EE535: Numerical Solutions to EM Problems

FDTD Simulation of Tactical Body-Worn Patch Antennas

Final Project Report

Angelica Gutierrez

May 4th, 2025

Overview

This project investigates body-worn patch antennas for tactical communication systems by analyzing how proximity to human tissue and variations in ground plane size affect antenna performance. The study builds on the FDTD-based simulations from homework 9, using a microstrip-fed patch antenna placed over a dielectric slab representing human tissue. To reflect realistic body-loading conditions, the dielectric was assigned a relative permittivity of $\varepsilon r = 40$, consistent with values reported in biomedical literature. A total of five cases were simulated, varying the size of the patch's ground plane—from full to quarter coverage—while keeping other parameters constant. The analysis centers on two observables: wave propagation speed and reflection coefficient ($|S_{11}|$), the latter calculated using FFT methods inspired by Sheen et al. These metrics provide insight into how the reduction of ground plane size affects energy confinement, wave velocity, and signal reflection—key considerations for maintaining reliable communication in gear-heavy, body-proximate environments such as those faced in military operations.

Source Code Implementation

The simulation framework used in this project is based on the modular FDTD structure from homework 9. Key source files—main.c, grid3d.c, gaussian.c, abc3d.c, and snapshot3d.c—were reused with targeted modifications that allow dynamic configuration of geometry and material conditions at runtime. This modular setup streamlined the process of testing multiple scenarios without rewriting core logic.

The primary implementation change involved extending grid3d.c to support runtime toggling of the patch antenna, the inclusion of a dielectric slab, and the selection of ground plane coverage. These user-configurable options allowed each simulation to represent a different real-world mounting condition. The dielectric slab, used to model human tissue, was defined using a relative permittivity of $\varepsilon r = 40$, and the patch dimensions and ground plane widths were modified accordingly based on input. Region-aware update coefficients were then applied to accurately handle field updates in both air and dielectric regions, ensuring physical realism in the simulated wave behavior.

Additional enhancements were added to abc3d.c to implement region-dependent absorbing boundary conditions (ABCs). Specifically, boundary coefficients were chosen based on depth into the slab (pp < HEIGHT), enabling appropriate attenuation at the edges of dielectric-loaded versus free-space zones. This refinement reduced artificial reflections and allowed more realistic field termination in body-loaded environments.

The gaussian.c source code was also modified to accept runtime inputs for source delay and width, giving precise control over the excitation waveform's shape and timing across different simulation cases. Meanwhile, snapshot3d.c was set up to capture 2D Ez snapshots at a fixed z-plane (z = HEIGHT - 1), providing clear animations of lateral field propagation under each configuration.

Each of the five simulation cases corresponds to a different real-world deployment of a wearable antenna:

Case 1: Patch Only (Free Space)
Baseline test case. The patch antenna is placed in free space with no dielectric slab. This case serves as a control to observe wave propagation and antenna behavior without any body-loading or nearby materials. It mimics a patch antenna mounted on gear or clothing that is physically separated from the body.

Case 2: Patch + Full Ground Plane + Body Slab
Maximum shielding scenario. A dielectric slab lies beneath a full-width conducting ground plane and patch. This configuration represents a tactical setup where the antenna is mounted on a fully-shielded platform, such as a hard armor plate or a rigid vest, with the body underneath providing some degree of electromagnetic loading. It also serves to evaluate how much the body affects the antenna when fully shielded.

Case 3: Patch + Half Ground Plane + Body Slab
Partial shielding scenario. Similar to Case 2, but the ground plane width is reduced to half the patch's width. This setup could represent gear with incomplete shielding—like a vest with flexible flaps or a less rigid structure—allowing more interaction between the antenna and the underlying tissue.

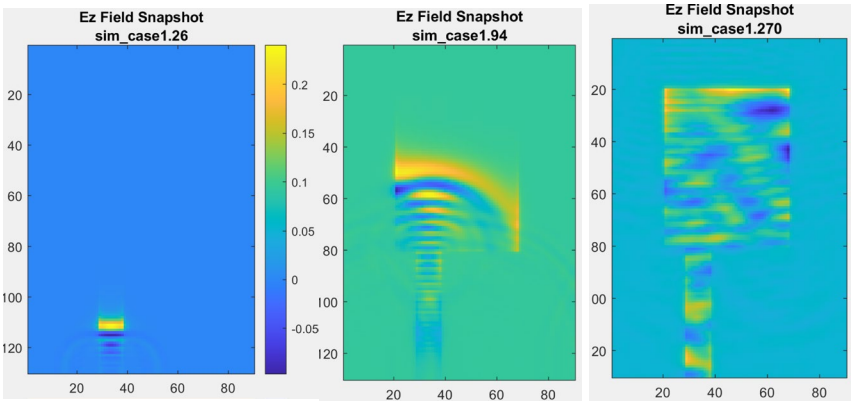
Case 4: Strip Line Only (No Patch, No Slab)
Reference case. A microstrip line without a patch or dielectric slab is simulated in free space. This case establishes a baseline "incident field" against which reflected fields in other cases can be compared. It reflects signal propagation in an idealized, unloaded environment.

Case 5: Patch + Quarter Ground Plane + Body Slab
Minimal shielding scenario. This case pushes the ground plane reduction further by limiting it to a quarter-width under the patch. It simulates an extreme tactical condition—such as a helmet-mounted or shoulder-mounted patch—where the underlying body material is significantly exposed. It's intended to probe how small the ground plane can be before the body begins to heavily influence antenna behavior.

