# EE535: Numerical Solutions to EM Problems

**Homework 7**

**Name: Angelica Gutierrez**

**Due: 29 March 2025**

## Overview –

*In this assignment, I implemented a 2D Finite-Difference Time-Domain (FDTD) simulation of Young's double-slit experiment using a Total Field / Scattered Field (TFSF) boundary and second-order absorbing boundary conditions (ABCs). A harmonic wave source was injected into the simulation, and the resulting diffraction pattern was observed as it passed through two narrow slits in a PEC (perfect electric conductor) wall.*

## Source Code Implementation –

I based my implementation on the modular code framework from HW5 and used HW6 as a steppingstone to understand ABCs and TFSF boundaries in more detail. I kept the same modular structure as my last few assignments (your approach Prof. Schneider), which really helped me stay organized and isolate each part of the simulation.

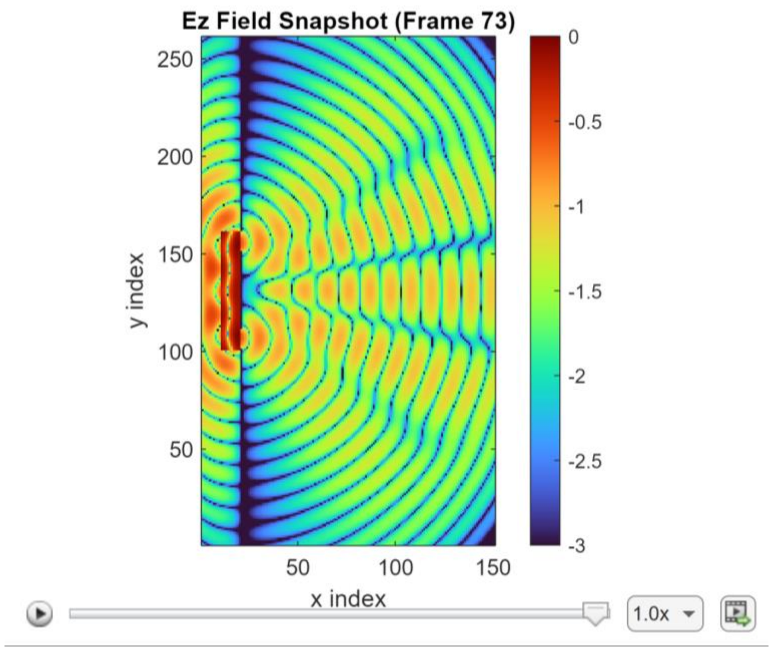
Most of my updates were made in four main files: gridtmzpec.c (to define the grid and slits), tfsftmz.c (to handle the TFSF interface), harmonic.c (which I created to define the source function), and snapshot2d.c (to record 2D field data). I also made small adjustments in fdtd-proto.h and fdtd-macro-tmz.h to support the new components. The main simulation loop lives in hw7main.c, where I initialized everything and stepped through the fields in time.

## Supporting Files & Results –

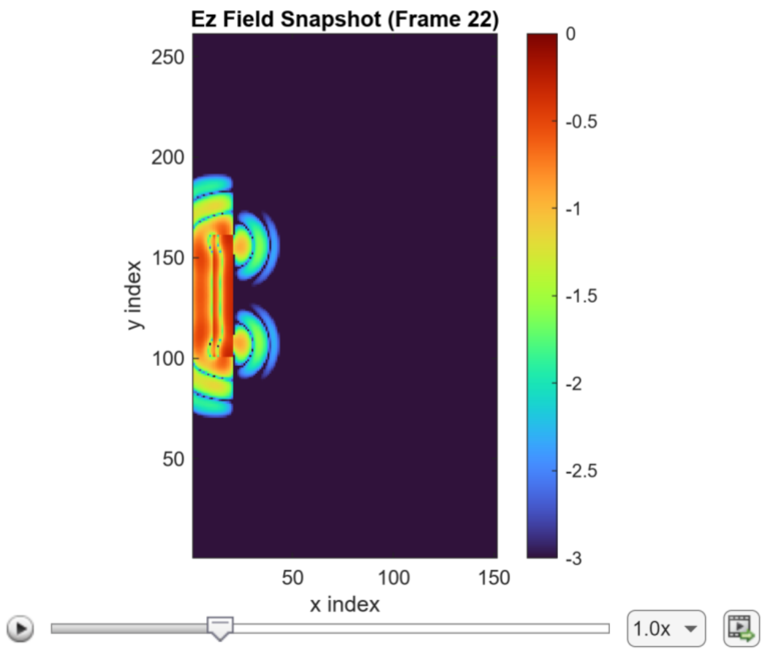
After running the simulation for 500 time steps, I visualized the evolution of the electric field (Ez) using MATLAB. The snapshot shown below is from the final frame (Frame 73), where the interference pattern is fully developed. The PEC wall appears as a vertical black region at x = 20, and the two slits are clearly visible between y = 100 −109 and y = 150−159, where the wave is allowed to pass through.

From this frame, it's clear how the harmonic wave propagates through the slits and forms a well-defined interference pattern on the right-hand side. The bright and dark bands represent constructive and destructive interference respectively—just like in the classical double-slit experiment. The symmetry and clarity of the pattern confirm that the PEC screen and slits were placed correctly, and that the TFSF boundary is injecting the source properly.

This result matches the expected behavior of TMz waves diffracting through narrow apertures and shows that the absorbing boundary conditions (ABCs) are working effectively since there are no visible reflections from the simulation domain edges.



