TIF330
Computational Continuum physics

Problem Set 1

Date: 2023-04-18

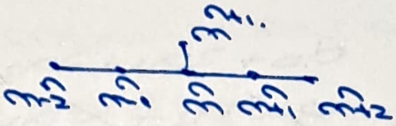
Students:
Afonso Sequeira Azenha

Problem set 1:

Problem 1: Explicit FDM with 5-point stencil for the heat equation.

· Heat equation: $\frac{\partial u}{\partial t} - a\frac{\partial^2 u}{\partial x^2} = 0$.

· 5 point stencil



→ $\frac{\partial u}{\partial t} \approx \frac{u_j^{n+1} - u_j^{n}}{\tau}$.

→ $\frac{\partial^2 u}{\partial x^2} \approx \frac{\alpha_{-2}u_{j-2}^n + \alpha_{-1}u_{j-1}^n + \alpha_0 u_j^n + \alpha_1 u_{j+1}^n + \alpha_2 u_{j+2}^n}{h^2}$

If we consider symmetry: $\alpha_{-2} = \alpha_2$ and $\alpha_{-1} = \alpha_1$.

· Residual: $r_{\tau h} = \underbrace{\frac{u_j^{n+1} - u_j^n}{\tau}}_{A} -$

$$- \frac{a}{h^2}\Big(\underbrace{\alpha_2 u_{j-2}^n}_{B} + \underbrace{\alpha_1 u_{j-1}^n}_{C} + \underbrace{\alpha_0 u_j^n}_{F} + \underbrace{\alpha_1 u_{j+1}^n}_{D} + \underbrace{\alpha_2 u_{j+2}^n}_{E}\Big).$$

· We now perform taylor expansions:

A: $\frac{u_j^n + \tau \dot u_j^n + O(\tau^2) - u_j^n}{\tau} = \dot u_j^n + O(\tau)$.

B: $\alpha_2\Big(u_j^n - 2h u'_{x}{}_j^n + \frac{4h^2}{2}u''_{xx}{}_j^n - \frac{8h^3}{6}u'''_{xxx}{}_j^n +$
$+ \frac{16h^4}{24}u''''_{xxxx}{}_j^n - \frac{32h^5}{120}u'''''_{xxxxx}{}_j^n + O(h^6)\Big).$

C: $\alpha_1\Big(u_j^n - h u'_x{}_j^n + \frac{h^2}{2}u''_{xx}{}_j^n - \frac{h^3}{6}u'''_{xxx}{}_j^n + \frac{h^4}{24}u''''_{xxxx}{}_j^n -$
$- \frac{h^5}{120}u'''''_{xxxxx}{}_j^n + O(h^6)\Big).$

F: $\alpha_0 u_j^n$.

D: $\alpha_1\Big(u_j^n + h u'_x{}_j^n + \frac{h^2}{2}u''_{xx}{}_j^n + \frac{h^3}{6}u'''_{xxx}{}_j^n + \frac{h^4}{24}u''''_{xxxx}{}_j^n +$
$+ \frac{h^5}{120}u'''''_{xxxxx}{}_j^n + O(h^6)\Big).$

E: $\alpha_2\Big(u_j^n + 2h u'_x{}_j^n + \frac{4h^2}{2}u''_{xx}{}_j^n + \frac{8h^3}{6}u'''_{xxx}{}_j^n +$
$+ \frac{16h^4}{24}u''''_{xxxx}{}_j^n + \frac{32h^5}{120}u'''''_{xxxxx}{}_j^n + O(h^6)\Big).$

Summing everything:

$r_{\tau h} = \dot u_j^n + O(\tau) - \frac{a}{h^2}\Big((\alpha_0 + 2\alpha_1 + 2\alpha_2)u_j^n + (4h^2\alpha_2 + h^2\alpha_1)u''_{xx}{}_j^n + \big(\frac{32}{24}h^4\alpha_2 + \frac{2h^4}{24}\alpha_1\big)u''''_{xxxx}{}_j^n + O(h^6)\Big).$

Problem 2: Find the derivative of the function using the definition.

$$f(x) = \frac{1}{x}$$

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

$$= \lim_{h \to 0} \frac{\frac{1}{x+h} - \frac{1}{x}}{h}$$

$$= \lim_{h \to 0} \frac{\frac{x - (x+h)}{x(x+h)}}{h}$$

$$= \lim_{h \to 0} \frac{\frac{-h}{x(x+h)}}{h}$$

$$= \lim_{h \to 0} \frac{-h}{x(x+h)} \cdot \frac{1}{h}$$

$$= \lim_{h \to 0} \frac{-1}{x(x+h)}$$

$$= \frac{-1}{x(x+0)}$$

$$= \frac{-1}{x^2}$$

$$= -x^{-2}$$

$$f'(x) = -x^{-2}$$

Problem 3: Find the derivative of the function using the definition.

