/$epsR/g |
sed s/Pw1Mono0/$Pw1Mono0/g |
sed s/CylRad/$CylRad/g |
sed s/FFsize/$FFsize/g |
sed s/Pml/$Pml/g |
sed s/FFe/$FFe/g |
sed s/PwFFboxDist/$PwFFboxDist/g |
sed s/PWle/$PWle/g |
sed s/PWsize/$PWsize/g |
sed s/RES/$RES/g |
sed s/NumSteps/$NumSteps/g |
sed s/delta/$delta/g |
sed s/courant/$courant/g >$1.in
#
#
#
meepCyl<$1.in>$1.out
grep FFP $1.out |awk -F ":" '{print $2}' >$1.dat
paste $1.dat $Exact >$1.sum
awk -f pat.awk $1.sum
```

inFile.jrd

This file is the template input file used to set up all input parameters using `doPW.sh`.

```
epsR
Pw1Mono0
CylRad
FFsize
Pml
FFe
PwFFboxDist
PWsize
PWle
RES
NumSteps
delta
courant
```

pat.awk

This file is the `awk` script that compares the far field pattern from the GMT solution with the MEEP far field pattern computed. The RMSE is computed using

$$\sqrt{\frac{1}{N}\sum_{\phi}\left(\frac{E_{GMT}(\phi)-E_{MEEP}(\phi)}{E_{GMT}(\phi)}\right)^2} \tag{11}$$

where N is the number of points in the summation. The value of ϕ is from 0 to 360. Since the endpoints are the same, the script skips the very first point ($\phi = 0$) and includes $\phi = 360°$.

It is well known that (11) is easily biased if the exact pattern ($4 in the script) is very small. To avoid biasing the RMSE error calculation, only points with an exact pattern larger than 30 dB below the peak are used in the sum of (11).

```
# $1, $3 are phi, $2 is E_{MEEP}, $4 is E_{GMT}
BEGIN {
    total=0;
    count=0;
}
NR!=1 {
    if($4>0.032)
    {
        temp=($2-$4)/$4;
        temp=temp*temp;
        total=total+temp;
        count++;
    }
}
END {
    total=total/count;
    total=sqrt(total);
    printf "%f", total;
}
```

doMono.sh

This script executes the code and performs a simple calculation to determine the RMSE for the pattern compared to the analytic (or GMT) solution for the monopole case.

```
#
# script to run meep on cylinder
#
#  there are 10 possible parameters:
#
#  cylinder radius:  CylRad
#  FFsize         :  FFsize
#  Pml            :  Pml
#  FFe            :  FFe
```

```
#  PwFFboxDist    :  PwFFboxDist
#  PWle           :  PWle
#  PWsize         :  PWsize
#  resolution     :  RES
#  NumTimeSteps   :  NumSteps
#  delta          :  delta
#
# To use this script, you need to:
#   A.  comment out the two variables that are being swept in the
#       code that sets the default values
#
#   B.  set up the default value to be $2 and/or $3 ($1 is output file name)

epsR=-5
Pw1Mono0=0
CylRad=0.5
FFsize=3
Pml=0.25
FFe=0.75
SrcLoc=0.5
RES=40
NumSteps=200
delta=0.025
courant=0.5
#
# name of analytic FF file name
#
Exact=MonoGMT.jrd
# Exact=MonoP5r1.jrd
#
#  here, set the actual input values
#
#  default values have RMSE = 1.11975
#
#

cat inMono.jrd |
sed s/epsR/$epsR/g |
sed s/Pw1Mono0/$Pw1Mono0/g |
sed s/CylRad/$CylRad/g |
sed s/FFsize/$FFsize/g |
sed s/Pml/$Pml/g |
sed s/FFe/$FFe/g |
sed s/SrcLoc/$SrcLoc/g |
sed s/RES/$RES/g |
sed s/NumSteps/$NumSteps/g |
sed s/delta/$delta/g |
sed s/courant/$courant/g >$1.in
#
#
#
meepCyl<$1.in>$1.out
```

```
grep FFP $1.out |awk -F ":" '{print $2}' >$1.dat
paste $1.dat $Exact >$1.sum
awk -f pat.awk $1.sum
```

inMono.jrd

This file is the template input file used to set up all input parameters using `doMono.sh`.

