FRANKFURT UNIVERSITY OF APPLIED SCIENCES

MASTER THESIS

Finite Difference Time Domain Method Implementation in C++

Author:

Xhoni ROBO

Supervisor:

Prof. Dr. Peter THOMA

Assistant Supervisor:

Prof. Dr. Egbert FALKENBERG

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in

High Integrity Systems

Faculty 2: Computer Science and Engineering

December 28, 2020

Declaration of Authorship

I, Xhoni ROBO, declare that this thesis titled, "Finite Difference Time Domain Method Implementation in C++" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

_____

Date:

_____

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Abstract

Faculty 2: Computer Science and Engineering

Master of Science

Finite Difference Time Domain Method Implementation in C++

by Xhoni ROBO

This thesis is the final documentation for a project in the Master of Science degree of High Integrity Systems supervised by Prof. Dr. Peter Thoma and Prof. Dr. Egbert Falkenberg. In this thesis, the author will explain the basics of electromagnetic simulation and demonstrate a simple application that will produce both electric and magnetic field data, which can then be visualized through the help of third party applications such as Paraview[15].

This thesis is heavily focused on theoretical aspect of such an application, and as such the code will be simplistic and not use any advanced external libraries not included by default in the basic C++ package. The application will not have a User Interface (UI), therefore the only way to customize the initial values of the code variables would be through an Integrated Development Environment (IDE) that can handle C++, such as Eclipse [9]. The benefit of not relying on any external open source libraries is the ability for this code to be used by any machine regardless of operating system and easy integration into applications that need such simulations.

Alongside this document, the project also included the code files found in the GitHub repository[17]. As the base LaTeX template of this thesis was found online[19], these files are also included, with the license allowing viewing and modification so long as it is for a non-commercial use. After the official deadline of January 5th 2021, this project will be considered complete and no further changes will be made.

Keywords: *Finite, Time, Domain, Difference, C++, High, Integrity, Systems, 1D, 2D, 3D*

Acknowledgements

First and foremost I would like to thank my academic supervisor, Prof. Dr. Peter Thoma. It is only due to his patience, perseverance, and willingness to spend his time aiding me, that I was able to complete this project. Even before that, I would like to thank him for his course of Simulation Methods in the High Integrity Systems M.Sc. degree, that convinced me that such a subject would make an interesting thesis for me.

I would also like to the Prof. Dr. Falkenberg, not only for his assistance throughout my higher academic studies, but also because I would not be able to officially start working on this thesis without his acceptance.

Thank you to my friends, who although far away have helped me keep my spirits high during a rather dark year not just for me, but for the world as a whole. It would have been difficult to push through to the end of this degree otherwise, if not impossible.

And lastly, my deepest thanks to my parents, who taught me discipline, and also acceptance. They did their best to set me up for a rich academic life, by straining themselves physically, mentally, and economically just so that I could have the best possibilities available to me. It will take an eternity to repay them back for their sacrifices, but hopefully this is, at the very least, a step in the right direction.

Sincerely,
Xhoni Robo

Contents

List of Figures

List of Abbreviations

FDTD	**F**inite **D**ifference **T**ime **D**omain (Method)
E	Electric Field Intenensity
D	Electric Displacement (Electric Divergence)
H	Magnetic Field Intenensity
B	Magnetic Induction (Magnetic Divergence)
J	Current Density
ρ	Density of charge
EMF	**E**lectro**m**otive **F**orce
FEM	**F**inite **E**element **M**ethod
FIT	**F**inite **I**ntegration **T**echnique

Physical Constants

Vacuum permeability	$\mu_0 = 1.256\,637\,062\,12(19) \times 10^{-6}\,\mathrm{H\,m^{-1}}$
Vacuum permitivity	$\epsilon_0 = 8.854\,187\,812\,8(13) \times 10^{-12}\,\mathrm{F\,m^{-1}}$
Impedance of Vacuumn	$Z_0 = 376.730\,313\,668(57)\,\Omega$

1 Introduction

As humanity strives to better understand the world and universe around it, the physical limitations of our species become more and more apparent. While we have made considerable progress in our struggles to move forward, such as being able to record the movement of a light particle on camera despite it being the fastest moving object we know of so far[20], or being able to capture an image of a black hole[12], such achievements would not have been possible if our scientists did not have realistic expectations of how they should approach these challenges, or the expected results.