These configurations allow the system to isolate and observe how different tactical mounting conditions—specifically the extent of shielding and presence of nearby tissue—impact electromagnetic behavior. Runtime inputs provide flexibility in testing, while shared grid dimensions and excitation parameters ensure consistency across cases.

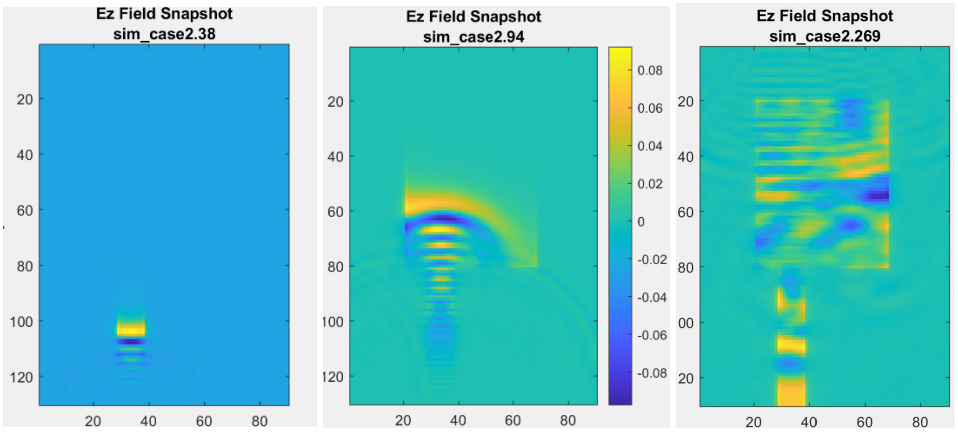
Case-by-Case Field Behavior Observations

Case 1: Patch Antenna in Free Space (No Ground, No Body)



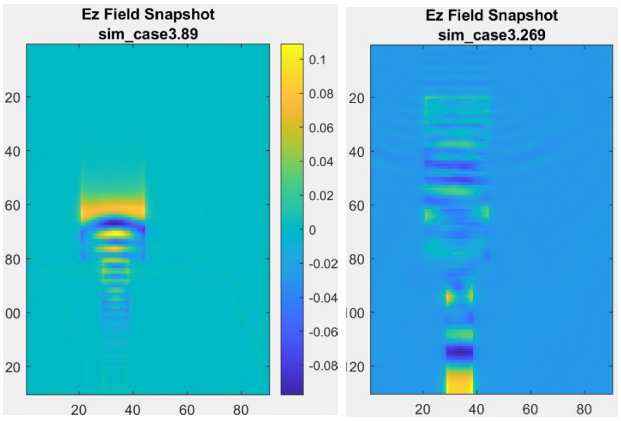
The wavefront propagates quickly and cleanly, consistent with expectations for a free-space environment. The field remains well-confined around the patch throughout the simulation, with minimal dispersion. This scenario serves as a high-speed baseline for comparison, where wave behavior is unaffected by nearby materials.

Case 2: Patch Antenna with Full Ground Plane over Dielectric (Body-Loaded)



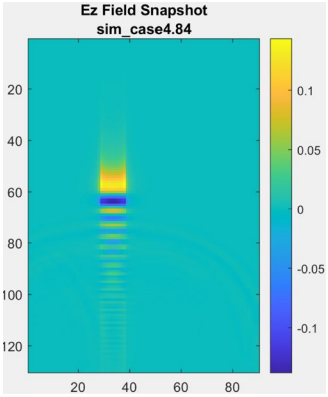
Despite the presence of a high-permittivity body slab beneath the full ground plane, the wavefront appears visually similar to Case 1—remaining tight and localized near the patch with little lateral spread. However, wave propagation is noticeably slower, consistent with increased dielectric loading, which reduces phase velocity even when fully shielded by a ground plane.

Case 3: Patch Antenna with Half Ground Plane over Dielectric (Partially Shielded)



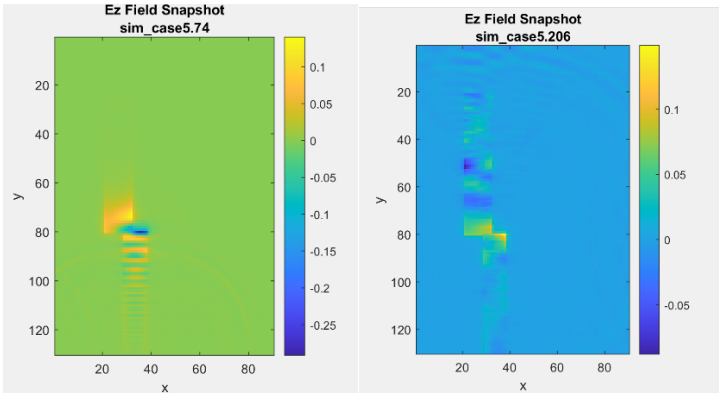
Compared to Cases 1 and 2, Case 3 reveals increased lateral leakage and less uniform confinement under the patch. The reduced ground plane permits more interaction with the underlying body slab. Despite this, the wavefront propagates faster than in Case 2, likely due to reduced reflection and impedance mismatch at the patch edges.

Case 4: Microstrip Feed Only in Free Space (No Patch, No Body)



This case provides a clean incident field in free space, with rapid, symmetric wave propagation and no significant reflections. It serves as a baseline reference.

