TIF330
Computational Continuum physics

Problem Set 1

Date: 2023-04-18

Students:
Afonso Sequeira Azenha

Problem 3

In this problem, we are asked to develop a `C++` implementation of the 1D FDTD method for numerical solution of Maxwell's equations in vacuum:

$$\begin{cases} \frac{\partial E_y}{\partial t} = -\frac{\partial B_z}{\partial x} \\ \frac{\partial B_z}{\partial t} = -\frac{\partial E_y}{\partial x} \end{cases} \tag{1}$$

Moreover, we consider the propagation of a linearly polarized electromagnetic pulse, for $x \in [-1, 1]$, towards the positive x direction, and for a time $t \in [0, 2.5]$, starting from the following initial conditions:

$$E_y(x, t = 0) = \begin{cases} \sin(20\pi x), & |x| < 0.1 \\ 0 & |x| \geq 0.1 \end{cases} \tag{2}$$

Since the pulse is linearly polarized, moving towards the positive x direction, this unambiguously defines the z component of the B field, for a given E_y. In order to implement this in our initial conditions, we make use of the fact that the simulation grid is staggered, not only space, but also in time. This allows us to define B_z after "half" a time-step as:

$$B_z(x, t = \tau/2) = \begin{cases} \sin(20\pi x), & |x| < 0.1 \\ 0 & |x| \geq 0.1 \end{cases} \tag{3}$$

By the time grid being staggered, it is meant that, in one time-step, τ, one first computes the B field (everywhere), followed by the E field, such that one can consider that each of them was computed after "half" a step. In practice, I implement the initial condition for the B field in the absolute first iteration of my time loop, before the E field is updated.

Additional notes: By default, I work with 2048 grid points, which is a lot! This might make the `gif` generation process, in `Python`, take a while. For quicker results, change the `SIZE` macro's value, in both the `C++` code and the `Python` script. Also, note that to speed up the whole running process (and save-up on memory), I only save 1 time-step's values every $a = 2$ time-steps (This can be changed in the `Write_to_file` and `Write_Vec_to_file` functions). For flawless operation of the `Python` script, it's important that a be a divisor of `SIZE`. If that's not the case, the file `timeVec.txt` will have an extra **comma** (`,`) at the end, that must be removed manually. Of course, this could have also been automated, but that wasn't implemented, since I generally work with multiples of 2 (or 4) for `SIZE`, which makes this problem nonexistent.

Periodic boundary conditions

In this case, we are asked to plot $E_y(x)$ and $B_z(x)$ at $t = 2.5$. My code will generate a `gif` with an animation of the propagation of the electromagnetic fields (as well as many images, used to build that `gif`). The requested plot will be the last image. For convenience, I reproduce it here.