In order to achieve what they have, scientists needed to first understand the phenomena they were studying: the light having the particular properties of both particle and wave and the ability of black holes to distort space around them. All of this would not have been possible without simulations.

This thesis is going to describe the implementation of a C++ algorithm that uses the FDTD method to simulate how an electromagnetic field behaves in vacuum. The goal of the app is to be able to generate data that can then be used by a third party visualization program. This project features three standalone implementations, for one-dimensional, two-dimensional, and three-dimensional cases.

The basis for all of these calculations are the well known Maxwell Equations, which govern all electromagnetic phenomena. They will be used alongside the FDTD method to create update equations that will run in a loop for a certain amount of time. The initial values, domain size, domain environment,

and simulation time can all be changed in code. To start off the simulation, we will need a starting impulse. For this project we will adapt the Gaussian pulse equation to our needs and use it to add an excitation to our simulation that would otherwise be static.

At the end, we will conclude our findings and discuss uses for this application as well as further improvements. Anyone that would like to use this project as a basis for their own work should take a look at the A should they have any issues.

1.1 Electromagnetic Simulations

With the fast development of technology came new opportunities for gaining a better understanding of vast natural phenomena. We will be focusing on one of them, that being Electromagnetic Wave Propagation.

As one can imagine, analyzing electromagnetic fields through plain observation is near impossible, with only a few exceptions[1]. Even if we supposed that it was possible to easily achieve an acceptable amount of information from observing experiments, the cost and quality of the resulting data would mostly be of scientific use, with little to no practical use whatsoever. Considering that electromagnetic waves are widely used in almost every industry, either as part of the building process or as a finished product, having data that cannot be used practically does not help.

That is why, thanks to the progress made in the computational capabilities of computers so far and the use of the theories and formulas gathered from past scientific endeavors, one can create data that is a close approximate of reality. Both shall be discussed in this chapter, but we cannot proceed without first going into what is believed by scientists to be the equations that govern large-scale electromagnetic phenomena[18]: Maxwell Equations.

1.1.1 Maxwell Equations

As mentioned above, Maxwell Equations are believed to dictate the behavior of all kinds of electromagnetic fields at a macroscopic level. These equations are the following:

$$\vec{\nabla} \times \vec{E}(\vec{r},t) = -\frac{\partial \vec{B}(\vec{r},t)}{\partial t} \tag{1.1}$$

$$\vec{\nabla} \times \vec{H}(\vec{r},t) = \vec{J}(\vec{r},t) + \frac{\partial \vec{D}(\vec{r},t)}{\partial t} \tag{1.2}$$

$$\vec{\nabla} \cdot \vec{B}(\vec{r},t) = 0 \tag{1.3}$$

$$\vec{\nabla} \cdot \vec{D}(\vec{r},t) = \rho(\vec{r}) \tag{1.4}$$

To give a brief explanation over the meaning of each equation:

Equation 1.1 explains the effects of the electric field $\vec{E}$ on the rate of change of the magnetic induction $\vec{B}$. This can also be referred to as the equation of electromagnetic induction, where the right hand side is the EMF or voltage and the left hand side is the magnetic flux. To those who study this particular area of physics, this equation will seem familiar, because it was derived from Faraday's law of induction.

Equation 1.2 is also known as Ampère–Maxwell law, because although it originated from Ampère, the current form was derived by Maxwell to include the magnetic current density $\vec{J}$. The equation explains the effects of the magnetic field on the electric current.