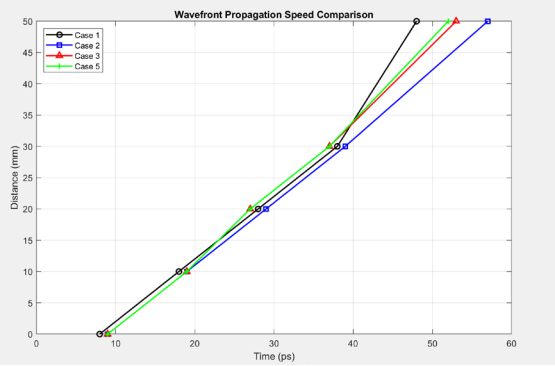
Case 5: Patch Antenna with Quarter Ground Plane over Dielectric (Minimally Shielded)



This case demonstrates significantly less field confinement due to minimal ground plane coverage. The fields appear to spread laterally and upward into the dielectric region, with less directional energy flow compared to earlier cases. This broader spread corresponds with a slower wavefront speed and increased body-loading interaction, indicating a loss in efficiency when shielding is reduced to a quarter ground plane.

Results

Wave Propagation Speed



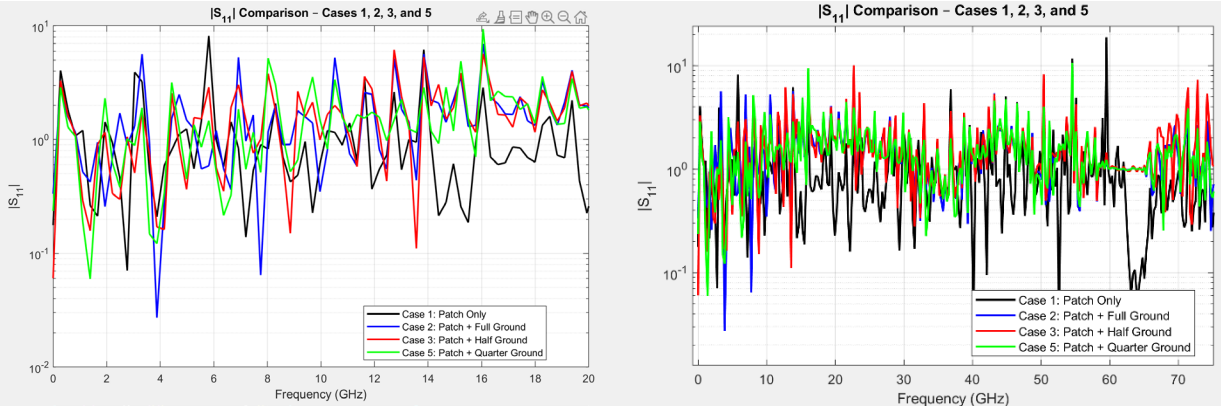
Command Window

```
>> wave_propagation_speed

Case                                          Arrival Time (ps) Speed (m/s)
-------------------------------------------------------------------------
Case 1: Patch Only                            48.00       1.200e+06
Case 2: Patch + Full Ground + Body            57.00       1.039e+06
Case 3: Patch + Half Ground + Body            53.00       1.136e+06
Case 5: Patch + Quarter Ground + Body         52.00       1.160e+06
```

To quantify the propagation speed differences initially observed in the field snapshots, I plotted wavefront travel distance over time and computed arrival times numerically. These results confirmed the visual cues: Case 1 (Patch Only) exhibited the fastest propagation, reaching the 50 mm mark in 48 ps, corresponding to a wave speed of 1.200×10^6 m/s. In contrast, Case 2 (Patch + Full Ground + Body) showed the slowest propagation, taking 57 ps to reach the same distance, with a reduced speed of 1.039×10^6 m/s—highlighting the impedance introduced by the dielectric slab. Case 3 (Half Ground) and Case 5 (Quarter Ground) demonstrated intermediate speeds of 1.136×10^6 m/s and 1.160×10^6 m/s, respectively. These trends validate the expectation that wave propagation slows as electromagnetic energy interacts more with body-loading materials and suggest that reducing the ground plane can modestly mitigate this slowdown.

Reflection Coefficient ($|S_{11}|$) Comparison and Interpretation



The comparison of |S11| across Cases 1, 2, 3, and 5 reveals notable differences in how energy is reflected back from the antenna structure—closely tied to how much of the wave is absorbed or radiated away. Across the full frequency range, Case 1 (Patch Only in Free Space) consistently exhibits lower reflection magnitudes, suggesting stronger radiation and better matching. However, there are a few localized spikes—for instance, near 6 GHz—where |S11| briefly exceeds other cases, indicating frequency-sensitive mismatches. Case 2 (Patch + Full Ground + Body) shows high reflection in the lower frequency band but becomes more stable as frequency increases, implying that full body shielding causes early mismatches that dissipate at higher frequencies. Meanwhile, Case 3 (Half Ground) and Case 5 (Quarter Ground) swap prominence depending on frequency scale: in the zoomed-in view (up to 20 GHz), the quarter-ground configuration (Case 5) appears slightly more reflective than Case 3, while in the full-bandwidth view (up to 75 GHz), Case 3 (Half Ground) becomes more reflective than Case 5. This reversal may be due to how different ground plane sizes interact with specific wavelength ranges. Notably, the quarter-ground case consistently maintains the lowest reflection among the body-loaded setups, suggesting some degree of beneficial coupling or cancellation effect at play—an important consideration for wearable antennas where full shielding isn't always practical.

Conclusion

This project served as a phase 1 investigation into how ground plane coverage and body proximity influence the performance of body-worn patch antennas in tactical settings. Through a series of FDTD simulations, we observed that both wave propagation speed and reflection characteristics ($|S_{11}|$) are significantly affected by the presence of a dielectric slab mimicking human tissue and the extent of ground plane shielding. These findings correlate directly with real-world scenarios in which wearable antennas interact closely with the human body and surrounding gear.