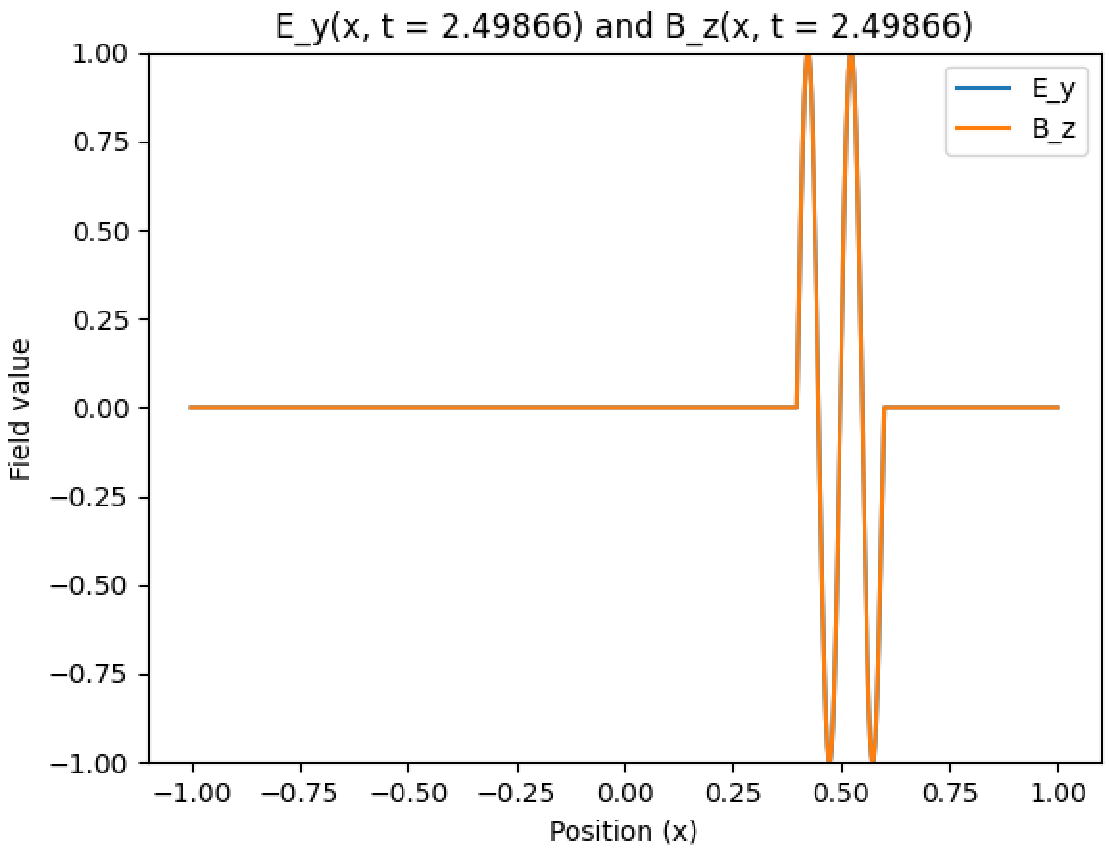

Figure 1: Electromagnetic fields at $t = 2.5$, in the periodic boundary condition case. The onset of numerical dispersion is clearly visible!

In order to produce the aforementioned `gifs` and images, first compile and run the `C++` code, followed by the `Python` script. The first will generate a bunch of `.txt` files with the values of the E and B fields (for all the many time-steps), resultant from the FDTD calculations, for each of the three boundary condition cases. The later will generate the corresponding images and `gifs`, to help in the visualization of the results. Import aspects to mention about the developed code:

1. My space grid always starts at an E field point and, likewise, always ends at an E field point. This is just a choice that I made. As a consequence, we have one less B field point than E field points. By doing things this way, I also only need to implement boundary conditions for the (extreme) E field points (**not** for the B field), since there is no B field point at the grid boundary;

2. The time-step is chosen to be equal to **twice** the distance between adjacent E and B field points, or, mathematically: $\tau = 2h$. This will be important, later, for the implementation of the perfect boundary condition. Also, note that

(Courant) stability is assured so long as: $2h/\tau \geq c_0$[1] (I assume propagation in vacuum. Otherwise, we'd need to include n [refractive index] here as well). This condition, in our system of units, is written as $2h/\tau \geq 1$, since c_0 is taken to be 1. Thus, we have chosen $2h/\tau = 1$ (Stable!), which means that a certain value of the $E(B)$ field will take one time-step (τ) to travel to the next position (index) in its corresponding vector, i.e., to go from `ez[0]` to `ez[1]`[2], if we take Fig. 2 as reference. Additionally, the choice of $\tau = 2h$ is special, in that it yields the exact dispersion relation of the wave equation, i.e., results in **no** numerical dispersion. For this extraordinary property, this choice is sometimes referred to as the "magic choice". We will use it all throughout the three considered cases, below.

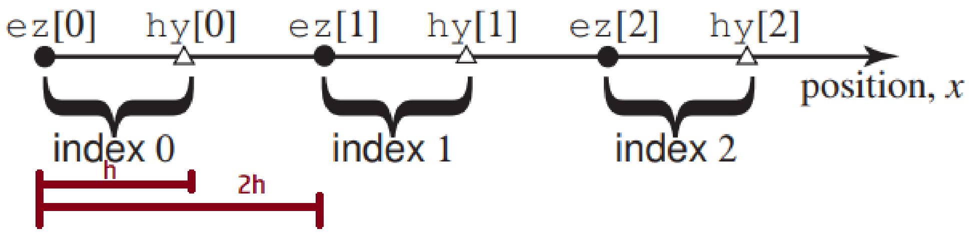

Figure 2: Considered FDM grid and relevant distances. Taken from Ref. [1]. Slight modifications to include distances h and $2h$.