```
epsR
Pw1Mono0
CylRad
FFsize
Pml
FFe
SrcLoc
RES
NumSteps
delta
courant
```

16. NF2FF.

The near-field to far-field (`nf2ff.cc`) code computes the analytic solution for a plane wave scattered by a perfectly conducting cylinder. The electric and magnetic fields around a far-field box similar to the MEEP code are computed. Then, the near field to far field transformation is applied to the analytically computed fields.

The analytic expression for the electric field is:

$$E_z^s = E_o \sum_{n=-\infty}^{\infty} -j^n \frac{J_n(ka)}{H_n^{(2)}(ka)} H_n^{(2)}(k\rho) e^{jn\phi} \tag{12}$$

and for the magnetic field:

$$H_\phi^s = \frac{E_o}{j\eta} \sum_{n=-\infty}^{\infty} (j)^{-n} \frac{J_n(ka)}{H_n^{(2)}(ka)} H_n^{(2)}(k\rho) \left[\frac{n}{k\rho} H_n^{(2)}(k\rho) - H_{n+1}^{(2)}(k\rho) \right] e^{jn\phi} \tag{13}$$

where a is the radius of the cylinder, and

$$H_x = -H_\phi \sin\phi \qquad H_y = H_\phi \cos\phi \tag{14}$$

The `nf2ff` code has a makefile, listed here.

```
#
#  makefile for vector spherical wave functions
#
LCL=$(HOME)/bin/local
INCLS=$(LCL)/include
LIB=$(LCL)/lib
CC=g++ -Wall

nf2ff:                nf2ff.cc
               $(CC) -o nf2ff nf2ff.cc -I$(INCLS) -L$(LIB) \
               -lspf -lgsl -lgslcblas -lm

clean:
               rm -f *~ nf2ff nf2ff.ffp
```

nf2ff.cc

```
#include <stdlib.h>
#include <iostream>
#include <complex>
#include "std-def.h"
#include "spec_fns.h"

#define DELTA 0.001
#define NUMPTS 100000
#define TOL 1.e-9
#define FFSIZE 3.
// flag==0 -> Hx, flag==1 -> Hy

/*
    Function to compute the Magnetic field near the cylinder at (x,y)
*/

COMPLEX H(double x, double y, int flag)
{
  double rho,phi,a,k;
  double pi=M_PI,ka,kRho,eta;
  int n;
  COMPLEX ctmp1,ctmp2,ctmp3,sum;
  COMPLEX ij(0,1);
  a=.5;
  k=2.*pi;
  eta=sqrt(4.*pi*1.e-7/8.854e-12);
  ka=k*a;
  rho=sqrt(x*x+y*y);
  phi=atan2(y,x);

  kRho=k*rho;

  n=0;

  ctmp1=pow(ij,-n)/(ij*eta);
  ctmp2=j_n(n,ka)/h_n2(n,ka);
  ctmp3=(double)n/(kRho) * h_n2(n,kRho) - h_n2(n+1,kRho);
  ctmp2*=ctmp3;
  ctmp1*=ctmp2;
  ctmp1*=exp(ij*(double)n*phi);

  if(flag==0)
    ctmp1*=(-1.*sin(phi));
  if(flag==1)
    ctmp1*=cos(phi);

  sum=ctmp1;

  for(n=1;n<30;n++)
    {
```

```
      // +n term
      ctmp1=pow(ij,-n)/(ij*eta);
      ctmp2=j_n(n,ka)/h_n2(n,ka);
      ctmp3=(double)n/(kRho) * h_n2(n,kRho) - h_n2(n+1,kRho);
      ctmp2*=ctmp3;
      ctmp1*=ctmp2;
      ctmp1*=exp(ij*(double)n*phi);

      if(flag==0)
        ctmp1*=(-1.*sin(phi));
      if(flag==1)
        ctmp1*=cos(phi);

      sum+=ctmp1;

      // -n term
      ctmp1=pow(ij,n)/(ij*eta);
      //      ctmp1*=exp(-ij*(double)n*pi);
      ctmp2=j_n(-n,ka)/h_n2(-n,ka);
      ctmp3=(double)(-n)/(kRho) * h_n2(-n,kRho) - h_n2(-n+1,kRho);
      // in above, should it be -n-1 or -n+1 ?
      ctmp2*=ctmp3;
      ctmp1*=ctmp2;
      ctmp1*=exp(ij*(double)(-n)*phi);

      if(flag==0)
        ctmp1*=(-1.*sin(phi));

      if(flag==1)
        ctmp1*=cos(phi);

      sum+=ctmp1;

      if( (n>15) && (abs(ctmp1)/abs(sum)<TOL))
        n=29;