As a natural next step, future work could focus on scaling the geometry to represent actual anatomical features—such as a full-scale human shoulder or head—and incorporating frequency bands of operational interest. Rather than using a uniform slab, more anatomically realistic models could be introduced to better reflect tissue heterogeneity. Additional refinements might include using a lossy dielectric model for human tissue, experimenting with different feed placements, or exploring time-gated or impedance-matched excitation methods. Overall, this study lays a practical foundation for developing reliable, body-integrated antenna systems and opens the door for more in-depth, mission-specific antenna design in the next phase of exploration.

References

- [1] D. A. Sheen, S. M. Ali, M. D. Abouzahra, and D. McMakin, “Three-dimensional finite-difference time-domain simulation of the interaction of electromagnetic waves with a dielectric body,” IEEE Transactions on Microwave Theory and Techniques, vol. 38, no. 7, pp. 849–857, Jul. 1990.
- [2] D. Naranjo-Hernández, A. Callejón-Leblic, C. Lučev Vasić, M. Seyedi, and Y.-M. Gao, “Wearable antennas: A review of materials, structures, and challenges,” Sensors, vol. 21, no. 21, pp. 1–25, Oct. 2021. [Online]. Available: https://www.mdpi.com/1424-8220/21/21/6530
- [3] European Antennas Ltd., “Past results, present trends and future challenges in wearable antennas,” European-Antennas.co.uk, [Online]. Available: https://www.european-antennas.co.uk/

Source Files

main.c

```c
#include "fdtd-alloc.h"
#include "fdtd-macro.h"
#include "fdtd-proto.h"
#include "geometry.h"

int main()
{
  int pp;
  double voltage;
  FILE *out;

  Grid *g;

  ALLOC_1D(g, 1, Grid); // allocate memory for grid structure
  gridInit(g);        // initialize 3D grid

  abcInit(g);         // initialize ABC
  snapshotInit3d(g);  // initialize snapshots

  out = fopen("obs", "w");

  /* do time stepping */
  for (Time = 0; Time < MaxTime; Time++) {
    printf("working on time step %d\n", Time);
    updateH(g);     // update magnetic fields
    updateE(g);     // update electric fields
    abc(g);         // apply ABC
    snapshot3d(g);  // snapshot
    /* record the voltage */
    voltage = 0.0;
    for (pp = 0; pp < HEIGHT; pp++)
      voltage += Ez(LINE_X_START + LINE_WIDTH / 2, PATCH_Y_START - 10, pp);
    fprintf(out,"%g\n", voltage);
  } // end of time-stepping

  return 0;
}
```

gaussian.c

```c
#include "ezinc.h"

static double width = 0, delay;

/* initialize source-function variables */
void ezIncInit(Grid *g){

  printf("Enter width of Gaussian: ");
  scanf(" %lf", &width);
  printf("Enter delay: ");
  scanf(" %lf", &delay);
  return;
}

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

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

  arg = (time - delay) / width;
  if (arg > 30.) {
    return 0.0;
  } else {
    return exp(-arg * arg);
  }
}
```

ezinc.h

```
#ifndef _EZINC_H
#define _EZINC_H

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

void ezIncInit(Grid *g);
double ezInc(double time);

#endif
```

geometry.h

```
#ifndef _GEOMETRY_H
#define _GEOMETRY_H

// ====================
// Material Properties
// ====================
#define EPSR       40.0          // Duroid permittivity 40.0 for real skin
#define EPSR_BODY  50.0          // Body slab permittivity
#define SLAB_THICKNESS 5         // Number of z-layers for slab

// ====================
// Source Control
// ====================
#define SWITCH_SRC 225           // Switch source to ABC at this time step

// ====================
// Vertical Structure
// ====================
#define HEIGHT 3                 // Duroid thickness (z=0 to HEIGHT-1)

// ====================
// Microstrip Feed Line
// ====================
#define LINE_X_START 28
#define LINE_WIDTH    9

// ====================
// Patch Antenna Geometry
// ====================
#define PATCH_X_START  20
#define PATCH_X_WIDTH  47
#define PATCH_Y_START  50
#define PATCH_Y_LENGTH 60

#endif
```

grid3d.c

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

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

  int patchPresent, slabPresent, groundPlaneMode;
  int patchWidth;

  // Runtime prompts
  printf("Is the patch present? (1=yes, 0=no) ");
  scanf(" %d", &patchPresent);

  printf("Include dielectric slab? (1=yes, 0=no) ");
  scanf(" %d", &slabPresent);

  printf("Ground plane mode (1=full, 2=half, 3=quarter)? ");
  scanf(" %d", &groundPlaneMode);

  // Set up grid properties
  Type = threeDGrid;
  SizeX = 90;
  SizeY = 130;
  SizeZ = 20;
  MaxTime = 8192;
  Cdtds = 1.0 / sqrt(3.1);

  // 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);

  // Initialize electric field update coefficients
  for (mm = 0; mm < SizeX - 1; mm++)
    for (nn = 0; nn < SizeY; nn++)
      for (pp = 1; pp < SizeZ; pp++) {
        Cexe(mm, nn, pp) = 1.0;
        if (pp < HEIGHT)
          Cexh(mm, nn, pp) = Cdtds * imp0 / EPSR;
        else if (pp == HEIGHT)
          Cexh(mm, nn, pp) = Cdtds * imp0 / ((EPSR + 1.0) / 2.0);
        else
          Cexh(mm, nn, pp) = Cdtds * imp0;
      }

  for (mm = 0; mm < SizeX; mm++)
    for (nn = 0; nn < SizeY - 1; nn++)
      for (pp = 1; pp < SizeZ; pp++) {
        Ceye(mm, nn, pp) = 1.0;
        if (pp < HEIGHT)
          Ceyh(mm, nn, pp) = Cdtds * imp0 / EPSR;
        else if (pp == HEIGHT)
          Ceyh(mm, nn, pp) = Cdtds * imp0 / ((EPSR + 1.0) / 2.0);
        else
          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;
        if (pp < HEIGHT)
          Cezh(mm, nn, pp) = Cdtds * imp0 / EPSR;
        else
          Cezh(mm, nn, pp) = Cdtds * imp0;
      }
```

