EE535: Numerical Solutions to EM Problems

Homework 8

Name: Angelica Gutierrez

Due: 10 April 2025

Overview –

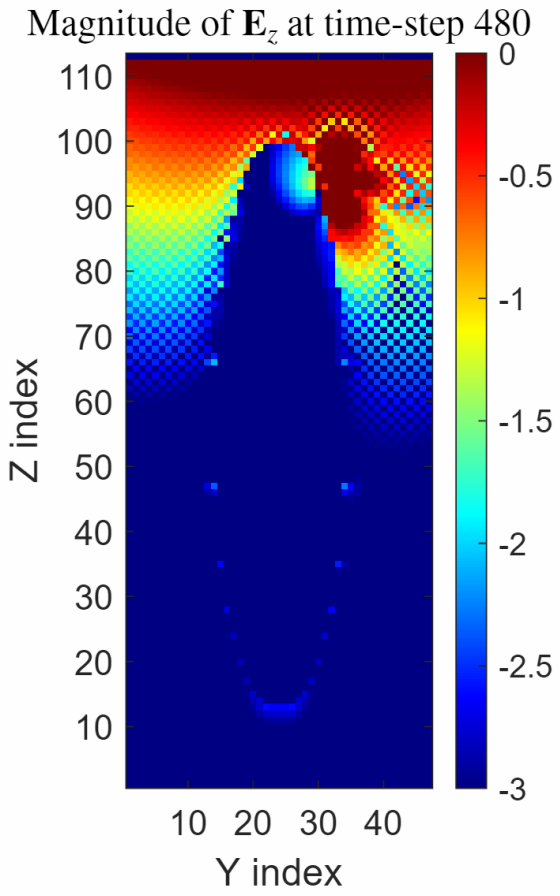
In this assignment, I implemented a 3D Finite-Difference Time-Domain (FDTD) simulation of a dipole source radiating near a dielectric ellipsoid using first-order absorbing boundary conditions (ABCs). The source emits a Ricker wavelet and is positioned just in front of the ellipsoid. According to the HW8 specifications, this configuration is meant to represent a person holding a cell phone—the ellipsoid approximates the human head, and the dipole models the phone. The simulation runs for 520 time steps, and I visualized the evolution of the Ez field over a constant-x plane to analyze how the wave radiates and interacts with the dielectric structure. This assignment builds upon the modular grid structure and snapshot strategies used in earlier homework but introduces a 3D geometry and updated source orientation.

Source Code Implementation –

The simulation is coordinated in hw8main.c, where I allocate the grid, initialize the dielectric ellipsoid, and inject the dipole source. Grid setup and material assignment are handled in grid3d.c, which sets the permittivity and loss characteristics of the ellipsoid based on spatial logic and modifies the update coefficients accordingly. The dipole source is implemented in ricker.c as a scaled Ricker wavelet and is injected into the Ez field at position (18, 32, 93). I used first-order ABCs from abc.c, which stores previous field values on the six boundary planes and apply updates to absorb outgoing waves. Time-stepping is handled in update3d.c, which updates all electric and magnetic field components across the 3D grid. Snapshots are written using snapshot3d.c, which outputs constant-x and constant-y Ez slices in binary format every 10 steps, based on user input at runtime.

Supporting Files & Results –

In addition to the main simulation files, I used fdtd-alloc.h, fdtd-macro.h, fdtd-proto.h, and fdtd-grid.h to define macros and data structures for memory allocation and field access. These were essential to maintaining a clean and modular implementation. After running the simulation for 520 time steps, I saved 128 constant-x Ez slices (every 10 steps) using snapshot3d.c. I then visualized the results in MATLAB using a log-scaled playback script. The animation successfully captures the source radiation, diffraction around the ellipsoid, and clean boundary absorption. The colormap and field intensity normalization closely match the example reference from the homework 8 PDF. For fun I included an mp4 file within my zip folder that displays an animation of my ellipsoid.



The snapshot at time-step 480 shown above clearly shows the wavefront propagating from the source and interacting with the ellipsoid. The figure confirms that the first-order ABCs are working properly—there are no visible reflections at the boundaries—and the energy visibly diffracts around the object, mimicking the scattering pattern from a ‘human-sized’ ellipsoid.