*Shown above: Final Ez field snapshot (Frame 73) showing fully developed interference pattern from two-slit diffraction. The wavefronts emerging from the two slits in the PEC wall at x = 20 interfere constructively and destructively, forming a characteristic fringe pattern. The simulation captures clear diffraction and interference effects, with the PEC barrier visibly blocking wave propagation except through the slits.*



*Shown above: Early-stage Ez field snapshot (Frame 22) showing wavefront emergence from the two slits in the PEC wall at x = 20. The wave begins to diffract through the slits, forming circular wavefronts on the transmission side, with minimal field propagation beyond the PEC wall at this time step. The slits are clearly visible as localized breaks in the high-reflection PEC boundary.*

## MATLAB Code

### EE535_HW7_V1_PLOT.m

```matlab
% Number of snapshot files
num_frames = 73;

% File name components
file_prefix = "sim.";
file_suffix = "";

% Read first file to get dimensions
first_file = file_prefix + "0" + file_suffix;
fid = fopen(first_file, 'r');
if fid == -1
    error("Failed to open file: %s", first_file);
end

% Read dimensions
Nx = fscanf(fid, "%d", 1);
Ny = fscanf(fid, "%d", 1);

% Read field values from first snapshot
ez = fscanf(fid, "%f");
fclose(fid);

% Preallocate 3D matrix: [Ny, Nx, num_frames]
Z = zeros(Ny, Nx, num_frames);

% Store first frame (reshaped and transposed)
Z(:, :, 1) = reshape(ez, [Nx, Ny])';

% Read the rest of the snapshot files
for i = 1:num_frames-1
    filename = file_prefix + num2str(i) + file_suffix;
    fid = fopen(filename, 'r');
    if fid == -1
        warning("Could not open file %s – skipping", filename);
        continue;
    end
    fscanf(fid, "%d", 2);        % Skip Nx, Ny headers
    ez = fscanf(fid, "%f");      % Read field values
    fclose(fid);
    Z(:, :, i+1) = reshape(ez, [Nx, Ny])';
end

% Normalize the data
zmax = max(Z, [], 'all');
decades = 3;
epsilon = 1e-6;

% Animate the Ez field snapshots
for i = 1:num_frames
    % Log-scaled snapshot with small epsilon
    Ez_log = log10(abs(Z(:, :, i)) / zmax + epsilon);
    imagesc(Ez_log, [-decades, 0]);
    set(gca, 'YDir', 'normal');  % Y-axis with 0 at bottom

    % Plot settings
    colormap turbo;
    colorbar;
    title(sprintf("Ez Field Snapshot (Frame %d)", i), 'FontWeight', 'bold');
    xlabel("x index");
    ylabel("y index");
    daspect([1 1 1]);
    drawnow;
end
```

## C Codes

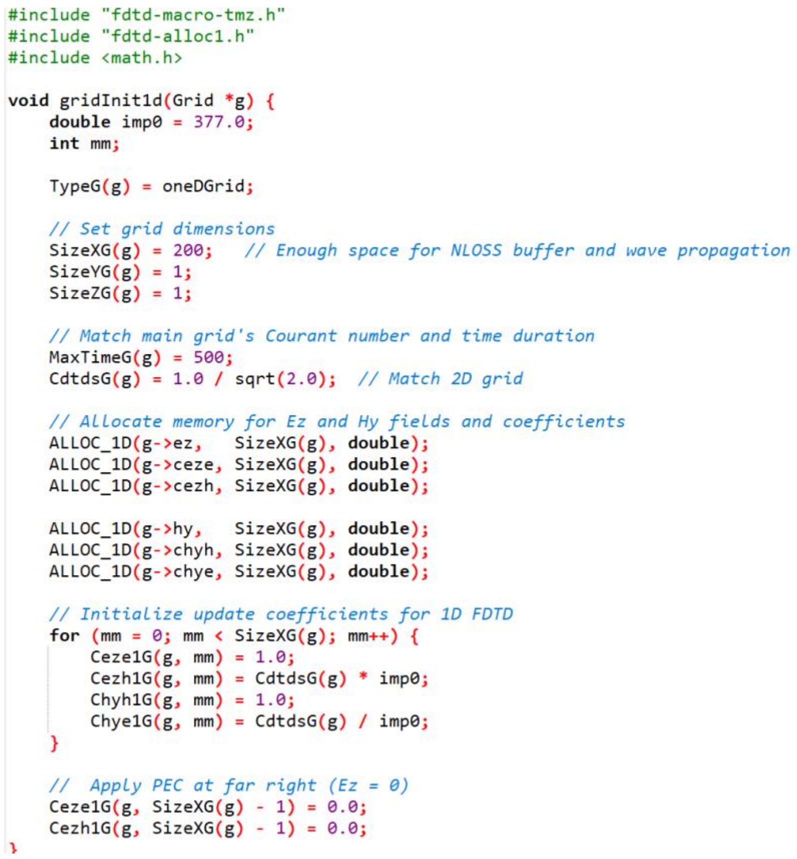
### hw7main.c

```c
/* TMz simulation with a TFSF boundary and a second-order ABC.
   Here the maximum value of the fields along the right side of the
   grid are recorded. */

#include <math.h>
#include <stdio.h>
#include "fdtd-alloc1.h"
#include "fdtd-macro-tmz.h"
#include "fdtd-proto.h"

int main() {
  double *ezMax;
  int nn;
  FILE *out;

  Grid *g;

  // --- Allocate and initialize grid ---
  ALLOC_1D(g, 1, Grid); // allocate memory for 2D grid
  gridInit(g);          // initialize 2D grid: coefficients, PEC screen

  abcInit(g);           // initialize second-order ABC (2D grid)
  tfsfInit(g);          // initialize TFSF region and 1D grid + source
  snapshotInit2d(g);    // set up snapshot control (temporal & spatial)

  // --- Allocate and zero max Ez tracker ---
  ALLOC_1D(ezMax, SizeY, double);
  for (nn = 0; nn < SizeY; nn++)
    ezMax[nn] = 0.0;

  // --- Time-stepping loop ---
  for (Time = 0; Time < MaxTime; Time++) {
    updateH2d(g);     // update magnetic fields (Hx, Hy, Hy1)
    tfsfUpdate(g);    // inject harmonic source and correct TFSF boundary
    updateE2d(g);     // update electric fields (Ez, Ez1) + 1D ABC
    abc(g);           // apply 2D ABC

    // --- Track max |Ez| at right side for last 60 steps ---
    if (Time > MaxTime - 60) {
      for (nn = 0; nn < SizeY; nn++) {
        if (fabs(Ez(SizeX - 2, nn)) > ezMax[nn])
          ezMax[nn] = fabs(Ez(SizeX - 2, nn));
      }
    }

    snapshot2d(g);  // save Ez snapshot (if conditions met)
  }

  // --- Output Ez max profile ---
  out = fopen("ezmax", "w");
  for (nn = 0; nn < SizeY; nn++)
    fprintf(out, "%lf\n", ezMax[nn]);
  fclose(out);

  return 0;
}
```