```
  // Optional: add dielectric slab for body-loading
  if (slabPresent) {
    for (mm = 0; mm < SizeX; mm++)
      for (nn = 0; nn < SizeY; nn++)
        for (pp = 0; pp < SLAB_THICKNESS; pp++) {
          Ceze(mm, nn, pp) = 1.0;
          Cezh(mm, nn, pp) = Cdtds * imp0 / EPSR_BODY;
        }
  }

  // Determine patch width based on selected ground plane mode
  patchWidth = PATCH_X_WIDTH;
  if (groundPlaneMode == 2) patchWidth /= 2;
  else if (groundPlaneMode == 3) patchWidth /= 4;

  // Insert microstrip and patch
  pp = HEIGHT;

  if (patchPresent) {
    // Microstrip line
    for (mm = LINE_X_START; mm < LINE_X_START + LINE_WIDTH; mm++)
      for (nn = 0; nn < PATCH_Y_START; nn++) {
        Cexe(mm, nn, pp) = 0.0;
        Cexh(mm, nn, pp) = 0.0;
      }
    for (mm = LINE_X_START; mm <= LINE_X_START + LINE_WIDTH; mm++)
      for (nn = 0; nn < PATCH_Y_START; nn++) {
        Ceye(mm, nn, pp) = 0.0;
        Ceyh(mm, nn, pp) = 0.0;
      }

    // Patch (reduced if needed)
    for (mm = PATCH_X_START; mm < PATCH_X_START + patchWidth; mm++)
      for (nn = PATCH_Y_START; nn <= PATCH_Y_START + PATCH_Y_LENGTH; nn++) {
        Cexe(mm, nn, pp) = 0.0;
        Cexh(mm, nn, pp) = 0.0;
      }
    for (mm = PATCH_X_START; mm <= PATCH_X_START + patchWidth; mm++)
      for (nn = PATCH_Y_START; nn < PATCH_Y_START + PATCH_Y_LENGTH; nn++) {
        Ceye(mm, nn, pp) = 0.0;
        Ceyh(mm, nn, pp) = 0.0;
      }

  } else {
    // Microstrip only
    for (mm = LINE_X_START; mm < LINE_X_START + LINE_WIDTH; mm++)
      for (nn = 0; nn < SizeY; nn++) {
        Cexe(mm, nn, pp) = 0.0;
        Cexh(mm, nn, pp) = 0.0;
      }
    for (mm = LINE_X_START; mm <= LINE_X_START + LINE_WIDTH; mm++)
      for (nn = 0; nn < SizeY - 1; nn++) {
        Ceye(mm, nn, pp) = 0.0;
        Ceyh(mm, nn, pp) = 0.0;
      }
  }

  // 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;
}
```

update3d.c

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

/* update magnetic field */
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;
}  /* end updateH() */
```

```c
/* update electric field */
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;
}  /* end updateE() */
```

abc3d.c

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

/* Macros to access stored "old" value */
#define Eyx0(N, P) eyx0[(N) * (SizeZ)+(P)]
#define Ezx0(N, P) ezx0[(N) * (SizeZ - 1)+(P)]
#define Eyx1(N, P) eyx1[(N) * (SizeZ)+(P)]
#define Ezx1(N, P) ezx1[(N) * (SizeZ - 1)+(P)]

#define Exy0(M, P) exy0[(M) * (SizeZ)+(P)]
#define Ezy0(M, P) ezy0[(M) * (SizeZ - 1)+(P)]
#define Exy1(M, P) exy1[(M) * (SizeZ)+(P)]
#define Ezy1(M, P) ezy1[(M) * (SizeZ - 1)+(P)]

#define Exz1(M, N) exz1[(M) * (SizeY)+(N)]
#define Eyz1(M, N) eyz1[(M) * (SizeY - 1)+(N)]

/* global variables not visible outside of this package */
static double abcDie = 0.0, abcAvg, abcAir;
static double
  *eyx0, *ezx0, *eyx1, *ezx1, *exy0, *ezy0,
  *exy1, *ezy1, *exz1, *eyz1;

/* initialization function */
void abcInit(Grid *g)
{
  double epsr = EPSR;

  ezIncInit(g); // initialize source function

  abcAir = (Cdtds - 1.0) / (Cdtds + 1.0);
  abcDie = (Cdtds / sqrt(epsr) - 1.0) /
    (Cdtds/sqrt(epsr) + 1.0);
  abcAvg = (Cdtds/sqrt((epsr + 1.0) / 2.0) - 1.0)/
    (Cdtds/sqrt((epsr + 1.0) / 2.0) + 1.0);

  /* allocate memory for ABC arrays */
  ALLOC_2D(eyx0, SizeY - 1, SizeZ, double);
  ALLOC_2D(ezx0, SizeY, SizeZ - 1, double);
  ALLOC_2D(eyx1, SizeY - 1, SizeZ, double);
  ALLOC_2D(ezx1, SizeY, SizeZ - 1, double);