Equation 1.3, otherwise known as Gauss's law for magnetism, talks about the divergence of the magnetic field. According to this law the divergence is always 0. What this means is that in a magnetic field there is no such thing as a source (positive divergence) or a sink (negative divergence), rather a magnetic field can be more closely compared to a closed loop that flows in

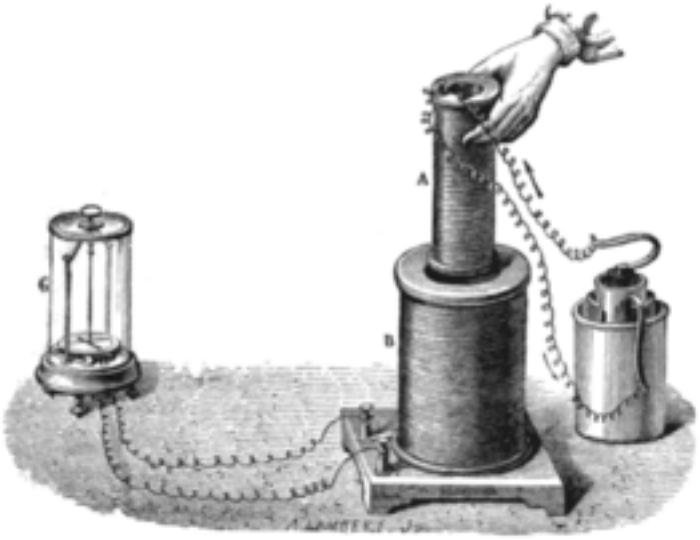


FIGURE 1.1: Faraday's Experiment,[16]
which resulted in the law that was later used by Maxwell in making his equation.

one direction. That is why every magnet that we know of has two poles. To translate this into something more easily understandable, it basically means that, if we were to pick any subset of the area of a magnetic field, no matter what area we pick, we would have vector fields going inside this area, and vector fields going outside in equal number. A simple representation of this rule can be seen in 1.2, where it can be noted that the number of vectors heading towards the north pole are equal to the vectors going outside of it. Interestingly enough, if the bar magnet were to be cut in half, then the result would be two smaller bar magnets with 2 poles each and the exact same vector field.

Equation 1.4, is the main Gauss law for electric currents. It looks rather similar to his law of magnetism on the left hand side; the previous magnetic divergence is now replaced with the electric divergence. The bigger difference is the ρ on the right hand side, which is the density of charge of the electric field. In basic terms, it means that the divergence of the electric field is equal to the charge density for that point.

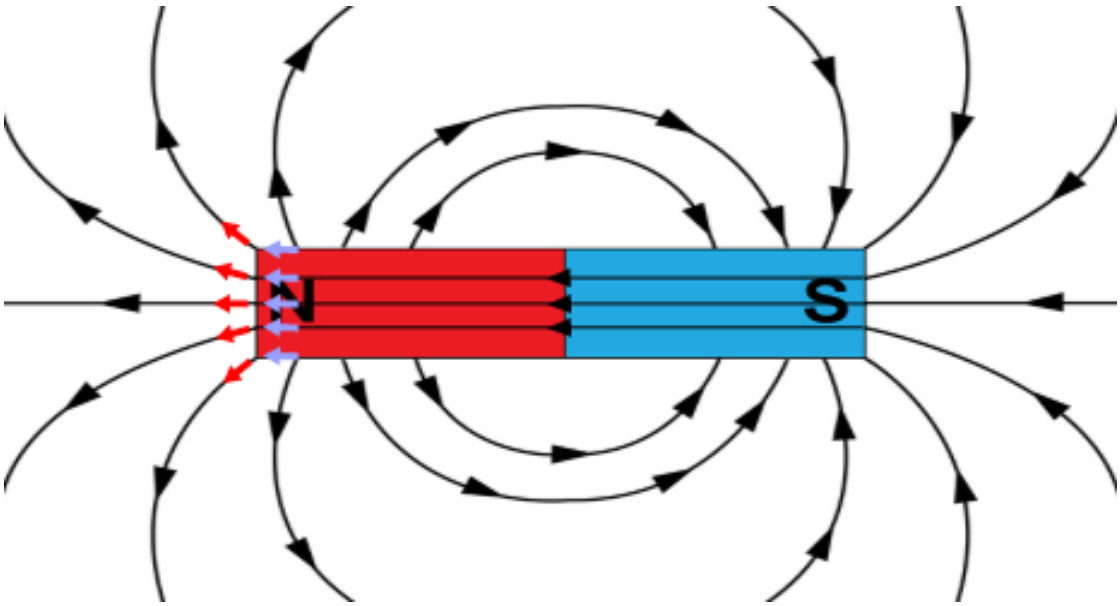


FIGURE 1.2: A crude representation of magnetic divergence through the use of a bar magnet.