    }
  return sum;
}

/*
    Function to compute Electric Field near cylinder at (x,y)

    should it be + or minus (should be minus)
*/

COMPLEX E(double x, double y)
{
  double rho,phi,a,k;
  double pi=M_PI,ka,kRho;
  int n;
  COMPLEX ctmp1,ctmp2,ctmp3,sum;
```

```
  COMPLEX ij(0,1);
  a=0.5;
  k=2.*pi;
  ka=k*a;
  rho=sqrt(x*x+y*y);
  phi=atan2(y,x);

  kRho=k*rho;

  n=0;

  ctmp1=pow(ij,-n)*j_n(n,ka)/h_n2(n,ka);
  ctmp2=h_n2(n,kRho)*exp(ij*(double)n*phi);
  ctmp1*=ctmp2;

  sum=ctmp1;

  for(n=1;n<30;n++)
    {
      // +n term
      ctmp1=pow(ij,-n)*j_n(n,ka)/h_n2(n,ka);
      ctmp2=h_n2(n,kRho)*exp(ij*(double)n*phi);
      ctmp1*=ctmp2;

      sum+=ctmp1;
      // -n term
      ctmp1=pow(ij,n)*j_n(-n,ka)/h_n2(-n,ka);
      ctmp2=h_n2(-n,kRho)*exp(-ij*(double)n*phi);
      ctmp1*=ctmp2;

      sum+=ctmp1;

      if( (n>15) && (abs(ctmp1)/abs(sum)<TOL) )
        n=29;

    }
  return sum;
}

/*
     Main entry point for code, computes 4 sides and integrates
     equivalent currents to the far field
*/


int main()
{
  int i,flag,count;
  double xx,yy;
  double x[NUMPTS],y[NUMPTS],factor[NUMPTS],Jmag[NUMPTS],Jphs[NUMPTS];
  COMPLEX Mx[NUMPTS], My[NUMPTS];
  COMPLEX sum;
```

```
int iPhi,NumRec;
double phi,eta=sqrt(4.*M_PI*1.e-7/8.854e-12);

// to be safe, first zero all arrays
for(i=0;i<NUMPTS;i++)
  {
    x[i]=0.;
    y[i]=0.;
    factor[i]=0.;
    Jmag[i]=0.;
    Jphs[i]=0.;
    Mx[i]=0.;
    My[i]=0.;
  }

count=0;

/*   Side 1  */

flag=0;
yy=-FFSIZE;
for(xx=-FFSIZE; xx<=FFSIZE+DELTA/2.; xx=xx+DELTA)
  {
    // flag==0 -> Hxx, flag==1 -> Hyy
    sum=H(xx,yy,flag);
    x[count]=xx;
    y[count]=yy;
    factor[count]=1.;
    // Trapezoidal rule
    if(xx< -FFSIZE+DELTA/2.)
      {
        factor[count]*=0.5;
        //          COUT<<xx<<"  "<<yy<<"  "<<"factor  "<<factor[count]<<"  ";
      }
    if(xx>FFSIZE-DELTA/2.)
      {
                  factor[count]*=0.5;
        //          COUT<<factor[count]<<"  "<<xx<<"  "<<yy<<"\n";
      }
    Jmag[count]=abs(sum);
    Jphs[count]=atan2(imag(sum),real(sum));
    sum=E(xx,yy);
    Mx[count]=sum;
    count++;
    if(count>NUMPTS-1) exit(1);
  }

/*  Side 2  */

flag=1;
xx=FFSIZE;
for(yy=-FFSIZE; yy<=FFSIZE+DELTA/2.; yy=yy+DELTA)
```

```
  {
    // flag==0 -> Hxx, flag==1 -> Hyy
    sum=H(xx,yy,flag);

    x[count]=xx;
    y[count]=yy;
    factor[count]=1.;
    // Trapezoidal rule
    if(yy< -FFSIZE+DELTA/2.)
      {
        factor[count]*=0.5;
        //          COUT<<xx<<"  "<<yy<<"  "<<"factor  "<<factor[count]<<"  ";
      }
    if(yy>FFSIZE-DELTA/2.)
      {
        factor[count]*=0.5;
        //          COUT<<factor[count]<<"  "<<xx<<"  "<<yy<<"\n";
      }
    Jmag[count]=abs(sum);
    Jphs[count]=atan2(imag(sum),real(sum));
    sum=E(xx,yy);
    My[count]=sum;
    count++;
    if(count>NUMPTS-1) exit(1);