**gridtmzpec.c**

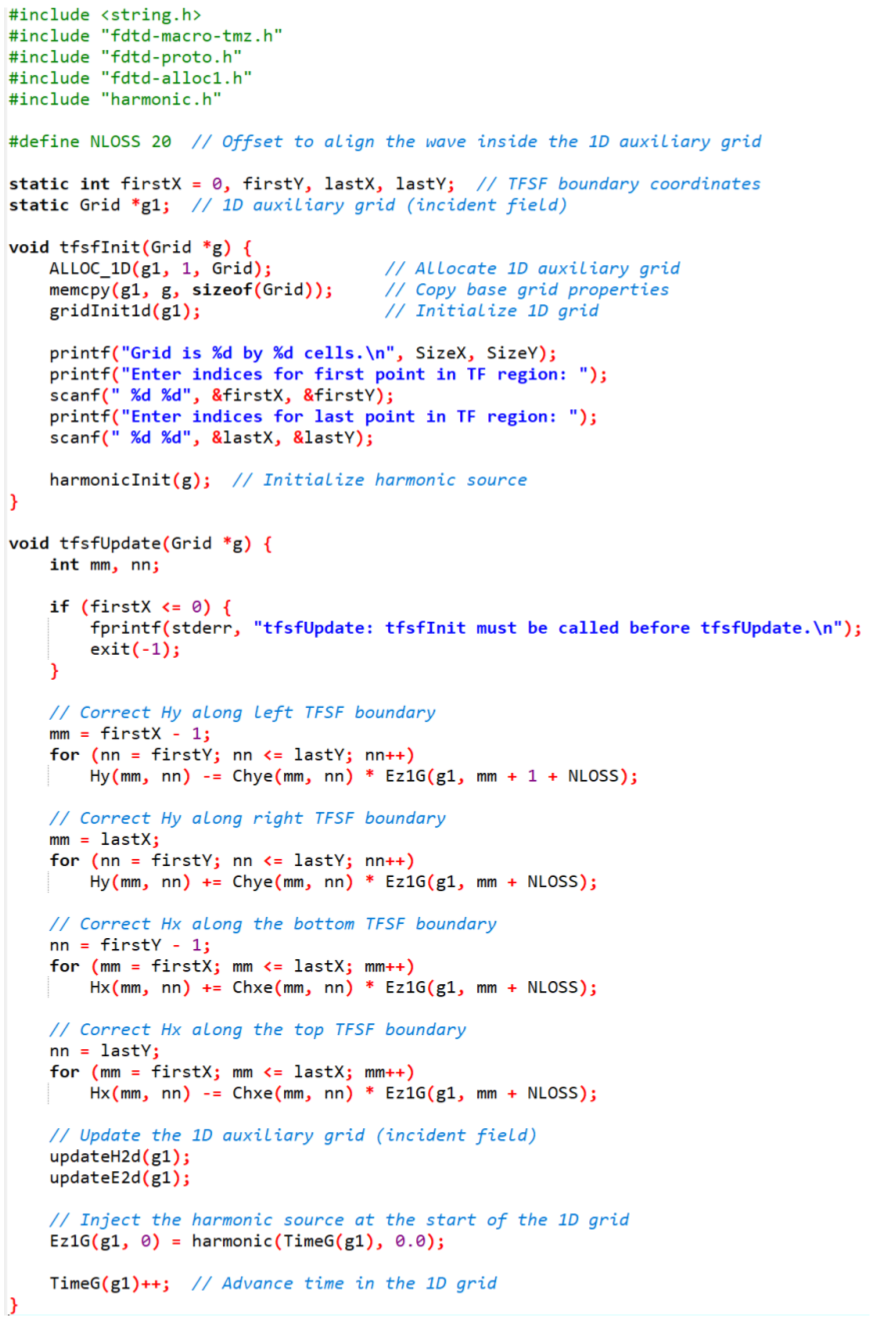
```c
/* Function to create the 2D grid with the appropriate slits in a PEC
   screen to perform the Young's experiment. */

#include "fdtd-macro-tmz.h"
#include "fdtd-alloc1.h"
#include <math.h>

#define PEC_LOCATION 20  // x-index of the PEC screen

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

  Type    = tmZGrid;
  SizeX   = 151; // x size of domain
  SizeY   = 261; // y size of domain
  MaxTime = 500; // simulation time steps
  Cdtds   = 1.0 / sqrt(2.0); // Courant number for 2D

  ALLOC_2D(g->hx,   SizeX, SizeY - 1, double);
  ALLOC_2D(g->chxh, SizeX, SizeY - 1, double);
  ALLOC_2D(g->chxe, SizeX, SizeY - 1, double);
  ALLOC_2D(g->hy,   SizeX - 1, SizeY, double);
  ALLOC_2D(g->chyh, SizeX - 1, SizeY, double);
  ALLOC_2D(g->chye, SizeX - 1, SizeY, double);
  ALLOC_2D(g->ez,   SizeX, SizeY, double);
  ALLOC_2D(g->ceze, SizeX, SizeY, double);
  ALLOC_2D(g->cezh, SizeX, SizeY, double);

  // Initialize electric field update coefficients
  for (mm = 0; mm < SizeX; mm++) {
    for (nn = 0; nn < SizeY; nn++) {
      Ceze(mm, nn) = 1.0;
      Cezh(mm, nn) = Cdtds * imp0;
    }
  }

  // Apply PEC wall with two slits at x = PEC_LOCATION
  mm = PEC_LOCATION;
  for (nn = 0; nn < SizeY; nn++) {
    if (!((nn >= 100 && nn <= 109) || (nn >= 150 && nn <= 159))) {
      Ceze(mm, nn) = 0.0;
      Cezh(mm, nn) = 0.0;
    }
  }

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

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

  return;
}
```