For the above equations, we also have the following material relations which will be useful later on:

$$\vec{D}(\vec{r},t) = \epsilon(\vec{r}) \cdot \vec{E}(\vec{r},t) \tag{1.5}$$

$$\vec{B}(\vec{r},t) = \mu(\vec{r}) \cdot \vec{H}(\vec{r},t) \tag{1.6}$$

The equations above are shown in their derivative form, but they can also be shown as their integral equivalent. There is no difference in implementation, regardless of which form is used.

1.1.2 Solving the Wave Equation

Using the formulas above, we can derive the electromagnetic wave equation, which is needed to proceed with the implementation further. Firstly for convenience, we will assume that the environment is the vacuum of space. Secondly, we will also assume that there is no charge in this space. This means that $\epsilon(\vec{r}) = \epsilon_0$ and $\mu(\vec{r}) = \mu_0$. This allows us to simplify the above equations and give us the numerical solution for the wave equation

$$\Delta\vec{E}(\vec{r},t) - \frac{1}{c^2}\frac{\partial^2\vec{E}(\vec{r},t)}{\partial t^2} = \mu_0\frac{\partial\vec{J}(\vec{r},t)}{\partial t}, \tag{1.7}$$

where:

$$c = \frac{1}{\sqrt{\mu_0\epsilon_0}}$$

This equation is useful because it will be used to derive the formulas that are going to be needed for the simulations. Going forwards, this will be used as a basis to adapt our solution to every environment, regardless of dimensionality. From this point, there are many ways to proceed. Some of the most notable are the Finite Element Method (FEM), the Finite Integration Technique (FIT), and the Finite Difference Time Domain Method (FDTD). All of them are also known as Approximation Methods.

FEM is a well known numerical method used to obtain an approximation for a given boundary value problem. The basic principle is to divide a system into smaller subsets, called finite elements (hence the name), which are much simpler to solve. This is done by discretizing the given space for each of its dimensions, and then constructing a mesh of the object. As a result, from the initial boundary value problem we get a system of equations that are further used to approximate each singular simplified function over the given domain. These equations are then compiled together into a system of equations that is then used to model the initial problem. The solution is then approximated by solving this system and minimizing the error function.[13]

The finite integration technique (FIT) is a bit more straightforward. It can help numerically solve electromagnetic field problems by disctretizing in both the time and frequency domain. The first to introduce this technique was Thomas Weiland in 1977.[21] It has later seen continuous improvements

and can now cover all electromagnetic problems and applications. This approach works by using the Maxwell equations above and applying their integral form to a set of staggered grids (e.g. Cartesian grid). This allows for a memory efficient implementation as well as giving the ability to handle different boundary conditions and variable material properties.

Finally we have a well known computational electromagnetic technique used for approximation, the Finite Difference Time Domain Method. It is arguably the easiest method out of the three to understand and implement, which is surprising when considering the capabilities it has in solving wave equations.

This simplicity is also the reason it was chosen for this thesis, as it is the only technique that one person can realistically implement by themselves in a reasonable time frame.[8] As the name implies this is a time-domain method, meaning that a wide range of frequencies can be covered with a single simulation run. The only caveat is that the time step needs to be small enough to not cause any instabilities in the system.

1.2 Finite Difference Time Domain Method

FDTD was first proposed by Kane Yee in a 1966 paper and was initially called the Yee Algorithm, taken from the author's name. It was modified later from further research, resulting in the modern version that is widely used to this day. This method can be basically summarized in the following steps:

1. Replace the derivatives from the Maxwell Equations with finite differences

2. Discretize the space and time of the domain, while staggering the electric fields from the magnetic ones (e.g. by half a time step, and spatially by using a different axis)

3. Get the update equations
4. Use the update equations to get the future step for magnetic and electric fields
5. Repeat the step above throughout the set duration

The most important steps are 2 and 3, as the rest are relatively easy to do and it is highly unlikely for any mistakes to occur, especially once this is implemented in code. In the next section we will go quickly through the first step, which will the basis that will be used moving forward. Steps 2-5 are implementation specific and vary depending on the domain. As such, they will be explained for each scenario in their respective chapters.