MATLAB Code

EE535_HW8_ELLIPSOID_480_SNAP_FINAL.m

```
%EE535 HW8 Angelica Gutierrez Code

basename = 'sim';        % filename
frameIndex = 48;         % 480 / stride (10)
stride = 10;
timestep = frameIndex * stride;

% Load the snapshot binary file
filename = sprintf('%s-x.%d', basename, frameIndex);
fid = fopen(filename, 'rb');
if fid == -1
    error("File not found: %s", filename);
end

dimY = fread(fid, 1, 'float');  % Y size (rows)
dimZ = fread(fid, 1, 'float');  % Z size (columns)
data = fread(fid, [dimY, dimZ], 'float');
fclose(fid);

Ez = flipud(data');

% === Log-magnitude scaling ===
epsilon = 1e-6;
Ez_mag = log10(abs(Ez) + epsilon);

% === Plot ===
figure;
imagesc(1:dimY, 1:dimZ, Ez_mag, [-3, 0]);  % dB scale from -3 to 0
axis image;
colormap(jet);
colorbar;

% Axis labeling and ticks
xticks(0:10:dimY);
yticks(0:10:dimZ);
xlabel('Y index', 'FontSize', 12);
ylabel('Z index', 'FontSize', 12);

title(sprintf('Magnitude of $\\mathbf{E}_z$ at time-step %d', timestep), ...
    'Interpreter', 'latex', 'FontSize', 14);

set(gca, 'YDir', 'normal', 'FontSize', 12);
```

C Codes

hw8main.c

```c
/* Simulation of a source next to an ellipsoid */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#include "fdtd-alloc.h"
#include "fdtd-macro.h"
#include "fdtd-proto.h"
#include "ezinc.h"

#define SEE_SRC  // optional: define to output src.dat

int main() {
  Grid *g;

  #ifdef SEE_SRC
  FILE *fid_src = fopen("src.dat", "w");
  #endif

  // --- Allocate and initialize grid ---
  ALLOC_1D(g, 1, Grid);       // allocate memory for grid structure
  gridInit(g);                // set grid size and ellipsoid coefficients
  ezIncInit(g);               // initialize dipole source (Ricker)
  abcInit(g);                 // initialize 1st-order ABC
  snapshot3dInit(g);          // setup for Ez field snapshots

  // --- Time-stepping loop ---
  for (Time = 0; Time < MaxTime; Time++) {
    updateH(g);               // update magnetic fields
    updateE(g);               // update electric fields

    // --- Apply dipole source (inject into Ez at specified location) ---
    Ez(18, 32, 93) += ezInc(Time, 0.0);

    #ifdef SEE_SRC
    fprintf(fid_src, "%g\n", Ez(18, 32, 93));   // log source value
    #endif

    abc(g);                   // apply absorbing boundary condition
    snapshot3d(g);            // take snapshots if conditions met
  }

  #ifdef SEE_SRC
  fclose(fid_src);
  #endif

  return 0;
}
```