**grid1da.c**



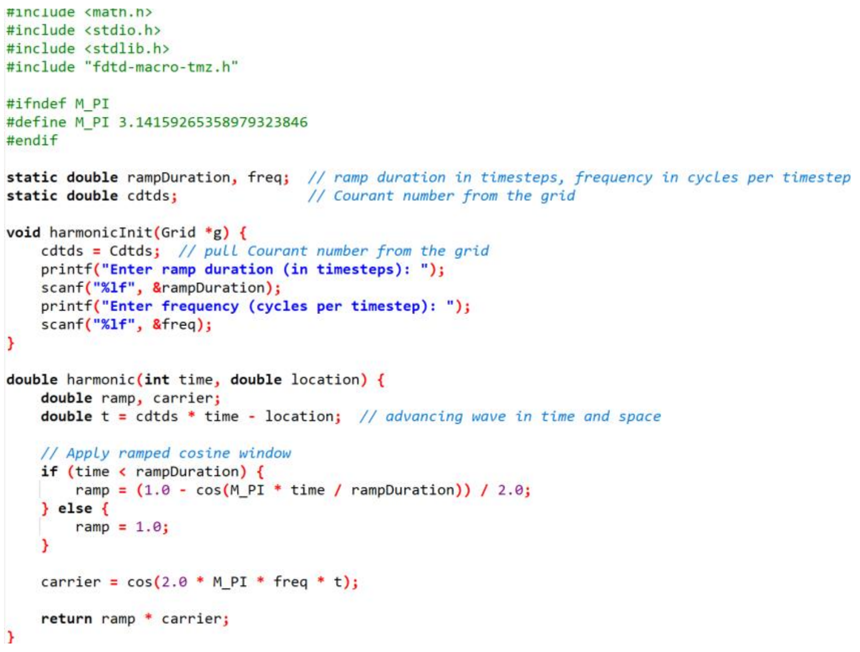
```c
#include "fdtd-macro-tmz.h"
#include "fdtd-alloc1.h"
#include <math.h>

void gridInit1d(Grid *g) {
    double imp0 = 377.0;
    int mm;

    TypeG(g) = oneDGrid;

    // Set grid dimensions
    SizeXG(g) = 200;   // Enough space for NLOSS buffer and wave propagation
    SizeYG(g) = 1;
    SizeZG(g) = 1;

    // Match main grid's Courant number and time duration
    MaxTimeG(g) = 500;
    CdtdsG(g) = 1.0 / sqrt(2.0);  // Match 2D grid

    // Allocate memory for Ez and Hy fields and coefficients
    ALLOC_1D(g->ez,   SizeXG(g), double);
    ALLOC_1D(g->ceze, SizeXG(g), double);
    ALLOC_1D(g->cezh, SizeXG(g), double);

    ALLOC_1D(g->hy,   SizeXG(g), double);
    ALLOC_1D(g->chyh, SizeXG(g), double);
    ALLOC_1D(g->chye, SizeXG(g), double);

    // Initialize update coefficients for 1D FDTD
    for (mm = 0; mm < SizeXG(g); mm++) {
        Ceze1G(g, mm) = 1.0;
        Cezh1G(g, mm) = CdtdsG(g) * imp0;
        Chyh1G(g, mm) = 1.0;
        Chye1G(g, mm) = CdtdsG(g) / imp0;
    }

    //  Apply PEC at far right (Ez = 0)
    Ceze1G(g, SizeXG(g) - 1) = 0.0;
    Cezh1G(g, SizeXG(g) - 1) = 0.0;
}
```

tfsftmz.c



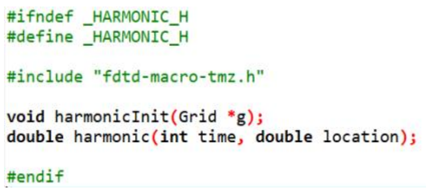
```c
#include <string.h>
#include "fdtd-macro-tmz.h"
#include "fdtd-proto.h"
#include "fdtd-alloc1.h"
#include "harmonic.h"

#define NLOSS 20  // Offset to align the wave inside the 1D auxiliary grid

static int firstX = 0, firstY, lastX, lastY;  // TFSF boundary coordinates
static Grid *g1;  // 1D auxiliary grid (incident field)

void tfsfInit(Grid *g) {
    ALLOC_1D(g1, 1, Grid);            // Allocate 1D auxiliary grid
    memcpy(g1, g, sizeof(Grid));      // Copy base grid properties
    gridInit1d(g1);                   // Initialize 1D grid

    printf("Grid is %d by %d cells.\n", SizeX, SizeY);
    printf("Enter indices for first point in TF region: ");
    scanf(" %d %d", &firstX, &firstY);
    printf("Enter indices for last point in TF region: ");
    scanf(" %d %d", &lastX, &lastY);

    harmonicInit(g);  // Initialize harmonic source
}

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

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

    // Correct Hy along left TFSF boundary
    mm = firstX - 1;
    for (nn = firstY; nn <= lastY; nn++)
        Hy(mm, nn) -= Chye(mm, nn) * Ez1G(g1, mm + 1 + NLOSS);

    // Correct Hy along right TFSF boundary
    mm = lastX;
    for (nn = firstY; nn <= lastY; nn++)
        Hy(mm, nn) += Chye(mm, nn) * Ez1G(g1, mm + NLOSS);

    // Correct Hx along the bottom TFSF boundary
    nn = firstY - 1;
    for (mm = firstX; mm <= lastX; mm++)
        Hx(mm, nn) += Chxe(mm, nn) * Ez1G(g1, mm + NLOSS);

    // Correct Hx along the top TFSF boundary
    nn = lastY;
    for (mm = firstX; mm <= lastX; mm++)
        Hx(mm, nn) -= Chxe(mm, nn) * Ez1G(g1, mm + NLOSS);

    // Update the 1D auxiliary grid (incident field)
    updateH2d(g1);
    updateE2d(g1);

    // Inject the harmonic source at the start of the 1D grid
    Ez1G(g1, 0) = harmonic(TimeG(g1), 0.0);

    TimeG(g1)++;  // Advance time in the 1D grid
}
```

## harmonic.c



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

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

static double rampDuration, freq;  // ramp duration in timesteps, frequency in cycles per timestep
static double cdtds;               // Courant number from the grid

void harmonicInit(Grid *g) {
    cdtds = Cdtds;  // pull Courant number from the grid
    printf("Enter ramp duration (in timesteps): ");
    scanf("%lf", &rampDuration);
    printf("Enter frequency (cycles per timestep): ");
    scanf("%lf", &freq);
}

double harmonic(int time, double location) {
    double ramp, carrier;
    double t = cdtds * time - location;  // advancing wave in time and space