1.3 FDTD Implementation

Before beginning with the discretization, we mentioned previously that Maxwell's Equations can be shown in both their differential form and their integral form:

$$\oint E \cdot ds = -\frac{d}{dt} \int B \cdot dA \tag{1.8}$$

$$\oint H \cdot ds = \int \frac{dD}{dt} \cdot dA \tag{1.9}$$

$$\int D \cdot dA = \int \rho dV \tag{1.10}$$

$$\int B \cdot dA = 0 \tag{1.11}$$

The material relations would look as follows:

$$D = \epsilon \cdot E \tag{1.12}$$

$$B = \mu \cdot H \tag{1.13}$$

By plugging 1.13 and 1.12 into equations 1.8 and 1.9 respectively, we get:

$$\oint E \cdot ds = -\frac{d}{dt} \int \mu \cdot H \cdot dA \tag{1.14}$$

$$\oint H \cdot ds = \frac{d}{dt} \iint \epsilon \cdot E \cdot dA \tag{1.15}$$

Equations 1.14 and 1.15 are going to be used later on during the implementation to derive the specific update equations. With all that, the general theoretical part is finished. For this project, we will need to implement the above functions into a program that can generate approximate data on electromagnetic waves, which will be discussed in the next section.

1.3.1 Application Requirements

The result of this project is not only this documentation, but also a relatively simple program that can be used either as is, or implemented into a bigger project with minor adjustments. As such, the resulting application must adhere to the following requirements:

- Allow for the generation of electromagnetic data in a set environment (e.g. vacuum)
- Have a smooth impulse to start off the simulation
- Use only the default C++ libraries, no external dependencies
- Support one dimensional, two dimensional, and three dimensional domains.
- Be simple and compact, so that it can be adapted to a bigger application if necessary

The first requirement is understandably the most basic functionality, the goal of the whole project. This data will be generated from scratch, using the

environment variables of permittivity ϵ and permeability μ. These values can be changed depending on the environment, which can be vacuum, copper, etc. The examples here will use the values for free space ϵ_0 and μ_0. Also, these values are going to be constant throughout the simulation, meaning we will have one material through out the whole domain.

Without a starting impulse, there would be nothing to simulate, as the data would simply keep its initial value of zero. If we were to add an immediate pulse of an arbitrary value to a single point, it would prove sufficient. However, this would result in a sudden explosion of a single electromagnetic pulse that would simply travel along the domain as a single point, thus providing us with a near useless visualization once the data is put into an appropriate program.

Instead, we can use a Gaussian pulse excitation in the middle of the domain, and have it propagate throughout it. This is ideal because we are using reflective boundaries, meaning the pulse will bounce back and forth between each boundary without any loss. If we were to use absorbing boundary conditions, such an excitation would eventually lead to the waves disappearing completely. For such cases, a steady sinusoidal excitation that keeps going would prove more interesting.

We will be using a Gaussian pulse for our example. The generic formula is given below:

$$f(t) = \alpha e^{-\beta(t-T_\varepsilon/2)^2} \tag{1.16}$$

where

$$\beta = -(\frac{2}{T_\varepsilon})^2 \ln \varepsilon \tag{1.17}$$

A good value for sufficient smoothness would be $\varepsilon = 0.001$, but this is heavily dependent on the implementation.

For the third requirement, the reason why we would want to avoid the use of libraries that are not included by default in C++ is that such libraries could make the program dependent on the operating system that the machine is running, PATH variables, etc. In short, it would complicate the setup to run such a program too much, possibly voiding the last requirement. On top of that, in insisting on using only the very basics that C++ has to offer, the resulting code will be easier to relate to the formulas that are shown in this thesis, since all the code will be visible at all times. With that said, various improvements could be made if the use of external libraries is allowed. That will be discussed in more details in the Conclusion.