3. Finally, my code computes all the three boundary condition cases all at once. So, it needs only to be run once, followed by the `Python` script (also once). **Notes**: run the `C++` and `Python` codes from the same directory, and keep in mind that the `Python` code requires `numpy`, `matplotlib` (plots) and `imageio` (for making the `gif`).

At each iteration, all the "middle" points are taken care of using "regular" 1D FDTD update equations. In my code, these would be:

```
E_y[mm] = E_y[mm] - (tau/(2*h))*(B_z[mm] - B_z[mm - 1]);
B_z[mm] = B_z[mm] - (tau/(2*h))*(E_y[mm + 1] - E_y[mm]);
```
(4)

The only place where we can have "problems" is at the boundaries, where we would require the value of the magnetic field (and in my case only the magnetic field, since the grid's extreme points are E at both ends) outside of the grid. In the case of periodic boundary conditions, we simply allow our grid to loop back around. This means that, for the last E point, we utilize the B point before it, as well as the first B point of the grid. The same goes for the first E point of the grid. In my code, this reads:

```
E_y[0]        = E_y[0]        - (tau/h)*(B_z[0] - B_z[SIZE - 2]);
E_y[SIZE - 1] = E_y[SIZE - 1] - (tau/h)*(B_z[0] - B_z[SIZE - 2]);
```
(5)

The results of implementing this type of boundary condition are very easily understood if we look at the produce `gif`, where we see the waves propagating towards the edge of the simulation grid, only to then reappear at the other end of the grid. (**Note**: All the [full] developed codes are presented in the appendix, below.)

[1]The reason why I write $2h$ instead of just h is because of the way I defined the grid in my code. There, the distance between adjacent $E(B)$ points is $2h$, whereas the distance between adjacent E and B points is just h (Alternating E and B points). Check Fig. 2.

[2]Also, note that Fig. 2 does not describe my code. It was taken from Ref. [1], where they have a

Perfect electric conductor boundary conditions

In order to simulate perfect electric conductor (PEC) boundaries, we merely have to set the electric field at the grid boundaries to zero. We no longer allow it to loop back around. This type of boundary condition is used for modelling of a lossless metallic surface (for example, a ground plane)[3], which imposes symmetry for magnetic fields and "magnetic currents" and anti-symmetry for electric fields and electric currents. Due to the way I defined my grid (E points on both extremes), having PEC boundary conditions on both ends simply means putting $E_y = 0$ there. Note that this is fundamentally different from a perfect magnetic conductor (PMC), for which B_z would be zero at the boundaries, instead.

Phenomenologically, the E_y field will be inverted when it reaches the end of the grid, just like waves on a rope, reflecting from a fixed end. The magnetic field, on the other hand, will reflect symmetrically, not inverting its sign. This can be very readily observed in the `gif` that my program produces. Explicitly, this type of boundary condition is implemented as such:

```
E_y[0]        = 0.;
E_y[SIZE - 1] = 0.;
```
(6)

Perfect boundary conditions

Finally, in order to implement what I call "perfect boundary conditions", we come back to the definition of τ as $\tau = 2h$. As mentioned previously, this makes it so that after **one** time-step, the E value that we had at position `e_y[n]` will now be at position `e_y[n + 1]` (Same thing for B). In my code, the perfect boundary conditions are implemented as follows:

```
E_y[0] = Res_E[(qTime + 1) - 1][1];
E_y[SIZE - 1] = Res_E[(qTime + 1) - 1][SIZE - 2];
```
(7)

The method that I used (and implemented) was inspired by Ref. [3], based on G. Mur's work on Absorbing Boundary Conditions (ABCs). More specifically, my implementation can be see as a Mur-type boundary condition, which heavily simplifies under the "magical choice" of time-step (c.f. Eq. (22) of Ref [3]). I simply say that the values of the E field at the grid boundaries are equal to what we had at the immediately adjacent E field point, in the previous time-step (This works precisely because $\tau = 2h$).