    // Apply ramped cosine window
    if (time < rampDuration) {
        ramp = (1.0 - cos(M_PI * time / rampDuration)) / 2.0;
    } else {
        ramp = 1.0;
    }

    carrier = cos(2.0 * M_PI * freq * t);

    return ramp * carrier;
}
```

## harmonic.h

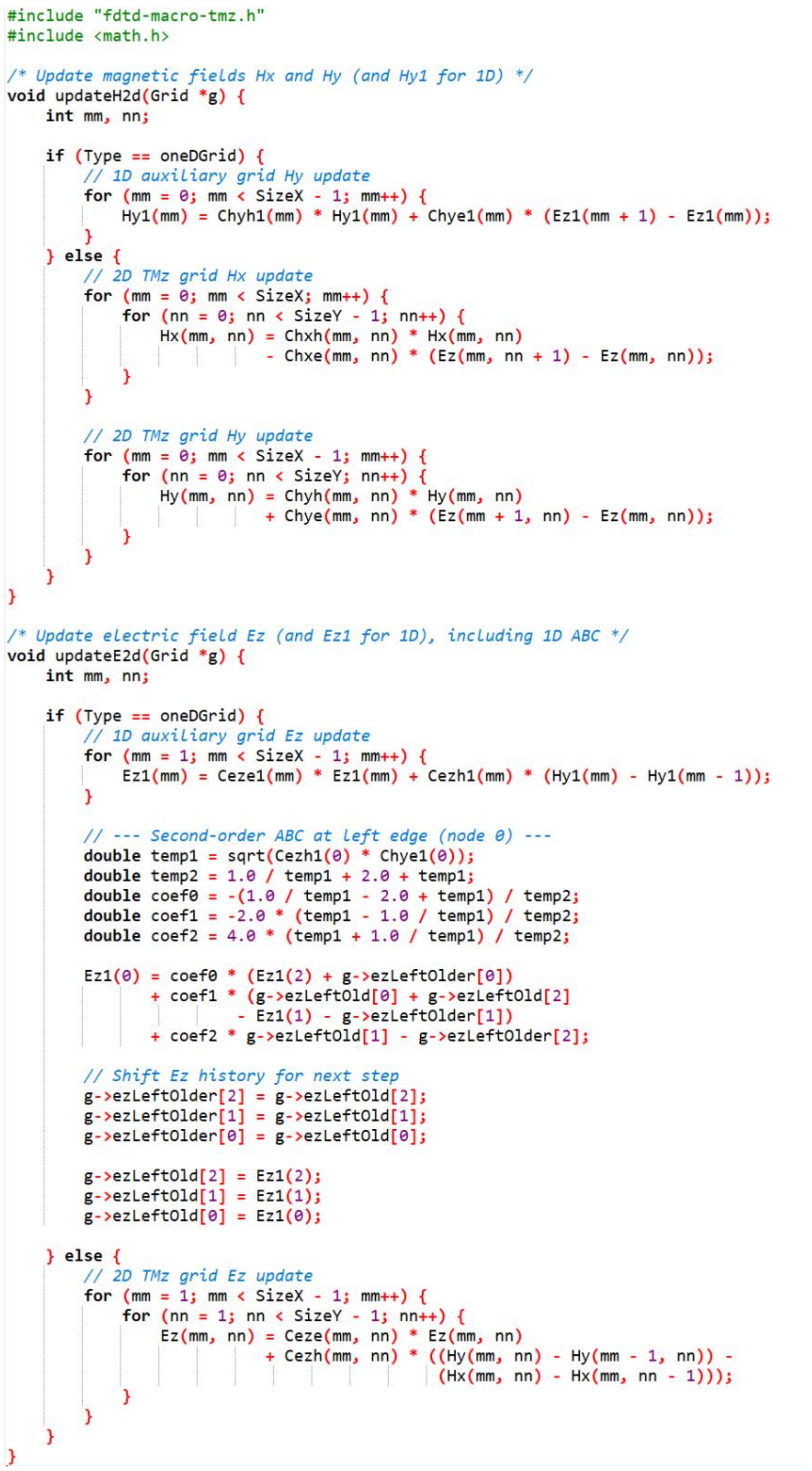


```
#ifndef _HARMONIC_H
#define _HARMONIC_H

#include "fdtd-macro-tmz.h"

void harmonicInit(Grid *g);
double harmonic(int time, double location);

#endif
```

## snapshot2d.c

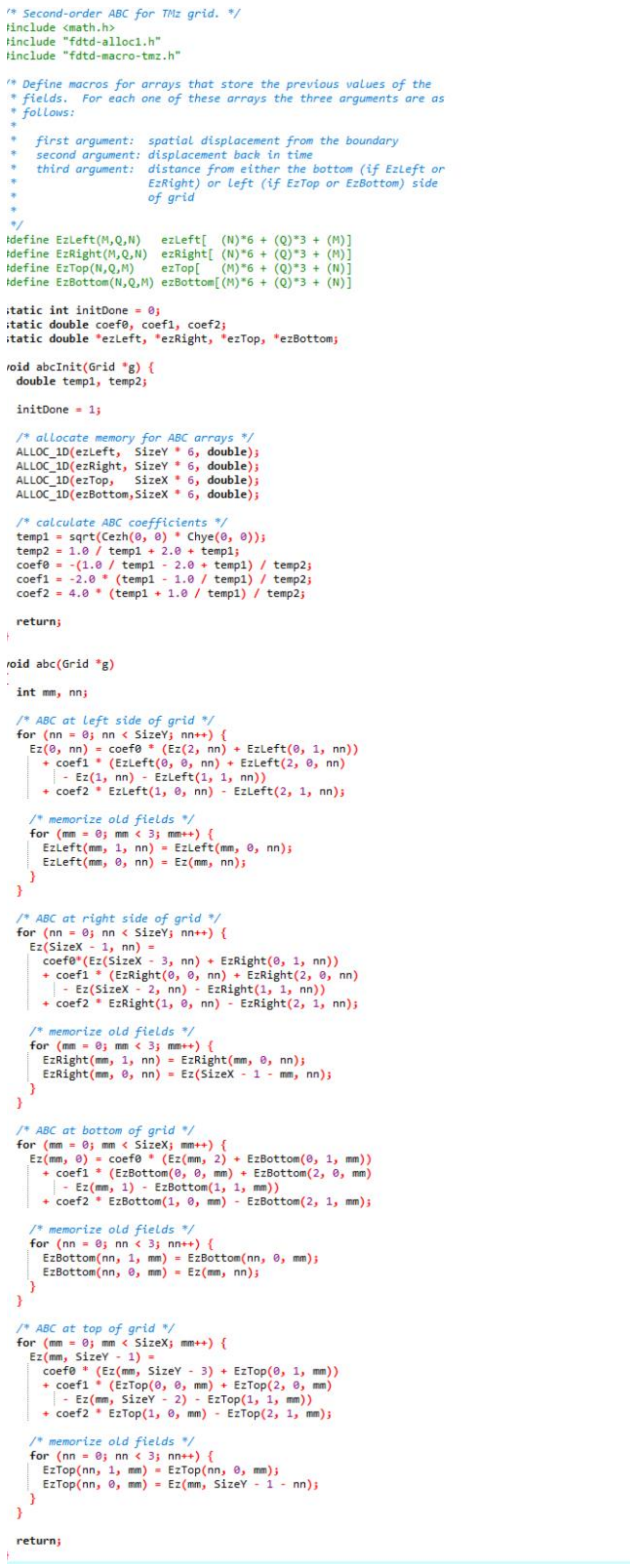
```c
#include <stdio.h>
#include <stdlib.h>
#include "fdtd-macro-tmz.h"