$$f(x) = \sqrt{x}$$

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

$$= \lim_{h \to 0} \frac{\sqrt{x+h} - \sqrt{x}}{h}$$

$$= \lim_{h \to 0} \frac{\sqrt{x+h} - \sqrt{x}}{h} \cdot \frac{\sqrt{x+h} + \sqrt{x}}{\sqrt{x+h} + \sqrt{x}}$$

$$= \lim_{h \to 0} \frac{(x+h) - x}{h(\sqrt{x+h} + \sqrt{x})}$$

$$= \lim_{h \to 0} \frac{h}{h(\sqrt{x+h} + \sqrt{x})}$$

$$= \lim_{h \to 0} \frac{1}{\sqrt{x+h} + \sqrt{x}}$$

$$= \frac{1}{\sqrt{x+0} + \sqrt{x}}$$

$$= \frac{1}{2\sqrt{x}}$$

$$f'(x) = \frac{1}{2\sqrt{x}}$$

$$\frac{\partial^2 f}{\partial x^2}\cdot\frac{\partial^2 f}{\partial y^2}-\left(\frac{\partial^2 f}{\partial x\partial y}\right)^2$$

$$\frac{\partial^2 f}{\partial x^2}\cdot\frac{\partial^2 f}{\partial y^2}-\left(\frac{\partial^2 f}{\partial x\partial y}\right)^2$$

$$\frac{\partial^2 f}{\partial x^2}\cdot\frac{\partial^2 f}{\partial y^2}-\left(\frac{\partial^2 f}{\partial x\partial y}\right)^2$$

$$\frac{\partial^2 f}{\partial x^2}\cdot\frac{\partial^2 f}{\partial y^2}-\left(\frac{\partial^2 f}{\partial x\partial y}\right)^2$$

Substituting in eqs. (1), (2) and (3), we obtain: $\lambda = e^{i\omega\tau}$. For eq. (1):

$$C_z \cdot \frac{\lambda^{1/2} - \lambda^{-1/2}}{\tau} = -C\left[\alpha_1 \frac{C_x}{R_x} \cdot 2i\sin\left(\frac{k_x R_x}{2}\right) e^{ik_y R_y} + \alpha_1 \frac{C_x}{R_x} \cdot 2i\sin\left(\frac{k_x R_x}{2}\right) e^{-ik_y R_y} + (1-2\alpha_1)\frac{C_x}{R_x} \cdot 2i\sin\left(\frac{k_x R_x}{2}\right) - \left[\alpha_2 \frac{C_y}{R_y} \cdot 2i\sin\left(\frac{k_y R_y}{2}\right) e^{ik_x R_x} + \alpha_2 \frac{C_y}{R_y} \cdot 2i\sin\left(\frac{k_y R_y}{2}\right) e^{-ik_x R_x} + (1-2\alpha_2)\frac{C_y}{R_y} \cdot 2i\sin\left(\frac{k_y R_y}{2}\right)\right]\right] \Rightarrow$$

$$\Rightarrow -\frac{C_z}{C\tau} \cdot \sin\left(\frac{\omega\tau}{2}\right) = \alpha_1 \frac{C_x}{R_x} \sin\left(\frac{k_x R_x}{2}\right)\underbrace{(2\cos(k_y R_y) - 2)}_{= -4\sin^2\left(\frac{k_y R_y}{2}\right)} + \frac{C_x}{R_x} \sin\left(\frac{k_x R_x}{2}\right) - \left[\alpha_2 \frac{C_y}{R_y} \sin\left(\frac{k_y R_y}{2}\right)\underbrace{(2\cos(k_x R_x) - 2)}_{= -4\sin^2\left(\frac{k_x R_x}{2}\right)} + \frac{C_y}{R_y} \sin\left(\frac{k_y R_y}{2}\right)\right] \Rightarrow$$

$$\Rightarrow -\frac{C_z}{C\tau} \cdot S_t = \frac{C_x S_x}{R_x}\left(1 - 4\alpha_1 S_y^2\right) - \frac{C_y S_y}{R_y}\left(1 - 4\alpha_2 S_x^2\right) \quad (1).$$

Notation:
- $S_t := \sin\left(\frac{\omega\tau}{2}\right)$
- $S_x := \sin\left(\frac{k_x R_x}{2}\right)$
- $S_y := \sin\left(\frac{k_y R_y}{2}\right)$

In order to go further, we use equations (2) and (3), which can be re-written as, after the substitution:

$$\frac{C_x}{\tau} ~~2~~ \sin\left(\frac{\omega\tau}{2}\right) = C \cdot \frac{C_z}{R_y} ~~2~~ \sin\left(\frac{k_y R_y}{2}\right)$$

$$\Rightarrow \boxed{C_z = \frac{S_t}{S_y} \cdot \frac{C_x R_y}{C\tau}} \quad (2).$$

Also: $\sin\left(\frac{\omega\tau}{2}\right) = -C\tau \frac{C_z}{C_y R_x} \sin\left(\frac{k_x R_x}{2}\right) \Rightarrow$

$$\Rightarrow \boxed{C_y = -\frac{S_x}{S_y} \cdot \frac{C_x R_y}{R_x}} \quad (3).$$

To get here, I also had to use the definition of C_z at eq. (2).