A different way of implementing this could be the following: Suppose we are at an extreme E point. In order to compute its next value, we require the value of the B field **one index** outside the grid, as if there was no boundary and the grid just continued infinitely. Now, since we know that after one time-step, our fields will have advanced by one index, we can recognize that the value we are looking for is merely the value of the B field one time-step ago (which we stored previously as part of our solution!). The description of this perfect boundary condition is derived in more detail in Ref. [4], where a more involved discussion of the topic is presented.

With all this being said, I can now easily simulate a part of infinite space, as requested. More specifically, my code demonstrates this by simulating the continuous emission of an antenna placed inside the simulation region. The produced `gif` makes

different convention. In my case, I compute E_y and B_z, not E_z and B_y. This figure is merely used to illustrate the distances involved (in the (2)h discussion from above).

[3]More detailed definition in Ref. [6].

this type of boundary condition very easy to visualize, as it appears the waves simply wander off to infinity.

References

[1] Chapter 3: Introduction to the Finite-Difference Time-Domain Method: FDTD in 1D, Lecture notes by John Schneider. Available at: `https://eecs.wsu.edu/~schneidj/ufdtd/chap3.pdf`;

[2] Perfect Magnetic Conductor. Available at: `https://doc.comsol.com/5.5/doc/com.comsol.help.rf/rf_ug_radio_frequency.07.12.html`;

[3] Chapter 3. The Finite-Difference Time Domain Method (FDTD). Available at: `https://my.ece.utah.edu/~ece6340/LECTURES/lecture%2014/FDTD.pdf`;

[4] Lecture 6 (FDTD) - Implementation of 1D FDTD. Available at: `https://www.youtube.com/watch?v=hNN7EtZsJuU`;

[5] Chapter 8: Two-Dimensional FDTD Simulations, Lecture notes by John Schneider. Available at: `https://eecs.wsu.edu/~schneidj/ufdtd/chap8.pdf`;

[6] Perfect Electric Conductor. Available at: `https://doc.comsol.com/5.5/doc/com.comsol.help.rf/rf_ug_radio_frequency.07.11.html`;