static int temporalStride = -2, startTime;
static int startNodeX, endNodeX, spatialStrideX;
static int startNodeY, endNodeY, spatialStrideY;
static char baseName[80];  // stored globally for reuse
static int frameIndex = 0; // counts how many snapshots taken

void snapshotInit2d(Grid *g) {
    int choice;

    printf("Do you want 2D snapshots? (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("In x direction grid has %d total nodes (ranging from 0 to %d).\n", SizeX, SizeX - 1);
    printf("Enter first node, last node, and spatial stride: ");
    scanf(" %d %d %d", &startNodeX, &endNodeX, &spatialStrideX);

    printf("In y direction grid has %d total nodes (ranging from 0 to %d).\n", SizeY, SizeY - 1);
    printf("Enter first node, last node, and spatial stride: ");
    scanf(" %d %d %d", &startNodeY, &endNodeY, &spatialStrideY);

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

    frameIndex = 0;
}

void snapshot2d(Grid *g) {
    int mm, nn;
    FILE *out;
    char filename[100];

    if (temporalStride == -1) return;

    if (temporalStride < -1) {
        fprintf(stderr,
            "snapshot2d: snapshotInit2d must be called before snapshot.\n"
            "            Temporal stride must be set to positive value.\n");
        exit(-1);
    }

    if (Time >= startTime && (Time - startTime) % temporalStride == 0) {
        // Name each file
        sprintf(filename, "%s.%d", baseName, frameIndex++);
        out = fopen(filename, "w");
        if (!out) {
            perror("snapshot2d");
            exit(-1);
        }

        // Write dimensions
        fprintf(out, "%d\n", (endNodeX - startNodeX) / spatialStrideX + 1);
        fprintf(out, "%d\n", (endNodeY - startNodeY) / spatialStrideY + 1);

        // Write Ez field snapshot
        for (nn = endNodeY; nn >= startNodeY; nn -= spatialStrideY) {
            for (mm = startNodeX; mm <= endNodeX; mm += spatialStrideX) {
                fprintf(out, "%g\n", Ez(mm, nn));
            }
        }

        fclose(out);
    }
}
```

**update2d.c**



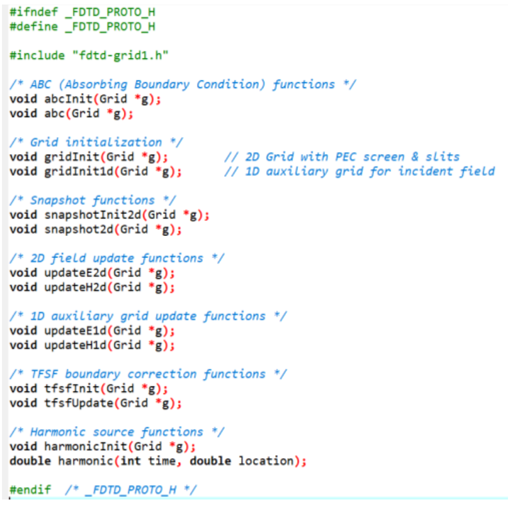
```c
#include "fdtd-macro-tmz.h"
#include <math.h>

/* Update magnetic fields Hx and Hy (and Hy1 for 1D) */
void updateH2d(Grid *g) {
    int mm, nn;

    if (Type == oneDGrid) {
        // 1D auxiliary grid Hy update
        for (mm = 0; mm < SizeX - 1; mm++) {
            Hy1(mm) = Chyh1(mm) * Hy1(mm) + Chye1(mm) * (Ez1(mm + 1) - Ez1(mm));
        }
    } else {
        // 2D TMz grid Hx update
        for (mm = 0; mm < SizeX; mm++) {
            for (nn = 0; nn < SizeY - 1; nn++) {
                Hx(mm, nn) = Chxh(mm, nn) * Hx(mm, nn)
                           - Chxe(mm, nn) * (Ez(mm, nn + 1) - Ez(mm, nn));
            }
        }

        // 2D TMz grid Hy update
        for (mm = 0; mm < SizeX - 1; mm++) {
            for (nn = 0; nn < SizeY; nn++) {
                Hy(mm, nn) = Chyh(mm, nn) * Hy(mm, nn)
                           + Chye(mm, nn) * (Ez(mm + 1, nn) - Ez(mm, nn));
            }
        }
    }
}

/* Update electric field Ez (and Ez1 for 1D), including 1D ABC */
void updateE2d(Grid *g) {
    int mm, nn;

    if (Type == oneDGrid) {
        // 1D auxiliary grid Ez update
        for (mm = 1; mm < SizeX - 1; mm++) {
            Ez1(mm) = Ceze1(mm) * Ez1(mm) + Cezh1(mm) * (Hy1(mm) - Hy1(mm - 1));
        }

        // --- Second-order ABC at left edge (node 0) ---
        double temp1 = sqrt(Cezh1(0) * Chye1(0));
        double temp2 = 1.0 / temp1 + 2.0 + temp1;
        double coef0 = -(1.0 / temp1 - 2.0 + temp1) / temp2;
        double coef1 = -2.0 * (temp1 - 1.0 / temp1) / temp2;
        double coef2 = 4.0 * (temp1 + 1.0 / temp1) / temp2;