  ALLOC_2D(exy0, SizeX - 1, SizeZ, double);
  ALLOC_2D(ezy0, SizeX, SizeZ - 1, double);
  ALLOC_2D(exy1, SizeX - 1, SizeZ, double);
  ALLOC_2D(ezy1, SizeX, SizeZ - 1, double);

  ALLOC_2D(exz1, SizeX - 1, SizeY, double);
  ALLOC_2D(eyz1, SizeX, SizeY - 1, double);

  return;
} /* end abcInit() */

/* function that applies ABC -- called once per time step */
void abc(Grid *g)
{
  int mm, nn, pp;
  double tempEz;

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

```
  /* ABC at "x0" */
  mm = 0;
  for (nn = 0; nn < SizeY - 1; nn++)
    for (pp = 0; pp < SizeZ; pp++) {
      if (pp < HEIGHT)
    Ey(mm, nn, pp) = Eyx0(nn, pp) +
      abcDie * (Ey(mm + 1, nn, pp) - Ey(mm, nn, pp));
      else if (pp == HEIGHT)
    Ey(mm, nn, pp) = Eyx0(nn, pp) +
      abcAvg * (Ey(mm + 1, nn, pp) - Ey(mm, nn, pp));
      else
    Ey(mm, nn, pp) = Eyx0(nn, pp) +
      abcAir * (Ey(mm + 1, nn, pp) - Ey(mm, nn, pp));
      Eyx0(nn, pp) = Ey(mm + 1, nn, pp);
    }
  for (nn = 0; nn < SizeY; nn++)
    for (pp = 0; pp < SizeZ - 1; pp++) {
      if (pp < HEIGHT)
    Ez(mm, nn, pp) = Ezx0(nn, pp) +
      abcDie * (Ez(mm + 1, nn, pp) - Ez(mm, nn, pp));
      else
    Ez(mm, nn, pp) = Ezx0(nn, pp) +
      abcAir * (Ez(mm + 1, nn, pp) - Ez(mm, nn, pp));
      Ezx0(nn, pp) = Ez(mm + 1, nn, pp);
    }

  /* ABC at "x1" */
  mm = SizeX - 1;
  for (nn = 0; nn < SizeY - 1; nn++)
    for (pp = 0; pp < SizeZ; pp++) {
      if (pp < HEIGHT)
    Ey(mm, nn, pp) = Eyx1(nn, pp) +
      abcDie * (Ey(mm - 1, nn, pp) - Ey(mm, nn, pp));
      else if (pp == HEIGHT)
    Ey(mm, nn, pp) = Eyx1(nn, pp) +
      abcAvg * (Ey(mm - 1, nn, pp) - Ey(mm, nn, pp));
      else
    Ey(mm, nn, pp) = Eyx1(nn, pp) +
      abcAir * (Ey(mm - 1, nn, pp) - Ey(mm, nn, pp));
      Eyx1(nn, pp) = Ey(mm - 1, nn, pp);
    }
  for (nn = 0; nn < SizeY; nn++)
    for (pp = 0; pp < SizeZ - 1; pp++) {
      if (pp < HEIGHT)
    Ez(mm, nn, pp) = Ezx1(nn, pp) +
      abcDie * (Ez(mm - 1, nn, pp) - Ez(mm, nn, pp));
      else
    Ez(mm, nn, pp) = Ezx1(nn, pp) +
      abcAir * (Ez(mm - 1, nn, pp) - Ez(mm, nn, pp));
      Ezx1(nn, pp) = Ez(mm - 1, nn, pp);
    }

  if (Time < SWITCH_SRC) {
    /* Ex and Ez nodes on "y0" wall have special updates while the
       wall is acting as the source plane */
    nn = 0;
    for (mm = 0; mm < SizeX - 1; mm++)
      for (pp = 1; pp < SizeZ - 1; pp++)
    Ex(mm, nn, pp) = Cexe(mm, nn, pp) * Ex(mm, nn, pp) +
      Cexh(mm, nn, pp) * (2.0 * Hz(mm, nn, pp) -
                          (Hy(mm, nn, pp) - Hy(mm, nn, pp - 1)));
    for (mm = 1; mm < SizeX - 1; mm++)
      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)) -
                          2.0 * Hx(mm, nn, pp));

    /* source under microstrip at "y0" plane */
    tempEz = ezInc(Time) / 3.0;
    for (mm = LINE_X_START; mm <= LINE_X_START + LINE_WIDTH; mm++)
      for (pp = 0; pp < HEIGHT; pp++)
    Ez(mm, nn, pp) = tempEz;

  } else {
```

```
  } else {
    /* ABC at "y0" after source has been turned off */
    nn = 0;
    for (mm = 0; mm < SizeX - 1; mm++)
      for (pp = 0; pp < SizeZ; pp++) {
    if (pp < HEIGHT)
      Ex(mm, nn, pp) = Exy0(mm, pp) +
        abcDie * (Ex(mm, nn + 1, pp) - Ex(mm, nn, pp));
    else if (pp == HEIGHT)
      Ex(mm, nn, pp) = Exy0(mm, pp) +
        abcAvg * (Ex(mm, nn + 1, pp) - Ex(mm, nn, pp));
    else
      Ex(mm, nn, pp) = Exy0(mm, pp) +
        abcAir * (Ex(mm, nn + 1, pp) - Ex(mm, nn, pp));
    Exy0(mm, pp) = Ex(mm, nn + 1, pp);
      }
    for (mm = 0; mm < SizeX; mm++)
      for (pp = 0; pp < SizeZ - 1; pp++) {
    if (pp < HEIGHT)
      Ez(mm, nn, pp) = Ezy0(mm, pp) +
        abcDie * (Ez(mm, nn + 1, pp) - Ez(mm, nn, pp));
    else
      Ez(mm, nn, pp) = Ezy0(mm, pp) +
        abcAir * (Ez(mm, nn + 1, pp) - Ez(mm, nn, pp));
    Ezy0(mm, pp) = Ez(mm, nn + 1, pp);
      }
  }

  /* ABC at "y1" */
  nn = SizeY - 1;
  for (mm = 0; mm < SizeX - 1; mm++)
    for (pp = 0; pp < SizeZ; pp++) {
      if (pp < HEIGHT)
    Ex(mm, nn, pp) = Exy1(mm, pp) +
      abcDie * (Ex(mm, nn - 1, pp) - Ex(mm, nn, pp));
      else if (pp == HEIGHT)
    Ex(mm, nn, pp) = Exy1(mm, pp) +
      abcAvg * (Ex(mm, nn - 1, pp) - Ex(mm, nn, pp));
      else
    Ex(mm, nn, pp) = Exy1(mm, pp) +
      abcAir * (Ex(mm, nn - 1, pp) - Ex(mm, nn, pp));
      Exy1(mm, pp) = Ex(mm, nn - 1, pp);
    }
  for (mm = 0; mm < SizeX; mm++)
    for (pp = 0; pp < SizeZ - 1; pp++) {
      if (pp < HEIGHT)
    Ez(mm, nn, pp) = Ezy1(mm, pp) +
      abcDie * (Ez(mm, nn - 1, pp) - Ez(mm, nn, pp));
      else
    Ez(mm, nn, pp) = Ezy1(mm, pp) +
      abcAir * (Ez(mm, nn - 1, pp) - Ez(mm, nn, pp));
      Ezy1(mm, pp) = Ez(mm, nn - 1, pp);
    }

  /* ABC at "z1" (top) */
  pp = SizeZ - 1;
  for (mm = 0; mm < SizeX - 1; mm++)
    for (nn = 0; nn < SizeY; nn++) {
      Ex(mm, nn, pp) = Exz1(mm, nn) +
    abcAir * (Ex(mm, nn, pp - 1) - Ex(mm, nn, pp));
      Exz1(mm, nn) = Ex(mm, nn, pp - 1);
    }
  for (mm = 0; mm < SizeX; mm++)
    for (nn = 0; nn < SizeY - 1; nn++) {
      Ey(mm, nn, pp) = Eyz1(mm, nn) +
    abcAir * (Ey(mm, nn, pp - 1) - Ey(mm, nn, pp));
      Eyz1(mm, nn) = Ey(mm, nn, pp - 1);
    }

  return;
} /* end abc() */
```