  }

/*  Side 3  */

flag=0;
yy=FFSIZE;
for(xx=FFSIZE; xx>=-FFSIZE-DELTA/2.; xx=xx-DELTA)
  {
    // flag==0 -> Hxx, flag==1 -> Hyy
    sum=H(xx,yy,flag);

    x[count]=xx;
    y[count]=yy;
    factor[count]= -1.;
    // Trapezoidal rule
    if(xx< -FFSIZE+DELTA/2.)
      {
        factor[count]*=0.5;
        //          COUT<<xx<<"  "<<yy<<"  "<<"factor  "<<factor[count]<<"  ";
      }
    if(xx>FFSIZE-DELTA/2.)
      {
        factor[count]*=0.5;
        //          COUT<<factor[count]<<"  "<<xx<<"  "<<yy<<"\n";
      }
    Jmag[count]=abs(sum);
    Jphs[count]=atan2(imag(sum),real(sum));
```

```
    sum=E(xx,yy);
    Mx[count]=sum;
    count++;
    if(count>NUMPTS-1) exit(1);


  }

/* Side 4  */

flag=1;
xx=-FFSIZE;
for(yy=FFSIZE; yy>=-FFSIZE-DELTA/2.; yy=yy-DELTA)
  {
    // flag==0 -> Hxx, flag==1 -> Hyy
    sum=H(xx,yy,flag);

    x[count]=xx;
    y[count]=yy;
    factor[count]= -1.;
    // Trapezoidal rule
    if(yy< -FFSIZE+DELTA/2.)
      {
        factor[count]*=0.5;
        //          COUT<<xx<<"  "<<yy<<"  "<<"factor  "<<factor[count]<<"  ";
      }
    if(yy>FFSIZE-DELTA/2.)
      {
        factor[count]*=0.5;
        //          COUT<<factor[count]<<"  "<<xx<<"  "<<yy<<"\n";
      }
    Jmag[count]=abs(sum);
    Jphs[count]=atan2(imag(sum),real(sum));
    sum=E(xx,yy);
    My[count]=sum;
    count++;
    if(count>NUMPTS-1) exit(1);


  }

NumRec=count;

double k,rho,phiP,angle,pi=M_PI,patMax=0.,pat[361];
COMPLEX ij(0,1),term,trm2,sum1,sum2;

k=2.*pi;

for(iPhi=0;iPhi<361;iPhi++)
  {
    phi=(double)iPhi*pi/180.;
```

```
        sum1=0.;
        sum2=0.;

        for(i=0;i<NumRec;i++)
          {
            // contribution from Jeq
            rho=sqrt(x[i]*x[i]+y[i]*y[i]);
            phiP=atan2(y[i],x[i]);
            angle=k*rho*cos(phi-phiP);
            term=Jmag[i]*exp(ij*Jphs[i])*exp(ij*angle)*factor[i];
            term*=eta;  // to fix uneven contributions
            // contribution from Meq
            trm2=(Mx[i]*sin(phi)-My[i]*cos(phi));
            trm2*=exp(ij*angle)*factor[i];
            if(iPhi==228)
              {
                // COUT<<" "<<abs(term)<<"  "<<abs(trm2)<<"  "<<abs(sum1)<<"  "<<abs(sum2)<<"
"<<abs(sum1+sum2)<<"\n";
              }
            sum1+=term;
            sum2+=trm2;
          }

        sum=sum1+sum2;
        //      COUT<<iPhi<<"  "<<abs(sum1)<<"  "<<abs(sum2)<<"\n";

        if(abs(sum)>patMax) patMax=abs(sum);
        pat[iPhi]=abs(sum);
        //      COUT<<iPhi<<"  "<<pat[iPhi]<<"  "<<phi<<"\n";

      }
    for(iPhi=0;iPhi<361;iPhi++)
      COUT<<" "<<iPhi<<"  "<<pat[iPhi]/patMax<<"\n";