        Ez1(0) = coef0 * (Ez1(2) + g->ezLeftOlder[0])
               + coef1 * (g->ezLeftOld[0] + g->ezLeftOld[2]
                       - Ez1(1) - g->ezLeftOlder[1])
               + coef2 * g->ezLeftOld[1] - g->ezLeftOlder[2];

        // Shift Ez history for next step
        g->ezLeftOlder[2] = g->ezLeftOld[2];
        g->ezLeftOlder[1] = g->ezLeftOld[1];
        g->ezLeftOlder[0] = g->ezLeftOld[0];

        g->ezLeftOld[2] = Ez1(2);
        g->ezLeftOld[1] = Ez1(1);
        g->ezLeftOld[0] = Ez1(0);

    } else {
        // 2D TMz grid Ez update
        for (mm = 1; mm < SizeX - 1; mm++) {
            for (nn = 1; nn < SizeY - 1; nn++) {
                Ez(mm, nn) = Ceze(mm, nn) * Ez(mm, nn)
                           + Cezh(mm, nn) * ((Hy(mm, nn) - Hy(mm - 1, nn)) -
                                             (Hx(mm, nn) - Hx(mm, nn - 1)));
            }
        }
    }
}
```

**abctmz.c**



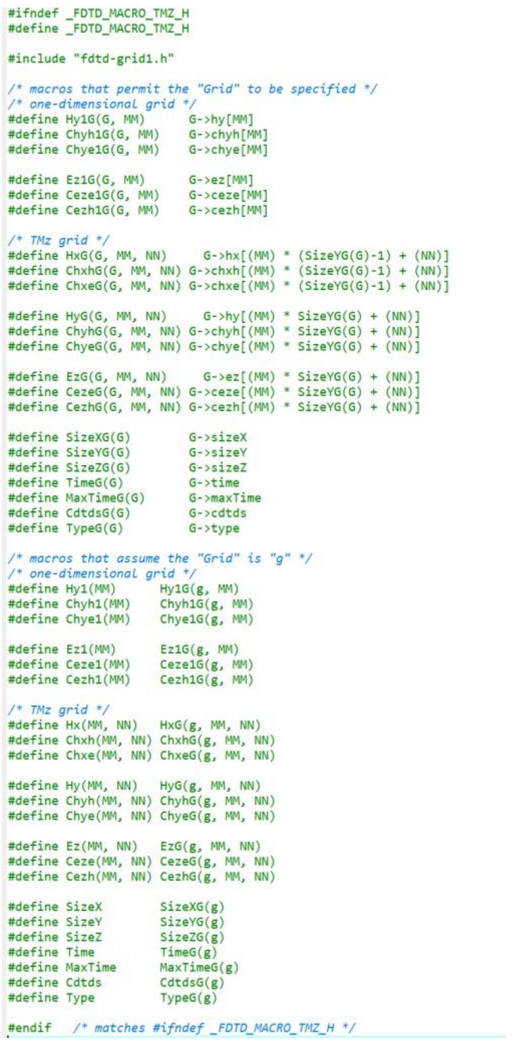
```c
/* Second-order ABC for TMz grid. */
#include <math.h>
#include "fdtd-alloc1.h"
#include "fdtd-macro-tmz.h"

/* Define macros for arrays that store the previous values of the
 * fields.  For each one of these arrays the three arguments are as
 * follows:
 *
 *   first argument:  spatial displacement from the boundary
 *   second argument: displacement back in time
 *   third argument:  distance from either the bottom (if EzLeft or
 *                    EzRight) or left (if EzTop or EzBottom) side
 *                    of grid
 *
 */
#define EzLeft(M,Q,N)   ezLeft[  (N)*6 + (Q)*3 + (M)]
#define EzRight(M,Q,N)  ezRight[ (N)*6 + (Q)*3 + (M)]
#define EzTop(N,Q,M)    ezTop[   (M)*6 + (Q)*3 + (N)]
#define EzBottom(N,Q,M) ezBottom[(M)*6 + (Q)*3 + (N)]

static int initDone = 0;
static double coef0, coef1, coef2;
static double *ezLeft, *ezRight, *ezTop, *ezBottom;

void abcInit(Grid *g) {
  double temp1, temp2;

  initDone = 1;

  /* allocate memory for ABC arrays */
  ALLOC_1D(ezLeft,  SizeY * 6, double);
  ALLOC_1D(ezRight, SizeY * 6, double);
  ALLOC_1D(ezTop,   SizeX * 6, double);
  ALLOC_1D(ezBottom,SizeX * 6, double);

  /* calculate ABC coefficients */
  temp1 = sqrt(Cezh(0, 0) * Chye(0, 0));
  temp2 = 1.0 / temp1 + 2.0 + temp1;
  coef0 = -(1.0 / temp1 - 2.0 + temp1) / temp2;
  coef1 = -2.0 * (temp1 - 1.0 / temp1) / temp2;
  coef2 = 4.0 * (temp1 + 1.0 / temp1) / temp2;

  return;
}

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

  /* ABC at left side of grid */
  for (nn = 0; nn < SizeY; nn++) {
    Ez(0, nn) = coef0 * (Ez(2, nn) + EzLeft(0, 1, nn))
      + coef1 * (EzLeft(0, 0, nn) + EzLeft(2, 0, nn)
         - Ez(1, nn) - EzLeft(1, 1, nn))
      + coef2 * EzLeft(1, 0, nn) - EzLeft(2, 1, nn);

    /* memorize old fields */
    for (mm = 0; mm < 3; mm++) {
      EzLeft(mm, 1, nn) = EzLeft(mm, 0, nn);
      EzLeft(mm, 0, nn) = Ez(mm, nn);
    }
  }

  /* ABC at right side of grid */
  for (nn = 0; nn < SizeY; nn++) {
    Ez(SizeX - 1, nn) =
      coef0*(Ez(SizeX - 3, nn) + EzRight(0, 1, nn))
      + coef1 * (EzRight(0, 0, nn) + EzRight(2, 0, nn)
         - Ez(SizeX - 2, nn) - EzRight(1, 1, nn))
      + coef2 * EzRight(1, 0, nn) - EzRight(2, 1, nn);