grid3d.c

```c
#include "fdtd-macro.h"
#include "fdtd-alloc.h"
#include <math.h>

#define EPSR 50.0
#define LOSS_FACTOR (0.0434719)

#define SIZE_X (37)
#define SIZE_Y (47)
#define SIZE_Z (113)

#define A_DIM (0.098 / 0.02)
#define B_DIM (0.195 / 0.02)
#define C_DIM (0.875 / 0.02)

#define X_CNTR (int)(SIZE_X / 2)
#define Y_CNTR (int)(SIZE_Y / 2)
#define Z_CNTR (int)(SIZE_Z / 2)

int isInEllipsoid(double x, double y, double z);

void gridInit(Grid *g) {
  double imp0 = 377.0, coef1, coef2;
  int mm, nn, pp;

  Type = threeDGrid;
  SizeX = SIZE_X;
  SizeY = SIZE_Y;
  SizeZ = SIZE_Z;
  MaxTime = 520;
  Cdtds = 1.0 / sqrt(3.0);

  // Allocate memory
  ALLOC_3D(g->hx, SizeX, SizeY - 1, SizeZ - 1, double);
  ALLOC_3D(g->chxh, SizeX, SizeY - 1, SizeZ - 1, double);
  ALLOC_3D(g->chxe, SizeX, SizeY - 1, SizeZ - 1, double);

  ALLOC_3D(g->hy, SizeX - 1, SizeY, SizeZ - 1, double);
  ALLOC_3D(g->chyh, SizeX - 1, SizeY, SizeZ - 1, double);
  ALLOC_3D(g->chye, SizeX - 1, SizeY, SizeZ - 1, double);

  ALLOC_3D(g->hz, SizeX - 1, SizeY - 1, SizeZ, double);
  ALLOC_3D(g->chzh, SizeX - 1, SizeY - 1, SizeZ, double);
  ALLOC_3D(g->chze, SizeX - 1, SizeY - 1, SizeZ, double);

  ALLOC_3D(g->ex, SizeX - 1, SizeY, SizeZ, double);
  ALLOC_3D(g->cexe, SizeX - 1, SizeY, SizeZ, double);
  ALLOC_3D(g->cexh, SizeX - 1, SizeY, SizeZ, double);

  ALLOC_3D(g->ey, SizeX, SizeY - 1, SizeZ, double);
  ALLOC_3D(g->ceye, SizeX, SizeY - 1, SizeZ, double);
  ALLOC_3D(g->ceyh, SizeX, SizeY - 1, SizeZ, double);

  ALLOC_3D(g->ez, SizeX, SizeY, SizeZ - 1, double);
  ALLOC_3D(g->ceze, SizeX, SizeY, SizeZ - 1, double);
  ALLOC_3D(g->cezh, SizeX, SizeY, SizeZ - 1, double);

  // Free-space coefficients
  for (mm = 0; mm < SizeX - 1; mm++)
    for (nn = 0; nn < SizeY; nn++)
      for (pp = 0; pp < SizeZ; pp++) {
        Cexe(mm, nn, pp) = 1.0;
        Cexh(mm, nn, pp) = Cdtds * imp0;
      }

  for (mm = 0; mm < SizeX; mm++)
    for (nn = 0; nn < SizeY - 1; nn++)
      for (pp = 0; pp < SizeZ; pp++) {
        Ceye(mm, nn, pp) = 1.0;
        Ceyh(mm, nn, pp) = Cdtds * imp0;
      }

  for (mm = 0; mm < SizeX; mm++)
    for (nn = 0; nn < SizeY; nn++)
      for (pp = 0; pp < SizeZ - 1; pp++) {
        Ceze(mm, nn, pp) = 1.0;
        Cezh(mm, nn, pp) = Cdtds * imp0;
      }
```

```
  // Ellipsoid coefficients
  coef1 = (1.0 - LOSS_FACTOR) / (1.0 + LOSS_FACTOR);
  coef2 = Cdtds * imp0 / EPSR / (1.0 + LOSS_FACTOR);

  for (mm = 0; mm < SizeX - 1; mm++)
    for (nn = 0; nn < SizeY - 1; nn++)
      for (pp = 0; pp < SizeZ - 1; pp++) {
        if (isInEllipsoid(mm + 0.5, nn + 0.5, pp + 0.5)) {
          Cexe(mm, nn, pp) = coef1;
          Cexh(mm, nn, pp) = coef2;
          Cexe(mm, nn + 1, pp) = coef1;
          Cexh(mm, nn + 1, pp) = coef2;
          Cexe(mm, nn, pp + 1) = coef1;
          Cexh(mm, nn, pp + 1) = coef2;
          Cexe(mm, nn + 1, pp + 1) = coef1;
          Cexh(mm, nn + 1, pp + 1) = coef2;

          Ceye(mm, nn, pp) = coef1;
          Ceyh(mm, nn, pp) = coef2;
          Ceye(mm + 1, nn, pp) = coef1;
          Ceyh(mm + 1, nn, pp) = coef2;
          Ceye(mm, nn, pp + 1) = coef1;
          Ceyh(mm, nn, pp + 1) = coef2;
          Ceye(mm + 1, nn, pp + 1) = coef1;
          Ceyh(mm + 1, nn, pp + 1) = coef2;

          Ceze(mm, nn, pp) = coef1;
          Cezh(mm, nn, pp) = coef2;
          Ceze(mm + 1, nn, pp) = coef1;
          Cezh(mm + 1, nn, pp) = coef2;
          Ceze(mm, nn + 1, pp) = coef1;
          Cezh(mm, nn + 1, pp) = coef2;
          Ceze(mm + 1, nn + 1, pp) = coef1;
          Cezh(mm + 1, nn + 1, pp) = coef2;
        }
      }

  // Magnetic field coefficients
  for (mm = 0; mm < SizeX; mm++)
    for (nn = 0; nn < SizeY - 1; nn++)
      for (pp = 0; pp < SizeZ - 1; pp++) {
        Chxh(mm, nn, pp) = 1.0;
        Chxe(mm, nn, pp) = Cdtds / imp0;
      }

  for (mm = 0; mm < SizeX - 1; mm++)
    for (nn = 0; nn < SizeY; nn++)
      for (pp = 0; pp < SizeZ - 1; pp++) {
        Chyh(mm, nn, pp) = 1.0;
        Chye(mm, nn, pp) = Cdtds / imp0;
      }

  for (mm = 0; mm < SizeX - 1; mm++)
    for (nn = 0; nn < SizeY - 1; nn++)
      for (pp = 0; pp < SizeZ; pp++) {
        Chzh(mm, nn, pp) = 1.0;
        Chze(mm, nn, pp) = Cdtds / imp0;
      }

  return;
}

int isInEllipsoid(double x, double y, double z) {
  double x_d = (x - X_CNTR) / A_DIM;
  double y_d = (y - Y_CNTR) / B_DIM;
  double z_d = (z - Z_CNTR) / C_DIM;

  return (x_d * x_d + y_d * y_d + z_d * z_d < 1.0);
}
```

ricker.c

```c
#include "ezinc.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

static double cdtds, ppw = 0;
static double scale = 100.0;  // Scaling factor to boost Ricker wavelet

/* initialize source-function variables */
void ezIncInit(Grid *g) {
  printf("Enter the points per wavelength for Ricker source: ");
  scanf(" %lf", &ppw);
  cdtds = Cdtds;
  return;
}

/* calculate source function at given time and location */
double ezInc(double time, double location) {
  double arg;

  if (ppw <= 0) {
    fprintf(stderr,
       "ezInc: ezIncInit() must be called before ezInc.\n"
       "       Points per wavelength must be positive.\n");
    exit(-1);
  }

  arg = M_PI * ((cdtds * time - location) / ppw - 1.0);
  arg = arg * arg;

  return scale * (1.0 - 2.0 * arg) * exp(-arg);
}
```