    //  COUT<<"\n\n"<<patMax<<"\n";
    return 0;
  }
```

17. Simulation Results.

The simulation results have been disappointing. In this section we shall first investigate the near field to far field transformation described in the Far Field Calculation section of this document. This is followed by a look at the PEC cylinder case and finally, the dielectric cylinder case. Monopole line source incident field results may also be described at the end.

All simulations are for a cylinder of radius $a = 0.5\lambda$. Far field exact patterns are computed using GMT, see [7] and [8].

The near field to far field calculation appears to be working; however, there are some unusual effects apparent in the results. Consider using the exact expressions for the near fields to compute the far fields using the `nf2ff` program.

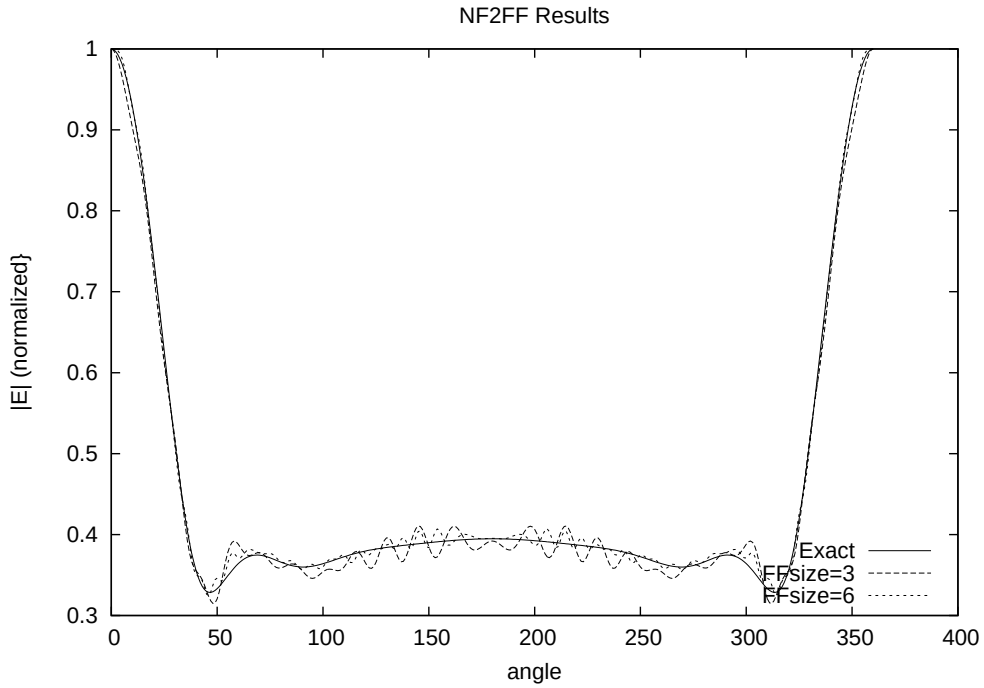

Figure 4. Far Field Pattern results from `nf2ff` at two different FFsize values.

The results for a perfectly conducting cylinder using the analytic near field expressions are shown in Fig. 4. There are some interesting features of Fig. 4. First, as the size of the far field box increases, the pattern appears to oscillate faster about the exact value. Second (and this may be fixed in the future), there are two data points that fall far from the exact value; these two points occur at roughly 38° and 228°.

The results, however, are somewhat encouraging. It does appear that the pattern is converging to the exact case. It has also been determined that the pattern is not sensitive to the number of points used on the far field box to do the integration.

The dependence of the MEEP results on the size of the far field box are illustrated by the data in Fig. 5. One does note the trend toward a smaller RMSE as the far field box size increases, along with some oscillations in the RMSE. For the most part, the results do not converge upon the pattern, as was expected.

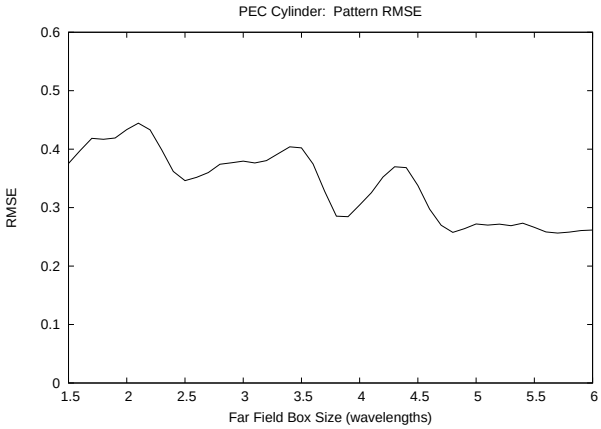

Figure 5. RMSE vs. FFsize.

The patterns for the case of FFsize at 2.5 and 5.7 λ are shown in Fig. 6. It can be seen that the oscillations present in the `nf2ff` data are also present here; however, the lack of convergence to the exact pattern is also apparent.

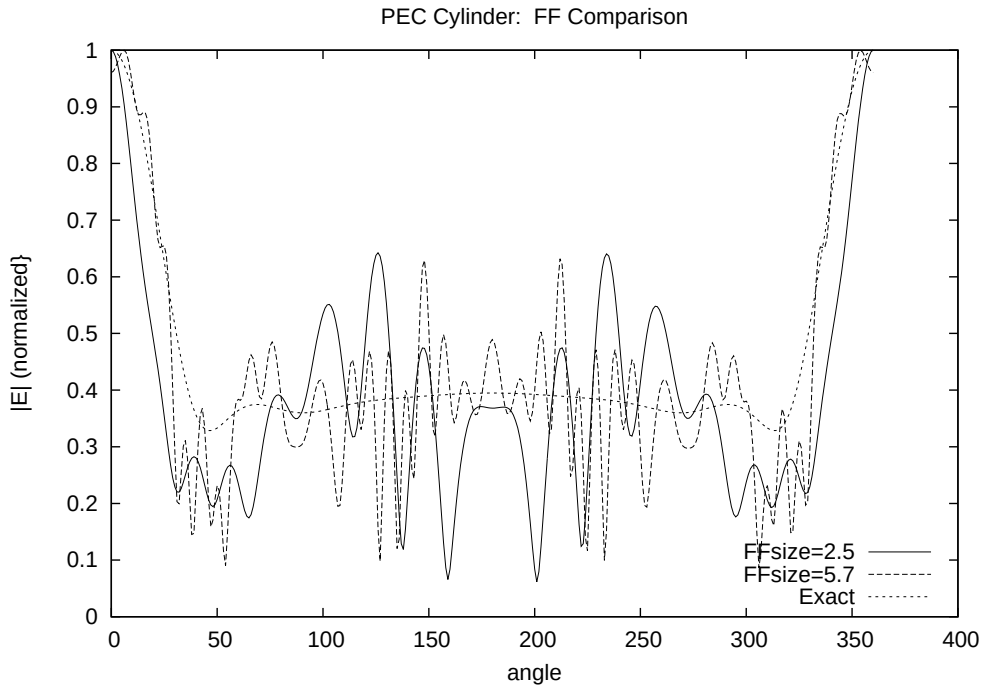