We can now substitute C_z and C_y in (1) to obtain:

$$+\left(\frac{S_t}{S_y} \frac{~~C_x~~ ~~R_y~~}{C\tau}\right) \frac{1}{C\tau} S_t = \left(\frac{S_x}{~~S_y~~} \frac{~~C_x~~ R_y}{R_x}\right) \frac{S_x}{R_x}\left(1 - 4\alpha_1 S_y^2\right) + ~~C_x~~ \frac{S_y^2}{R_y^2}\left(1 - 4\alpha_2 S_x^2\right) \Rightarrow$$

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:

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

Since the value of $B_z(x, t = 0)$ is not specified, it is assumed to be zero everywhere.

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 convinence, I reproduce it here.

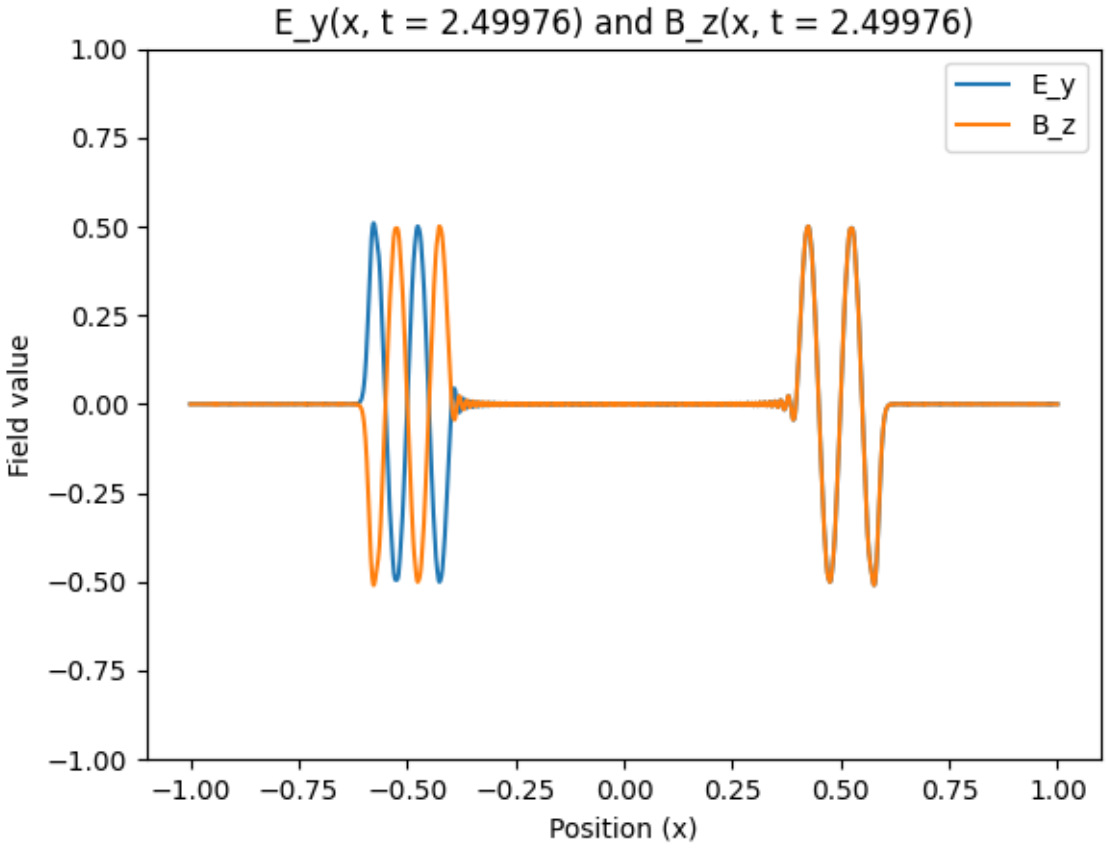


Figure 1: Electromagnetic fields at $t = 2.5$, in the periodic boundary condition case. The onset of numerical dispersion is clearly visible!

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 the distance between adjacent E and B field points. This will be important, later, for the implementation of the perfect boundary condition. Also, note that (Courant) stability is assured so long as: $2h/\tau > c_0$[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 $2h/\tau > 1$, since c_0 is taken to be 1. Thus, we have chosen $2h/\tau = 2$ (Stable!), which means that a certain value of the $E(B)$ field will take two time-steps (2τ) 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.