ezinc.h

```c
#ifndef _EZINC_H
#define _EZINC_H

#include "fdtd-macro.h"  // For Grid struct and macros

void ezIncInit(Grid *g);
double ezInc(double time, double location);  // changed 'int time' to 'double time'

#endif
```

snapshot3d.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "fdtd-macro.h"

static int temporalStride = -2, frameX = 0, frameY = 0, startTime;
static char basename[80];

void snapshot3dInit(Grid *g) {
  int choice;

  printf("Do you want 2D snapshots of the 3D grid? (1=yes, 0=no): ");
  scanf("%d", &choice);
  if (choice == 0) {
    temporalStride = -1;
    return;
  }

  printf("Duration of simulation is %d steps.\n", MaxTime);
  printf("Enter start time and temporal stride: ");
  scanf(" %d %d", &startTime, &temporalStride);

  printf("Enter the base name: ");
  scanf(" %s", basename);

  return;
}

void snapshot3d(Grid *g) {
  int mm, nn, pp;
  float dim1, dim2, temp;
  char filename[100];
  FILE *out;

  if (temporalStride == -1)
    return;

  if (temporalStride < -1) {
    fprintf(stderr,
      "snapshot3d: snapshot3dInit must be called before snapshot.\n"
      "            Temporal stride must be set to a positive value.\n");
    exit(-1);
  }

  if (Time >= startTime && (Time - startTime) % temporalStride == 0) {
    // -------- Write constant-x slice --------
    sprintf(filename, "%s-x.%d", basename, frameX++);
    out = fopen(filename, "wb");

    dim1 = SizeY;
    dim2 = SizeZ;
    fwrite(&dim1, sizeof(float), 1, out);
    fwrite(&dim2, sizeof(float), 1, out);

    mm = (SizeX - 1) / 2;  // middle x-slice (x = 18)
    for (pp = SizeZ - 1; pp >= 0; pp--) {
      for (nn = 0; nn < SizeY; nn++) {
        temp = (float)Ez(mm, nn, pp);
        fwrite(&temp, sizeof(float), 1, out);
      }
    }
    fclose(out);

    // -------- Write constant-y slice --------
    sprintf(filename, "%s-y.%d", basename, frameY++);
    out = fopen(filename, "wb");

    dim1 = SizeX - 1;
    dim2 = SizeZ;
    fwrite(&dim1, sizeof(float), 1, out);
    fwrite(&dim2, sizeof(float), 1, out);

    nn = SizeY / 2;  // middle y-slice
    for (pp = SizeZ - 1; pp >= 0; pp--) {
      for (mm = 0; mm < SizeX - 1; mm++) {
        temp = (float)Ez(mm, nn, pp);
        fwrite(&temp, sizeof(float), 1, out);
      }
    }
    fclose(out);
  }

  return;
}
```

update3d.c

```c
#include "fdtd-macro.h"
#include <stdio.h>

/* Update magnetic fields */
void updateH(Grid *g) {
  int mm, nn, pp;

  if (Type == oneDGrid) {
    for (mm = 0; mm < SizeX - 1; mm++)
      Hy1(mm) = Chyh1(mm) * Hy1(mm)
              + Chye1(mm) * (Ez1(mm + 1) - Ez1(mm));

  } else if (Type == tmZGrid) {
    for (mm = 0; mm < SizeX; mm++)
      for (nn = 0; nn < SizeY - 1; nn++)
        Hx2(mm, nn) = Chxh2(mm, nn) * Hx2(mm, nn)
                    - Chxe2(mm, nn) * (Ez2(mm, nn + 1) - Ez2(mm, nn));

    for (mm = 0; mm < SizeX - 1; mm++)
      for (nn = 0; nn < SizeY; nn++)
        Hy2(mm, nn) = Chyh2(mm, nn) * Hy2(mm, nn)
                    + Chye2(mm, nn) * (Ez2(mm + 1, nn) - Ez2(mm, nn));

  } else if (Type == teZGrid) {
    for (mm = 0; mm < SizeX - 1; mm++)
      for (nn = 0; nn < SizeY - 1; nn++)
        Hz2(mm, nn) = Chzh2(mm, nn) * Hz2(mm, nn)
                    - Chze2(mm, nn) * ((Ey2(mm + 1, nn) - Ey2(mm, nn)) -
                                       (Ex2(mm, nn + 1) - Ex2(mm, nn)));