Figure 6. Pattern results for FFsize at 2.5 and 5.7 λ.

A large number of simulations were performed in addition to the very small sample provided above. One important item of note is that the MEEP code should be run using complex fields; the results were very unreliable if only real fields were used.

Once the time step reaches the point where data is collected, the fields are synchronized and restored after each step. Because the electric field and magnetic field are combined in the equivalence principle, the phases of the fields need to be aligned before data is collected.

It was suggested in [2] that using a small conductivity in the free space region would dampen the spurious ringing of the perfectly conducting scatter case. This was attempted, but no appreciable improvement was observed. Since the method to specify σ was discovered, the PEC cylinder results were tested using a large σ instead of $\epsilon_r = -\infty$. No changes were observed.

The simulations included single sweeps of many parameters, including the spacing between points on the far field box (Delta), the far field buffer size (FFe), the far field box size (FFsize, as discussed earlier), the thickness of the PML layer, the distance between the plane wave and the left edge (PWle), the plane wave extent (PWsize), the MEEP resolution, and the distance between the plane wave and the far field box.

Most of the single parameter simulations resulted in flat RMSE results. The PML thickness needs to be over 0.2λ. The plane wave size did show some small variations, along with changes in the distance between the far field box and the plane wave. These differences were deemed minor in comparison to the average RMSE values seen.

The number of iterations in the MEEP simulation was also tested. It was found that allowing the simulation to continue longer does not alter the RMSE results. Steady state conditions are achieved quite quickly. Changing the MEEP resolution also showed very insignificant changes in the results.

The cause of poor simulation results is not known; it is believed that the plane wave source is the cause of much of the error. Careful inspection of the fields in the vicinity of the cylinder indicate that there is a significant reflection of the plane wave off the $y = 0$ and $y = y_{max}$ walls. This is observed by the variation in the plane wave as y changes for constant x.