snapshot3d.c

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

static int temporalStride = -2, frame = 0, startTime;
static char basename[80];
static char folder[80];

void snapshotInit3d(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 folder name for saving snapshots (e.g., snapshots_case4): ");
  scanf(" %79s", folder);

  printf("Enter base name for snapshot files (e.g., sim_case4): ");
  scanf(" %79s", basename);

  // Create folder if it doesn't exist
  char mkdirCmd[160];
  snprintf(mkdirCmd, sizeof(mkdirCmd), "mkdir -p %s", folder);
  system(mkdirCmd);

  return;
}

void snapshot3d(Grid *g) {
  int mm, nn, pp;
  float dim1, dim2, temp;
  char filename[200];
  FILE *out;

  if (temporalStride == -1) return;

  if (temporalStride < -1) {
    fprintf(stderr,
      "snapshot3d: snapshotInit3d 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) {

    // Format: folder/basename.frame#
    snprintf(filename, sizeof(filename), "%s/%s.%d", folder, basename, frame++);
    out = fopen(filename, "wb");

    if (!out) {
      fprintf(stderr, "Error: Could not open file %s for writing.\n", filename);
      exit(-1);
    }

    // Write dimensions as floats
    dim1 = SizeX;
    dim2 = SizeY;
    fwrite(&dim1, sizeof(float), 1, out);
    fwrite(&dim2, sizeof(float), 1, out);

    // Write Ez(x,y) field slice at z = HEIGHT - 1
    pp = HEIGHT - 1;
    for (nn = SizeY - 1; nn >= 0; nn--)
      for (mm = 0; mm < SizeX; mm++) {
        temp = (float)Ez(mm, nn, pp);
        fwrite(&temp, sizeof(float), 1, out);
      }

    fclose(out);
  }

  return;
}
```

fdtd-alloc.h

```
#ifndef _FDTD_ALLOC_H
#define _FDTD_ALLOC_H

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

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

#define ALLOC_2D(name,nx,ny,type)                               \
    name = (type *)calloc((nx)*(ny),sizeof(type));              \
    if (!name) {                                                \
      perror("ALLOC_2D");                                       \
      fprintf(stderr,                                           \
              "Allocation failed for " #name ".  Terminating...\n");\
      exit(-1);                                                 \
    }

#define ALLOC_3D(name,nx,ny,nz,type)                            \
    name = (type *)calloc((nx)*(ny)*(nz),sizeof(type));         \
    if (!name) {                                                \
      perror("ALLOC_2D");                                       \
      fprintf(stderr,                                           \
              "Allocation failed for " #name ".  Terminating...\n");\
      exit(-1);                                                 \
    }

#endif
```

fdtd-grid.h

```
#ifndef _FDTD_GRID_H
#define _FDTD_GRID_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;
};

typedef struct Grid Grid;