  } else if (Type == threeDGrid) {
    for (mm = 0; mm < SizeX; mm++)
      for (nn = 0; nn < SizeY - 1; nn++)
        for (pp = 0; pp < SizeZ - 1; pp++)
          Hx(mm, nn, pp) = Chxh(mm, nn, pp) * Hx(mm, nn, pp)
                         + Chxe(mm, nn, pp) * ((Ey(mm, nn, pp + 1) - Ey(mm, nn, pp)) -
                                               (Ez(mm, nn + 1, pp) - Ez(mm, nn, pp)));

    for (mm = 0; mm < SizeX - 1; mm++)
      for (nn = 0; nn < SizeY; nn++)
        for (pp = 0; pp < SizeZ - 1; pp++)
          Hy(mm, nn, pp) = Chyh(mm, nn, pp) * Hy(mm, nn, pp)
                         + Chye(mm, nn, pp) * ((Ez(mm + 1, nn, pp) - Ez(mm, nn, pp)) -
                                               (Ex(mm, nn, pp + 1) - Ex(mm, nn, pp)));

    for (mm = 0; mm < SizeX - 1; mm++)
      for (nn = 0; nn < SizeY - 1; nn++)
        for (pp = 0; pp < SizeZ; pp++)
          Hz(mm, nn, pp) = Chzh(mm, nn, pp) * Hz(mm, nn, pp)
                         + Chze(mm, nn, pp) * ((Ex(mm, nn + 1, pp) - Ex(mm, nn, pp)) -
                                               (Ey(mm + 1, nn, pp) - Ey(mm, nn, pp)));

  } else {
    fprintf(stderr, "updateH: Unknown grid type. Terminating...\n");
  }

  return;
}

/* Update electric fields */
void updateE(Grid *g) {
  int mm, nn, pp;

  if (Type == oneDGrid) {
    for (mm = 1; mm < SizeX - 1; mm++)
      Ez1(mm) = Ceze1(mm) * Ez1(mm)
              + Cezh1(mm) * (Hy1(mm) - Hy1(mm - 1));

  } else if (Type == tmZGrid) {
    for (mm = 1; mm < SizeX - 1; mm++)
      for (nn = 1; nn < SizeY - 1; nn++)
        Ez2(mm, nn) = Ceze2(mm, nn) * Ez2(mm, nn)
                    + Cezh2(mm, nn) * ((Hy2(mm, nn) - Hy2(mm - 1, nn)) -
                                       (Hx2(mm, nn) - Hx2(mm, nn - 1)));

  } else if (Type == teZGrid) {
    for (mm = 1; mm < SizeX - 1; mm++)
      for (nn = 1; nn < SizeY - 1; nn++)
        Ex2(mm, nn) = Cexe2(mm, nn) * Ex2(mm, nn)
                    + Cexh2(mm, nn) * (Hz2(mm, nn) - Hz2(mm, nn - 1));

    for (mm = 1; mm < SizeX - 1; mm++)
      for (nn = 1; nn < SizeY - 1; nn++)
        Ey2(mm, nn) = Ceye2(mm, nn) * Ey2(mm, nn)
                    - Ceyh2(mm, nn) * (Hz2(mm, nn) - Hz2(mm - 1, nn));

  } else if (Type == threeDGrid) {
    for (mm = 0; mm < SizeX - 1; mm++)
      for (nn = 1; nn < SizeY - 1; nn++)
        for (pp = 1; pp < SizeZ - 1; pp++)
          Ex(mm, nn, pp) = Cexe(mm, nn, pp) * Ex(mm, nn, pp)
                         + Cexh(mm, nn, pp) * ((Hz(mm, nn, pp) - Hz(mm, nn - 1, pp)) -
                                               (Hy(mm, nn, pp) - Hy(mm, nn, pp - 1)));

    for (mm = 1; mm < SizeX - 1; mm++)
      for (nn = 0; nn < SizeY - 1; nn++)
        for (pp = 1; pp < SizeZ - 1; pp++)
          Ey(mm, nn, pp) = Ceye(mm, nn, pp) * Ey(mm, nn, pp)
                         + Ceyh(mm, nn, pp) * ((Hx(mm, nn, pp) - Hx(mm, nn, pp - 1)) -
                                               (Hz(mm, nn, pp) - Hz(mm - 1, nn, pp)));

    for (mm = 1; mm < SizeX - 1; mm++)
      for (nn = 1; nn < SizeY - 1; nn++)
        for (pp = 0; pp < SizeZ - 1; pp++)
          Ez(mm, nn, pp) = Ceze(mm, nn, pp) * Ez(mm, nn, pp)
                         + Cezh(mm, nn, pp) * ((Hy(mm, nn, pp) - Hy(mm - 1, nn, pp)) -
                                               (Hx(mm, nn, pp) - Hx(mm, nn - 1, pp)));

  } else {
    fprintf(stderr, "updateE: Unknown grid type. Terminating...\n");
  }

  return;
}
```

abc.c

```c
#include "fdtd-alloc.h"
#include "fdtd-macro.h"
#include <stdio.h>
#include <stdlib.h>