A number of ideas were implemented to create a more uniform plane wave; most without success. The most elaborate and promising idea was to allow the far field box buffer (FFe) to grow very large and keep the plane wave size small. This was expected to delay the reflections off the upper and lower boundaries. A two-dimensional sweep of FFe (set the same as the distance between the plane wave and the left edge) and PWsize was performed. The results were flat, i.e., no difference in RMSE was observed for any case.

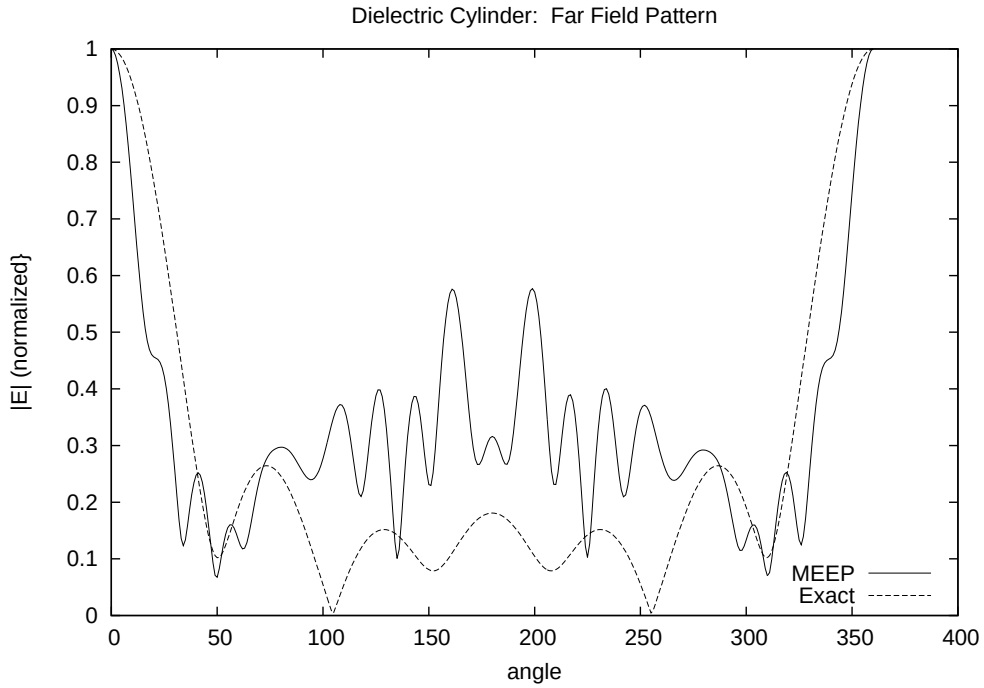

Figure 7. Pattern results for dielectric cylinder.

Fig. 7 shows the pattern results for a typical dielectric cylinder case ($\epsilon_r = 2.56$). One can readily see the oscillations in the MEEP pattern as compared to the exact pattern. In addition, the dielectric cylinder has side lobes that are quite small compared to the perfectly conducting case. This results in much larger RMSE values for all simulations.

The simulations performed for the dielectric cylinder are similar to the perfectly conducting case; a sweep of each individual parameter in the geometry was accomplished. The results were similar to the PEC case. The two-dimensional sweep to try and clean up the plane wave was not performed.

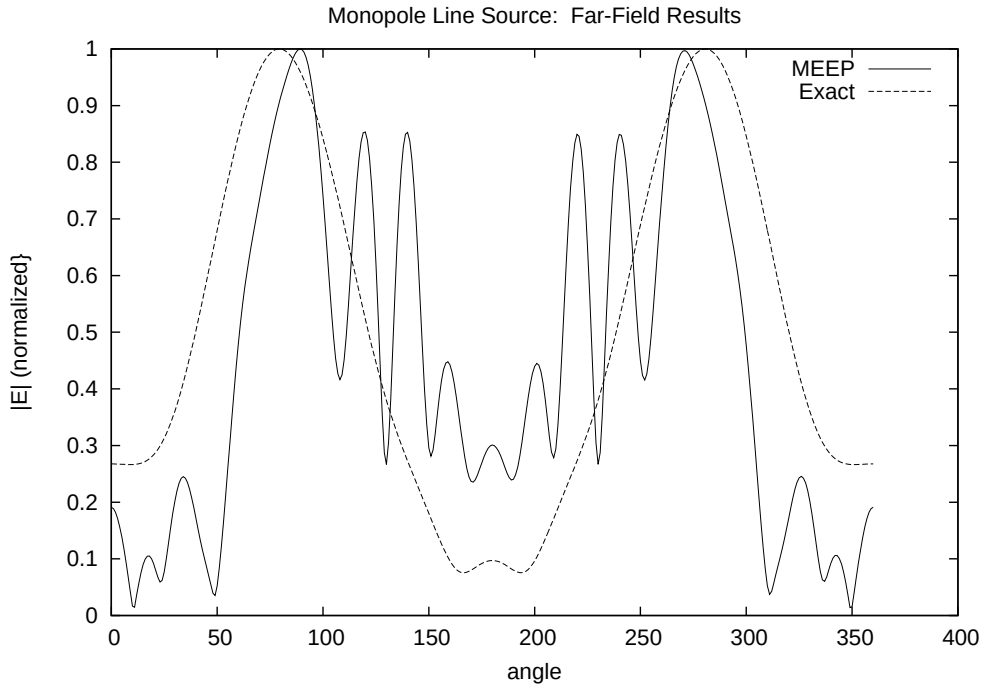

Figure 8. Pattern results for PEC cylinder scattering from a Monopole line source.

A monopole line source at $x = -1\lambda$ was investigated for the perfectly conducting and dielectric cylinder cases as well. In the case of a monopole, only the far field box size and far field box buffer size could be modified. The two-dimensional simulations performed showed poor RMSE that was not affected by geometrical changes.

Fig. 8 shows a typical far field MEEP pattern compared to the GMT pattern for the same geometry. A PEC cylinder is present in the case shown. One again sees the now-familiar ringing in the pattern as well as a sizable error between the exact and MEEP patterns.

The monopole results indicate that there is something besides the plane wave purity that is affecting the results. To investigate what may be causing additional error, a sweep of the PML thickness from 0.5 to 5 λ was simulated. The results were flat.