    /* memorize old fields */
    for (mm = 0; mm < 3; mm++) {
      EzRight(mm, 1, nn) = EzRight(mm, 0, nn);
      EzRight(mm, 0, nn) = Ez(SizeX - 1 - mm, nn);
    }
  }

  /* ABC at bottom of grid */
  for (mm = 0; mm < SizeX; mm++) {
    Ez(mm, 0) = coef0 * (Ez(mm, 2) + EzBottom(0, 1, mm))
      + coef1 * (EzBottom(0, 0, mm) + EzBottom(2, 0, mm)
         - Ez(mm, 1) - EzBottom(1, 1, mm))
      + coef2 * EzBottom(1, 0, mm) - EzBottom(2, 1, mm);

    /* memorize old fields */
    for (nn = 0; nn < 3; nn++) {
      EzBottom(nn, 1, mm) = EzBottom(nn, 0, mm);
      EzBottom(nn, 0, mm) = Ez(mm, nn);
    }
  }

  /* ABC at top of grid */
  for (mm = 0; mm < SizeX; mm++) {
    Ez(mm, SizeY - 1) =
      coef0 * (Ez(mm, SizeY - 3) + EzTop(0, 1, mm))
      + coef1 * (EzTop(0, 0, mm) + EzTop(2, 0, mm)
         - Ez(mm, SizeY - 2) - EzTop(1, 1, mm))
      + coef2 * EzTop(1, 0, mm) - EzTop(2, 1, mm);

    /* memorize old fields */
    for (nn = 0; nn < 3; nn++) {
      EzTop(nn, 1, mm) = EzTop(nn, 0, mm);
      EzTop(nn, 0, mm) = Ez(mm, SizeY - 1 - nn);
    }
  }

  return;
}
```

## fdtd-proto.h



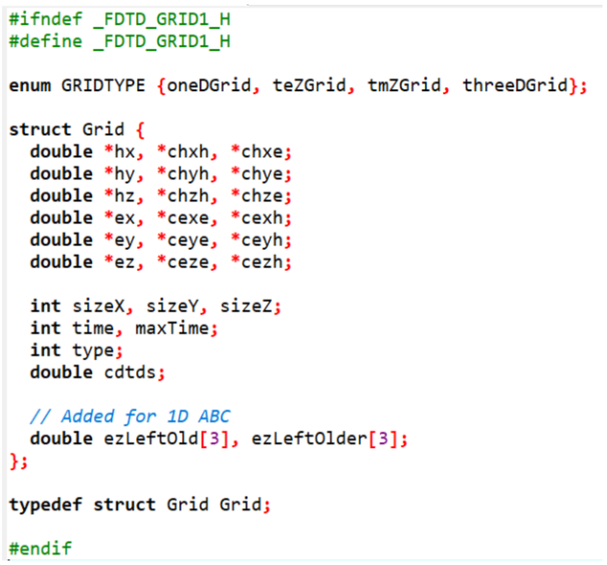
```
#ifndef _FDTD_PROTO_H
#define _FDTD_PROTO_H

#include "fdtd-grid1.h"

/* ABC (Absorbing Boundary Condition) functions */
void abcInit(Grid *g);
void abc(Grid *g);

/* Grid initialization */
void gridInit(Grid *g);        // 2D Grid with PEC screen & slits
void gridInit1d(Grid *g);      // 1D auxiliary grid for incident field

/* Snapshot functions */
void snapshotInit2d(Grid *g);
void snapshot2d(Grid *g);

/* 2D field update functions */
void updateE2d(Grid *g);
void updateH2d(Grid *g);

/* 1D auxiliary grid update functions */
void updateE1d(Grid *g);
void updateH1d(Grid *g);

/* TFSF boundary correction functions */
void tfsfInit(Grid *g);
void tfsfUpdate(Grid *g);

/* Harmonic source functions */
void harmonicInit(Grid *g);
double harmonic(int time, double location);

#endif  /* _FDTD_PROTO_H */
```

## fdtd-macro-tmz.h



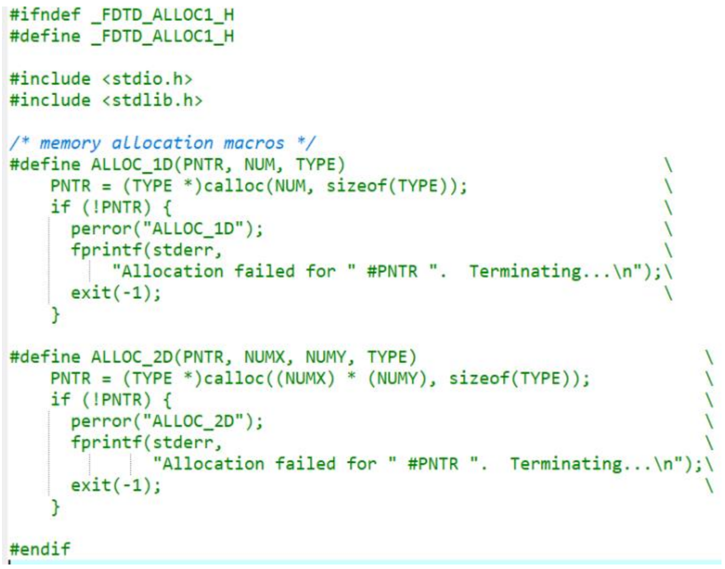
```
#ifndef _FDTD_MACRO_TMZ_H
#define _FDTD_MACRO_TMZ_H

#include "fdtd-grid1.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 HxG(G, MM, NN)     G->hx[(MM) * (SizeYG(G)-1) + (NN)]
#define ChxhG(G, MM, NN) G->chxh[(MM) * (SizeYG(G)-1) + (NN)]
#define ChxeG(G, MM, NN) G->chxe[(MM) * (SizeYG(G)-1) + (NN)]

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

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

#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 Hx(MM, NN)   HxG(g, MM, NN)
#define Chxh(MM, NN) ChxhG(g, MM, NN)
#define Chxe(MM, NN) ChxeG(g, MM, NN)

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

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

#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   /* matches #ifndef _FDTD_MACRO_TMZ_H */
```

## fdtd-grid1.h



```c
#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-alloc1.h



```c
#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);                                                    \
    }

#endif
```