/* Global ABC coefficient */
static double abccoef = 0.0;

/* Arrays to hold previous Ez values on all six faces */
static double *ezx0, *ezx1;
static double *ezy0, *ezy1;
static double *ezz0, *ezz1;

void abcInit(Grid *g) {
  abccoef = (Cdtds - 1.0) / (Cdtds + 1.0);

  /* Allocate memory for the six boundary planes */
  ALLOC_2D(ezx0, SizeY, SizeZ - 1, double);
  ALLOC_2D(ezx1, SizeY, SizeZ - 1, double);
  ALLOC_2D(ezy0, SizeX, SizeZ - 1, double);
  ALLOC_2D(ezy1, SizeX, SizeZ - 1, double);
  ALLOC_2D(ezz0, SizeX, SizeY, double);
  ALLOC_2D(ezz1, SizeX, SizeY, double);
}

void abc(Grid *g) {
  int mm, nn, pp;

  if (abccoef == 0.0) {
    fprintf(stderr,
      "abc: abcInit must be called before abc. Terminating...\n");
    exit(-1);
  }

  /* X = 0 and X = SizeX - 1 */
  for (nn = 0; nn < SizeY; nn++)
    for (pp = 0; pp < SizeZ - 1; pp++) {
      Ez(0, nn, pp) = ezx0[nn * (SizeZ - 1) + pp] +
        abccoef * (Ez(1, nn, pp) - Ez(0, nn, pp));
      ezx0[nn * (SizeZ - 1) + pp] = Ez(1, nn, pp);

      Ez(SizeX - 1, nn, pp) = ezx1[nn * (SizeZ - 1) + pp] +
        abccoef * (Ez(SizeX - 2, nn, pp) - Ez(SizeX - 1, nn, pp));
      ezx1[nn * (SizeZ - 1) + pp] = Ez(SizeX - 2, nn, pp);
    }

  /* Y = 0 and Y = SizeY - 1 */
  for (mm = 0; mm < SizeX; mm++)
    for (pp = 0; pp < SizeZ - 1; pp++) {
      Ez(mm, 0, pp) = ezy0[mm * (SizeZ - 1) + pp] +
        abccoef * (Ez(mm, 1, pp) - Ez(mm, 0, pp));
      ezy0[mm * (SizeZ - 1) + pp] = Ez(mm, 1, pp);

      Ez(mm, SizeY - 1, pp) = ezy1[mm * (SizeZ - 1) + pp] +
        abccoef * (Ez(mm, SizeY - 2, pp) - Ez(mm, SizeY - 1, pp));
      ezy1[mm * (SizeZ - 1) + pp] = Ez(mm, SizeY - 2, pp);
    }

  /* Z = 0 and Z = SizeZ - 1 */
  for (mm = 0; mm < SizeX; mm++)
    for (nn = 0; nn < SizeY; nn++) {
      Ez(mm, nn, 0) = ezz0[mm * SizeY + nn] +
        abccoef * (Ez(mm, nn, 1) - Ez(mm, nn, 0));
      ezz0[mm * SizeY + nn] = Ez(mm, nn, 1);

      Ez(mm, nn, SizeZ - 2) = ezz1[mm * SizeY + nn] +
        abccoef * (Ez(mm, nn, SizeZ - 3) - Ez(mm, nn, SizeZ - 2));
      ezz1[mm * SizeY + nn] = Ez(mm, nn, SizeZ - 3);
    }

  return;
}
```

fdtd-macro.h

```
#ifndef _FDTD_MACRO_H
#define _FDTD_MACRO_H

#include "fdtd-grid.h"

/* macros that permit the "Grid" to be specified */
/* one-dimensional grid */
#define Hy1G(G, M)          G->hy[M]
#define Chyh1G(G, M)        G->chyh[M]
#define Chye1G(G, M)        G->chye[M]

#define Ez1G(G, M)          G->ez[M]
#define Ceze1G(G, M)        G->ceze[M]
#define Cezh1G(G, M)        G->cezh[M]

/* TMz grid */
#define Hx2G(G, M, N)       G->hx[(M) * (SizeYG(G) - 1) + N]
#define Chxh2G(G, M, N)     G->chxh[(M) * (SizeYG(G) - 1) + N]
#define Chxe2G(G, M, N)     G->chxe[(M) * (SizeYG(G) - 1) + N]

#define Hy2G(G, M, N)       G->hy[(M) * SizeYG(G) + N]
#define Chyh2G(G, M, N)     G->chyh[(M) * SizeYG(G) + N]
#define Chye2G(G, M, N)     G->chye[(M) * SizeYG(G) + N]

#define Ez2G(G, M, N)       G->ez[(M) * SizeYG(G) + N]
#define Ceze2G(G, M, N)     G->ceze[(M) * SizeYG(G) + N]
#define Cezh2G(G, M, N)     G->cezh[(M) * SizeYG(G) + N]