Developed code

C++ code: P3_Final_V3.cpp

```cpp
#include <iostream>
#include <cmath>
#include <vector>
#include <fstream>
#include <string>

#define SIZE 2048

using namespace std;

/* Auxiliary functions */
void Write_to_file(vector<vector<double>> &Res, string filename)
{
    cout << "Entering \"Write_to_file\"... " << endl;
    cout << "Matrix dimensions: " << size(Res) << " " << size(Res[0])
   << endl;
    fstream w_File;
    w_File.open(filename, fstream::out);
    for (int i = 0; i < size(Res); i++)
    {
        if(i % 2 == 0) /* Don't need to save all the time-steps */
        {
            for (int j = 0; j < size(Res[0]); j++)
            {
            if(j < (size(Res[0]) - 1)) w_File << Res[i][j] << ",";
            else                       w_File << Res[i][j] << endl;
            }
        }

    }
    w_File.close();
}

void Write_Vec_to_file(vector<double> &Vec, string filename)
{
    cout << "Entering \"Write_Vec_to_file\"... " << endl;
    fstream w_File;
    w_File.open(filename, fstream::out);
    for (int i = 0; i < size(Vec); i++)
    {
        if(i % 2 == 0) /* Don't need to save all the time-steps */
        {
            if(i < (size(Vec) - 1)) w_File << Vec[i] << ",";
            else                    w_File << Vec[i] << endl;
        }
    }
}

void Add_to_Res(double* E_y, double* B_z, vector<vector<double>> &Res_E
    , vector<vector<double>> &Res_B)
{
    vector<double> E_y_, B_z_;
    E_y_.assign(E_y, E_y + SIZE);
    B_z_.assign(B_z, B_z + (SIZE - 1));
    Res_E.push_back(E_y_);
    Res_B.push_back(B_z_);
    E_y_.clear();
    B_z_.clear();
}

/* Main program */
int main(int argc, const char** argv)
{
/* Boundary condition type (different cases):
1 - Periodic;
2 - Perfect Electric Conductor;
3 - Perfect (Infinity). */
for (int BC_type = 1; BC_type < 4; BC_type++)
{
    /* Variables */
    vector<double> t_vec;
    vector<vector<double>> Res_E, Res_B;
    int             qTime, mm;
    double          E_y[SIZE]       = {0.}, B_z[SIZE - 1] = {0.};
    double          h               = 2/(2 * double(SIZE) - 2);
    double          tau             = 2*h;
    long int        maxTime         = 2.5/tau;
    double          x[2 * SIZE - 1] = {0.};
    bool            delay           = false;
    vector<string> names            = {"Periodic_BC", "PEC_BC", "
   Perfect_BC"};

    /* Printing for debugging/reference */
    printf("Parameters:                  \n");
    printf("h                       = %f\n", h);
    printf("gridSize                = %d\n", SIZE);
    printf("tau                     = %f\n", tau);
    printf("2h/tau                  = %f > c = 1 (Stable)\n", (2*h)/tau)
   ;
    printf("numberOfIterations      = %ld\n", maxTime);

    /* Filling up space grid vector */
    for(mm = 0; mm < (2 * SIZE - 1); mm++) x[mm] = -1 + (mm * h);
```

```
 if(BC_type < 3)
 {
     /* Initial condition */
     for(mm = 0; mm < SIZE; mm++)
     {
         if(abs(x[2 * mm]) < 0.1) E_y[mm] = sin(20. * M_PI * x[2 *
mm]);
         else E_y[mm] = 0.;
     }
 }

 /* 'Pushing back' intial condition */
 t_vec.push_back(0.);
 Add_to_Res(E_y, B_z, Res_E, Res_B);

 /* Do time stepping */
 for (qTime = 0; qTime < maxTime; qTime++)
 {
     /* Staggering the initial B_z field by "half" a time-step */
     if(qTime == 0 && BC_type < 3)
     {
         for(mm = 0; mm < SIZE; mm++)
         {
             if(abs(x[2 * mm]) < 0.1) B_z[mm] = sin(20. * M_PI * x[2
 * mm]);
             else B_z[mm] = 0.;
         }

     }
     else
     {
     /* Update magnetic field. Was missing factors of (tau/h) here (
and in the electric field equation(s))! */
     for (mm = 0; mm < (SIZE - 1); mm++) B_z[mm] = B_z[mm] - (tau
/(2*h))*(E_y[mm + 1] - E_y[mm]);
     }

     /* Update electric field */
     switch (BC_type)
     {
     case 1:
         /* Boundary conditions - Periodic! First and last points,
respectively */
         E_y[0]        = E_y[0]        - (tau/(2*h))*(B_z[0] - B_z[
SIZE - 2]);
         E_y[SIZE - 1] = E_y[SIZE - 1] - (tau/(2*h))*(B_z[0] - B_z[
SIZE - 2]);
         break;
     case 2:
         /* Perfect magnetic conductor at edges */
         // E_y[0]        = E_y[0]        - (tau/(2*h))*(B_z[0] - 0)
;
         // E_y[SIZE - 1] = E_y[SIZE - 1] - (tau/(2*h))*(0 - B_z[
SIZE - 2]);

         /* Perfect electric conductor at edges */
         E_y[0]        = 0.;
         E_y[SIZE - 1] = 0.;
         break;
     case 3:
         /* Perfect boundary conditions. Mur's boundary conditions!
*/
         if(qTime > 0)
         {
             E_y[0]        = Res_E[(qTime + 1) - 1][1];
             E_y[SIZE - 1] = Res_E[(qTime + 1) - 1][SIZE - 2];
         }
         break;
     }

     /* All the other 'middle' points */
     for (mm = 1; mm < SIZE - 1; mm++) E_y[mm] = E_y[mm] - (tau/(2*h
))*(B_z[mm] - B_z[mm - 1]);

     /* Delay counter: Antenna beeps and then waits for 500
iterations before beeping again */
     if(BC_type == 3)
     {
         if ((qTime+1) % 500 == 0) delay = !delay;
         if (!delay)                E_y[SIZE/2] = 0.75 * abs(sin(M_PI
 * (double(qTime)/500.)));
         else                       E_y[SIZE/2] = 0.;
     }

     /* Results */
     if (qTime % 1 == 0) // 'stride' was changed to 1 for now.
     {
         printf("\rProgress: %f%%", 100*(qTime/double(maxTime)));
         t_vec.push_back((qTime + 1)*tau);
         Add_to_Res(E_y, B_z, Res_E, Res_B);
     }

 } /* End of time-stepping */
 cout << endl;

 /* Writing to text file */
 switch (BC_type)
 {
   case 1:
       Write_to_file(Res_E, "E_Periodic_BC.txt");
       Write_to_file(Res_B, "B_Periodic_BC.txt");
       break;
   case 2:
       Write_to_file(Res_E, "E_PEC_BC.txt");
       Write_to_file(Res_B, "B_PEC_BC.txt");
       break;
   case 3:
       Write_to_file(Res_E, "E_Perfect_BC.txt");
       Write_to_file(Res_B, "B_Perfect_BC.txt");
       break;
   }

   /* Additionally saving the time vector, for easier plotting later
  */
   Write_Vec_to_file(t_vec, "timeVec.txt");
}

return(0);
}
}
```