18. Future Work.

In the end, the results of the scattering by two-dimensional cylinder structures computed using the MEEP software was disappointing. There are a few places where the data provided here could be incorrect. The most likely source of error is the `nf2ff` calculations.

There are a few other items that could be tested to see if improvements are possible, besides the near field to far field transformation. One obvious change is to switch the numerical integration from a simple Riemann sum to a more elaborate numerical integration routine; however, this was briefly attempted with very little change in the results.

One more possible change to the present technique would be to alter the far field box from a square to a circle. This is certainly possible; use of a circular boundary in the equivalence principle also appears to be more "natural" for the calculations.

19. References.

[1] A. F. Oskooi, D. Roundy, M. Ibanescu, P. Bermel, J. D. Joannopoulos, and S. G. Johnson, "MEEP: A flexible free-software package for electromagnetic simulations by the FDTD method," *Computer Physics Communications*, vol. 181, pp. 687–702, January 2010.

[2] M. N. O. Sadiku, *Numerical Techniques in Electromagnetics*. New York, NY: CRC Press, second ed., 2000.

[3] A. Taflove, *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Boston, Massachusetts: Artech House, 1995.

[4] W. L. Stutzman and G. A. Thiele, *Antenna Theory and Design*. New York, NY: John Wiley **S**ons, second ed., 1998.

[5] J. E. Richie, "TLINE: Lossless transmission line simulation via the finite difference – time domain technique," Tech. Rep. 31, Marquette University, Electromagnetic Simulations Laboratory, Milwaukee, WI, Feb. 2005.

[6] J. E. Richie, "LCPP: A library for input and output operations," Tech. Rep. 15, Marquette University, Electromagnetic Simulations Laboratory, Milwaukee, WI, Mar. 2004.

[7] J. E. Richie, "GMT2PEC," Tech. Rep. 34, Marquette University, Electromagnetic Simulations Laboratory, Milwaukee, WI, Feb. 2007.

[8] J. E. Richie, "GMT2DIEL: The generalized multipole technique applied to two-dimensional dielectric scatterers," Tech. Rep. 41, Marquette University, Electromagnetics Simulations Laboratory, Milwaukee, WI, Aug. 2009.

20. Index.

MEEPCYL