/* TEz grid */
#define Ex2G(G, M, N)       G->ex[(M) * SizeYG(G) + N]
#define Cexe2G(G, M, N)     G->cexe[(M) * SizeYG(G) + N]
#define Cexh2G(G, M, N)     G->cexh[(M) * SizeYG(G) + N]

#define Ey2G(G, M, N)       G->ey[(M) * (SizeYG(G) - 1) + N]
#define Ceye2G(G, M, N)     G->ceye[(M) * (SizeYG(G) - 1) + N]
#define Ceyh2G(G, M, N)     G->ceyh[(M) * (SizeYG(G) - 1) + N]

#define Hz2G(G, M, N)       G->hz[(M) * (SizeYG(G) - 1) + N]
#define Chzh2G(G, M, N)     G->chzh[(M) * (SizeYG(G) - 1) + N]
#define Chze2G(G, M, N)     G->chze[(M) * (SizeYG(G) - 1) + N]

/* 3D grid */
#define HxG(G, M, N, P)     G->hx[((M) * (SizeYG(G) - 1) + N) * (SizeZG(G) - 1) + P]
#define ChxhG(G, M, N, P)   G->chxh[((M) * (SizeYG(G) - 1) + N) * (SizeZG(G) - 1) + P]
#define ChxeG(G, M, N, P)   G->chxe[((M) * (SizeYG(G) - 1) + N) * (SizeZG(G) - 1) + P]

#define HyG(G, M, N, P)     G->hy[((M) * SizeYG(G) + N) * (SizeZG(G) - 1) + P]
#define ChyhG(G, M, N, P)   G->chyh[((M) * SizeYG(G) + N) * (SizeZG(G) - 1) + P]
#define ChyeG(G, M, N, P)   G->chye[((M) * SizeYG(G) + N) * (SizeZG(G) - 1) + P]

#define HzG(G, M, N, P)     G->hz[((M) * (SizeYG(G) - 1) + N) * SizeZG(G) + P]
#define ChzhG(G, M, N, P)   G->chzh[((M) * (SizeYG(G) - 1) + N) * SizeZG(G) + P]
#define ChzeG(G, M, N, P)   G->chze[((M) * (SizeYG(G) - 1) + N) * SizeZG(G) + P]

#define ExG(G, M, N, P)     G->ex[((M) * SizeYG(G) + N) * SizeZG(G) + P]
#define CexeG(G, M, N, P)   G->cexe[((M) * SizeYG(G) + N) * SizeZG(G) + P]
#define CexhG(G, M, N, P)   G->cexh[((M) * SizeYG(G) + N) * SizeZG(G) + P]

#define EyG(G, M, N, P)     G->ey[((M) * (SizeYG(G) - 1) + N) * SizeZG(G) + P]
#define CeyeG(G, M, N, P)   G->ceye[((M) * (SizeYG(G) - 1) + N) * SizeZG(G) + P]
#define CeyhG(G, M, N, P)   G->ceyh[((M) * (SizeYG(G) - 1) + N) * SizeZG(G) + P]

#define EzG(G, M, N, P)     G->ez[((M) * SizeYG(G) + N) * (SizeZG(G) - 1) + P]
#define CezeG(G, M, N, P)   G->ceze[((M) * SizeYG(G) + N) * (SizeZG(G) - 1) + P]
#define CezhG(G, M, N, P)   G->cezh[((M) * SizeYG(G) + N) * (SizeZG(G) - 1) + P]

/* Grid parameters */
#define SizeXG(G)           G->sizeX
#define SizeYG(G)           G->sizeY
#define SizeZG(G)           G->sizeZ
#define TimeG(G)            G->time
#define MaxTimeG(G)         G->maxTime
#define CdtdsG(G)           G->cdtds
#define TypeG(G)            G->type
```

```
/* macros that assume the "Grid" is 'g' */
#define Hy1(M)              Hy1G(g, M)
#define Chyh1(M)            Chyh1G(g, M)
#define Chye1(M)            Chye1G(g, M)

#define Ez1(M)              Ez1G(g, M)
#define Ceze1(M)            Ceze1G(g, M)
#define Cezh1(M)            Cezh1G(g, M)

#define Hx2(M, N)           Hx2G(g, M, N)
#define Chxh2(M, N)         Chxh2G(g, M, N)
#define Chxe2(M, N)         Chxe2G(g, M, N)

#define Hy2(M, N)           Hy2G(g, M, N)
#define Chyh2(M, N)         Chyh2G(g, M, N)
#define Chye2(M, N)         Chye2G(g, M, N)

#define Ez2(M, N)           Ez2G(g, M, N)
#define Ceze2(M, N)         Ceze2G(g, M, N)
#define Cezh2(M, N)         Cezh2G(g, M, N)

#define Hz2(M, N)           Hz2G(g, M, N)
#define Chzh2(M, N)         Chzh2G(g, M, N)
#define Chze2(M, N)         Chze2G(g, M, N)