Listing 1: C++ code utilized in Problem 3 to resolve the FDTD scheme. For ease of copying: `https://codeshare.io/6p1Wbm`.

Python script: `P3_Gif.py`

```
import matplotlib.pyplot as plt
import numpy as np
import imageio.v2 as imageio
from pathlib import Path

SIZE = 2048

# Definitions
h          = 2/(2 * float(SIZE) - 2); tau = 4/(SIZE**2)
x_E        = [-1 + (j * 2*h) for j in range(SIZE)]
x_B        = [(-1 + h) + (j * 2*h) for j in range(SIZE - 1)]
Data       = {}

ids        = ['E_y_1', 'B_z_1',
             'E_y_2', 'B_z_2',
             'E_y_3', 'B_z_3', 'TimeVec']

cases      = ['./Periodic_BC', './PEC_BC', './Perfect_BC']

filenames = ['./E_Periodic_BC.txt', './B_Periodic_BC.txt',
             './E_PEC_BC.txt'     , './B_PEC_BC.txt',
             './E_Perfect_BC.txt' , './B_Perfect_BC.txt', 'timeVec.txt'
```

```
   ]

for folder in cases:
    Path(f"{folder}").mkdir(parents=True, exist_ok=True)

for file, id in zip(filenames, ids):
    with open(file, 'r') as f:
        Data[id] = [[float(num) for num in line.split(',')] for line in
    f]

for ind in range(1,4):
    for i in range(0, len(Data[f'E_y_{ind}'])):
        if((i % 40 == 0) or (i == (len(Data[f'E_y_{ind}']) - 1))):
            plt.plot(x_E, Data[f'E_y_{ind}'][i], label = 'E_y')
            plt.plot(x_B, Data[f'B_z_{ind}'][i], label = 'B_z')
            plt.title(f"E_y(x, t = {Data['TimeVec'][0][i]:.5f}) and B_z
   (x, t = {Data['TimeVec'][0][i]:.5f})")
            plt.ylabel(f"Field value")
            plt.xlabel("Position (x)")
            plt.ylim(-1, 1)
            plt.legend()
            plt.savefig(f"./{cases[(ind - 1)]}/Img{int(i/40)}.png", dpi
    = 100)
            plt.savefig(f"./{cases[(ind - 1)]}/Img{int(i/40)}.pdf", dpi
    = 100)
            plt.clf()

    # Small piece of code to join all the 127 images into a gif, for
   easier visualization. Takes some minutes to run.
    filenames = [f'./{cases[(ind - 1)]}/Img{n}.png' for n in range(0,
   int(i/40) + 1)]
    with imageio.get_writer(f'./{cases[(ind - 1)]}/Movie_E+B.gif', mode
   ='I') as writer:
        for filename in filenames:
            image = imageio.imread(filename)
            writer.append_data(image)
```
Listing 2: Python code utilized in Problem 3 to generate the plots and gif. For ease of copying: `https://codeshare.io/OdyxwN`.