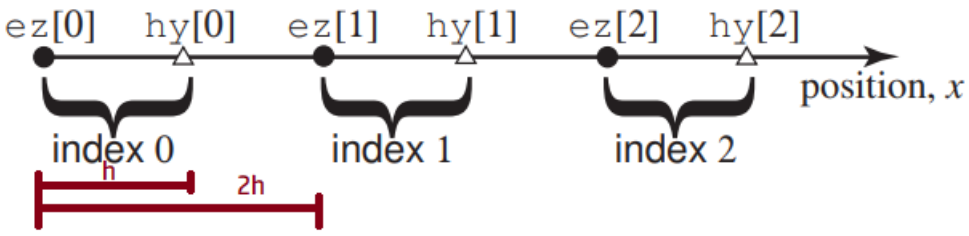


Figure 2: Considered FDM grid and relevant distances. Taken from Ref. [1]. Slight modifications to include distances h and $2h$.

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]);
```
(3)

[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 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).

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]);
```
(4)

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.)

Perfect magnetic conductor boundary conditions

In order to simulate perfect magnetic conductor (PMC) boundaries, we merely have to set the magnetic field outside the grid to zero. We no longer allow it to loop back around. On external boundaries, a perfect magnetic conductor can be interpreted as a "high surface impedance"[3], which doesn't allow current to flow into it (surface current density is zero along this boundary surface). Due to the way I defined my grid (E points on both extremes), we have a PMC on both ends. Had I defined it with a B point in either of its edges, setting the E field outside the grid to zero would be equivalent to having a perfect electric conductor-type boundary, instead. Either way, given how I implemented my grid, the B field will be inverted when it reaches the end of the grid, just like waves on a rope, reflecting from a fixed end. The electric 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]        = 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]);
```
(5)

Perfect boundary conditions

Finally, in order to implement what I call "perfect boundary conditions", we come back to the definition of τ as $\tau = h$. As mentioned previously, this makes it so that after **two** time-steps, 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] = E_y[0] - (tau/(2*h))*(B_z[0] - Res_B[(qTime + 1) - 2][0]);
E_y[SIZE - 1] = E_y[SIZE - 1]  -
(tau/(2*h))*(Res_B[(qTime + 1) - 2][SIZE - 2] - B_z[SIZE - 2]);
```
(6)

The reason why this works is 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

[3]More detailed definition in Ref. [2].

we know that after two time-steps, 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 two time-steps ago (which we stored previously as part of our solution!). The description of this perfect boundary condition is described in more detail in Ref. [3], 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 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] Lecture 6 (FDTD) - Implementation of 1D FDTD. Available at: `https://www.youtube.com/watch?v=hNN7EtZsJuU`;

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

[5] 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.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++)
    {
        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 < (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 Magnetic Counductor;
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             = h;
    long int       maxTime         = 2.5/tau;
    double         x[2 * SIZE - 1] = {0.};
    bool           delay           = false;
    vector<string> names           = {"Periodic_BC", "PMC_BC", "
   Perfect_BC"};

    /* These ended up not being needed */
    int            numberOfImages  = 500; /* We only save 500 images */
    int            stride          = int(double(maxTime)/double(
   numberOfImages));

    /* 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);
    printf("numberOfImages         = %d\n", numberOfImages);
    printf("stride                 = %d\n", stride);

    /* 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++)
 {

     /* 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/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]);
         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]);
         break;
     case 3:
         /* Perfect boundary conditions */
         if(qTime > 0)
         {
             E_y[0]        = E_y[0]        - (tau/(2*h))*(B_z[0] -
Res_B[(qTime + 1) - 2][0]);
             E_y[SIZE - 1] = E_y[SIZE - 1] - (tau/(2*h))*(Res_B[(
qTime + 1) - 2][SIZE - 2] - B_z[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.)));
        }

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

    } /* End of time-stepping */

    /* 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_PMC_BC.txt");
        Write_to_file(Res_B, "B_PMC_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
*/
    // ofstream outFile("timeVec.txt");
    // for (const auto &e : t_vec) outFile << e << "\n"; // Find out
why this doesn't work!
```

```
    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

# Definitions
h         = 2/(2 * float(2048) - 2); tau = 4/(2048**2)
x_E       = [-1 + (j * 2*h) for j in range(2048)]
x_B       = [(-1 + h) + (j * 2*h) for j in range(2047)]
Data      = {}
ids       = ['E_y_1', 'B_z_1',
             'E_y_2', 'B_z_2',
             'E_y_3', 'B_z_3', 'TimeVec']
cases     = ['./Periodic_BC', './PMC_BC', './Perfect_BC']
filenames = ['./E_Periodic_BC.txt', './B_Periodic_BC.txt',
             './E_PMC_BC.txt'     , './B_PMC_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.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`.