#define Ex2(M, N)           Ex2G(g, M, N)
#define Cexe2(M, N)         Cexe2G(g, M, N)
#define Cexh2(M, N)         Cexh2G(g, M, N)

#define Ey2(M, N)           Ey2G(g, M, N)
#define Ceye2(M, N)         Ceye2G(g, M, N)
#define Ceyh2(M, N)         Ceyh2G(g, M, N)

#define Hx(M, N, P)         HxG(g, M, N, P)
#define Chxh(M, N, P)       ChxhG(g, M, N, P)
#define Chxe(M, N, P)       ChxeG(g, M, N, P)

#define Hy(M, N, P)         HyG(g, M, N, P)
#define Chyh(M, N, P)       ChyhG(g, M, N, P)
#define Chye(M, N, P)       ChyeG(g, M, N, P)

#define Hz(M, N, P)         HzG(g, M, N, P)
#define Chzh(M, N, P)       ChzhG(g, M, N, P)
#define Chze(M, N, P)       ChzeG(g, M, N, P)

#define Ex(M, N, P)         ExG(g, M, N, P)
#define Cexe(M, N, P)       CexeG(g, M, N, P)
#define Cexh(M, N, P)       CexhG(g, M, N, P)

#define Ey(M, N, P)         EyG(g, M, N, P)
#define Ceye(M, N, P)       CeyeG(g, M, N, P)
#define Ceyh(M, N, P)       CeyhG(g, M, N, P)

#define Ez(M, N, P)         EzG(g, M, N, P)
#define Ceze(M, N, P)       CezeG(g, M, N, P)
#define Cezh(M, N, P)       CezhG(g, M, N, P)

#define SizeX               SizeXG(g)
#define SizeY               SizeYG(g)
#define SizeZ               SizeZG(g)
#define Time                TimeG(g)
#define MaxTime             MaxTimeG(g)
#define Cdtds               CdtdsG(g)
#define Type                TypeG(g)

#endif  /* _FDTD_MACRO_H */
```

fdtd-proto.h

```
#ifndef _FDTD_PROTO_H
#define _FDTD_PROTO_H

#include "fdtd-grid.h"

/* Grid setup */
void gridInit(Grid *g);

/* Source injection */
void ezIncInit(Grid *g);
double ezInc(double time, double location);

/* ABC (absorbing boundary conditions) */
void abcInit(Grid *g);
void abc(Grid *g);

/* Snapshotting */
void snapshot3dInit(Grid *g);
void snapshot3d(Grid *g);

/* Field updates */
void updateE(Grid *g);
void updateH(Grid *g);

#endif  /* _FDTD_PROTO_H */
```

fdtd-grid.h

```
#ifndef _FDTD_GRID1_H
#define _FDTD_GRID1_H

enum GRIDTYPE {oneDGrid, teZGrid, tmZGrid, threeDGrid};

struct Grid {
  double *hx, *chxh, *chxe;
  double *hy, *chyh, *chye;
  double *hz, *chzh, *chze;
  double *ex, *cexe, *cexh;
  double *ey, *ceye, *ceyh;
  double *ez, *ceze, *cezh;

  int sizeX, sizeY, sizeZ;
  int time, maxTime;
  int type;
  double cdtds;

  // Added for 1D ABC
  double ezLeftOld[3], ezLeftOlder[3];
};

typedef struct Grid Grid;

#endif
```

fdtd-alloc.h

```
#ifndef _FDTD_ALLOC1_H
#define _FDTD_ALLOC1_H

#include <stdio.h>
#include <stdlib.h>

/* memory allocation macros */
#define ALLOC_1D(PNTR, NUM, TYPE)                               \
    PNTR = (TYPE *)calloc(NUM, sizeof(TYPE));                   \
    if (!PNTR) {                                                \
      perror("ALLOC_1D");                                       \
      fprintf(stderr,                                           \
          "Allocation failed for " #PNTR ".  Terminating...\n");\
      exit(-1);                                                 \
    }

#define ALLOC_2D(PNTR, NUMX, NUMY, TYPE)                            \
    PNTR = (TYPE *)calloc((NUMX) * (NUMY), sizeof(TYPE));           \
    if (!PNTR) {                                                    \
      perror("ALLOC_2D");                                           \
      fprintf(stderr,                                               \
          "Allocation failed for " #PNTR ".  Terminating...\n");    \
      exit(-1);                                                     \
    }

/* memory allocation macro for 3D grids */
#define ALLOC_3D(PNTR, NUMX, NUMY, NUMZ, TYPE)                      \
    PNTR = (TYPE *)calloc((NUMX) * (NUMY) * (NUMZ), sizeof(TYPE)); \
    if (!PNTR) {                                                    \
      perror("ALLOC_3D");                                           \
      fprintf(stderr,                                               \
          "Allocation failed for " #PNTR ".  Terminating...\n");    \
      exit(-1);                                                     \
    }

#endif
```