#endif
```

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,MM)     G->hy[MM]
#define Chyh1G(G,MM)   G->chyh[MM]
#define Chye1G(G,MM)   G->chye[MM]

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

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

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

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

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

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

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

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

#define HyG(G,MM,NN,PP)   G->hy[((MM)*SizeYG(G) + NN)*(SizeZG(G)-1) + PP]
#define ChyhG(G,MM,NN,PP) G->chyh[((MM)*SizeYG(G) + NN)*(SizeZG(G)-1) + PP]
#define ChyeG(G,MM,NN,PP) G->chye[((MM)*SizeYG(G) + NN)*(SizeZG(G)-1) + PP]

#define HzG(G,MM,NN,PP)   G->hz[((MM)*(SizeYG(G)-1) + NN)*SizeZG(G) + PP]
#define ChzhG(G,MM,NN,PP) G->chzh[((MM)*(SizeYG(G)-1) + NN)*SizeZG(G) + PP]
#define ChzeG(G,MM,NN,PP) G->chze[((MM)*(SizeYG(G)-1) + NN)*SizeZG(G) + PP]

#define ExG(G,MM,NN,PP)   G->ex[((MM)*SizeYG(G) + NN)*SizeZG(G) + PP]
#define CexeG(G,MM,NN,PP) G->cexe[((MM)*SizeYG(G) + NN)*SizeZG(G) + PP]
#define CexhG(G,MM,NN,PP) G->cexh[((MM)*SizeYG(G) + NN)*SizeZG(G) + PP]

#define EyG(G,MM,NN,PP)   G->ey[((MM)*(SizeYG(G)-1) + NN)*SizeZG(G) + PP]
#define CeyeG(G,MM,NN,PP) G->ceye[((MM)*(SizeYG(G)-1) + NN)*SizeZG(G) + PP]
#define CeyhG(G,MM,NN,PP) G->ceyh[((MM)*(SizeYG(G)-1) + NN)*SizeZG(G) + PP]

#define EzG(G,MM,NN,PP)   G->ez[((MM)*SizeYG(G) + NN)*(SizeZG(G)-1) + PP]
#define CezeG(G,MM,NN,PP) G->ceze[((MM)*SizeYG(G) + NN)*(SizeZG(G)-1) + PP]
#define CezhG(G,MM,NN,PP) G->cezh[((MM)*SizeYG(G) + NN)*(SizeZG(G)-1) + PP]

#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" */
/* one-dimensional grid */
#define Hy1(MM)     Hy1G(g,MM)
#define Chyh1(MM)   Chyh1G(g,MM)
#define Chye1(MM)   Chye1G(g,MM)

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

```
/* TMz grid */
#define Hx2(MM,NN)   Hx2G(g,MM,NN)
#define Chxh2(MM,NN) Chxh2G(g,MM,NN)
#define Chxe2(MM,NN) Chxe2G(g,MM,NN)

#define Hy2(MM,NN)   Hy2G(g,MM,NN)
#define Chyh2(MM,NN) Chyh2G(g,MM,NN)
#define Chye2(MM,NN) Chye2G(g,MM,NN)

#define Ez2(MM,NN)   Ez2G(g,MM,NN)
#define Ceze2(MM,NN) Ceze2G(g,MM,NN)
#define Cezh2(MM,NN) Cezh2G(g,MM,NN)

/* TEz grid */
#define Hz2(MM,NN)   Hz2G(g,MM,NN)
#define Chzh2(MM,NN) Chzh2G(g,MM,NN)
#define Chze2(MM,NN) Chze2G(g,MM,NN)

#define Ex2(MM,NN)   Ex2G(g,MM,NN)
#define Cexe2(MM,NN) Cexe2G(g,MM,NN)
#define Cexh2(MM,NN) Cexh2G(g,MM,NN)

#define Ey2(MM,NN)   Ey2G(g,MM,NN)
#define Ceye2(MM,NN) Ceye2G(g,MM,NN)
#define Ceyh2(MM,NN) Ceyh2G(g,MM,NN)

/* 3D grid */
#define Hx(MM,NN,PP)   HxG(g,MM,NN,PP)
#define Chxh(MM,NN,PP) ChxhG(g,MM,NN,PP)
#define Chxe(MM,NN,PP) ChxeG(g,MM,NN,PP)

#define Hy(MM,NN,PP)   HyG(g,MM,NN,PP)
#define Chyh(MM,NN,PP) ChyhG(g,MM,NN,PP)
#define Chye(MM,NN,PP) ChyeG(g,MM,NN,PP)

#define Hz(MM,NN,PP)   HzG(g,MM,NN,PP)
#define Chzh(MM,NN,PP) ChzhG(g,MM,NN,PP)
#define Chze(MM,NN,PP) ChzeG(g,MM,NN,PP)

#define Ex(MM,NN,PP)   ExG(g,MM,NN,PP)
#define Cexe(MM,NN,PP) CexeG(g,MM,NN,PP)
#define Cexh(MM,NN,PP) CexhG(g,MM,NN,PP)

#define Ey(MM,NN,PP)   EyG(g,MM,NN,PP)
#define Ceye(MM,NN,PP) CeyeG(g,MM,NN,PP)
#define Ceyh(MM,NN,PP) CeyhG(g,MM,NN,PP)

#define Ez(MM,NN,PP)   EzG(g,MM,NN,PP)
#define Ceze(MM,NN,PP) CezeG(g,MM,NN,PP)
#define Cezh(MM,NN,PP) CezhG(g,MM,NN,PP)

#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-proto.h

```
#ifndef _FDTD_PROTO_H
#define _FDTD_PROTO_H

#include "fdtd-grid.h"

/* Function prototypes */
void abcInit(Grid *g);
void abc(Grid *g);

void gridInit(Grid *g);

void snapshotInit3d(Grid *g
void snapshot3d(Grid *g);

void updateE(Grid *g);
void updateH(Grid *g);

#endif
```