Fourth, we want the application to support anything from one to three dimensions. While only a 3D simulation would be realistically desired, for a thesis the 1D and 2D scenarios are also interesting to study. Not only that, but the 1D application leads smoothly to the development of the 2D application, which in turn leads to the smooth development of the 3D application. This progression resulted in having a standalone program for each scenario, rather than one for all of them. Rather than unify these applications, they were intentionally left as separate programs in the end, because it helps in complying with the last requirement.

Lastly, after going through each of the previous requirements, this one is rather self-explanatory. To begin with, simple and compact code that works well standalone is one of the most important programming practices that developers should follow. While this program's goal is to simply be used as a demonstration of such an implementation, it should also be easily modifiable and adaptable, so that it can provide a good basis that can be used by larger applications that include far more features.

With that said, we will have to go through certain limitations that plague all computers simply due to their nature. Since these limitations cannot be bypassed as of yet, we can never achieve an exact, perfectly realistic simulation. Thus, it is good to keep them in mind while developing such applications.

1.3.2 Computational Limitations and Inaccuracies Explained

Programmers can use programming languages to instruct computers to perform certain commands in certain orders, thus creating applications. However, these instructions are not what the computers use to dictate what should happen. These programming languages are decoded by the interpreter of choice, and then passed down in the form of a lower level language. This process can occur more than once too, until we get to the smallest unit a computer can have: a bit.

A bit is simple; it can have either a value of zero or one. Instructions are basically translated into many such bits, thus making what is basically computer language. Each instruction could be translated into millions of bits, but that by itself would not cause issues normally. The issue is that, no matter how powerful the computer is or how much memory it has, these bits are finite. As such, they present limitations when dealing with infinite concepts.

One such concept is infinite numbers. To best explain this, let us use an example: summing thirds into a whole. If a person was to be asked to add $\frac{1}{9}$ nine times, they would do the following:

$$\frac{1}{9}+\frac{1}{9}+\frac{1}{9}+\frac{1}{9}+\frac{1}{9}+\frac{1}{9}+\frac{1}{9}+\frac{1}{9}+\frac{1}{9}=1\,,$$

which we know is correct. However, when we run the following code snippet:

```
double a = 1.0/9.0;
if (a + a + a + a + a + a + a + a + a == 1.0) {
        cout << "Equal to 1";
} else {
        cout << "Not equal";
}
```

we would get the following result (Fig. 1.3):

```
Not equal

...Program finished with exit code 0
Press ENTER to exit console.
```

FIGURE 1.3: According to our code, the total sum is not equal to 1.0, even though it should be.

What is happening is that we are asking a computer to store the infinite number $1/9 = 0.\overline{1}$ using a finite amounts of bit. Depending on the data type we use, we can have anywhere from 8 bits of precision, to 2^8 bits. However, that is still a finite amount, and no matter what we do the resulting value will be truncated to 0.1111...1, resulting in a sum of $0.9999...9 \neq 1.0$. That is why any simulation, regardless of the computer or the method used, will always be an approximation of reality. When performing thousands of calculations, this margin of error increases further. Despite that, these approximations are enough to give us a good idea of what to expect.

2 FDTD - One-Dimensional Scenario

In this chapter, we will go more in depth into developing an application that can generate electromagnetic data in a one-dimensional domain. In the previous chapter, we mentioned a series of steps to implement FDTD, and that a part of them depend on the particular implementation. A keen eye will notice moving forward, that while the code will not change too much, each implementation deserves a different approach in the theoretical sense.

2.1 1D Discretization

In the previous chapter, we ended up with the equations 1.14 and 1.15. They will now be used to do a FDTD discretization for the one-dimensional electromagnetic wave scenario. At the end of this section, we will have the update equations that we will use in our loops. Before moving on, we must first briefly discuss Transverse modes. A transverse mode is the type of pattern that an electromagnetic field, which is perpendicular with respect to the direction of the wave's propagation, has. For electromagnetic waves, the most relevant modes are TE (Transverse Electric) and TM (Transverse Magnetic). In our scenario, we will be using TEM mode for discretization, meaning neither electric nor the magnetic field are moving in the direction of propagation.

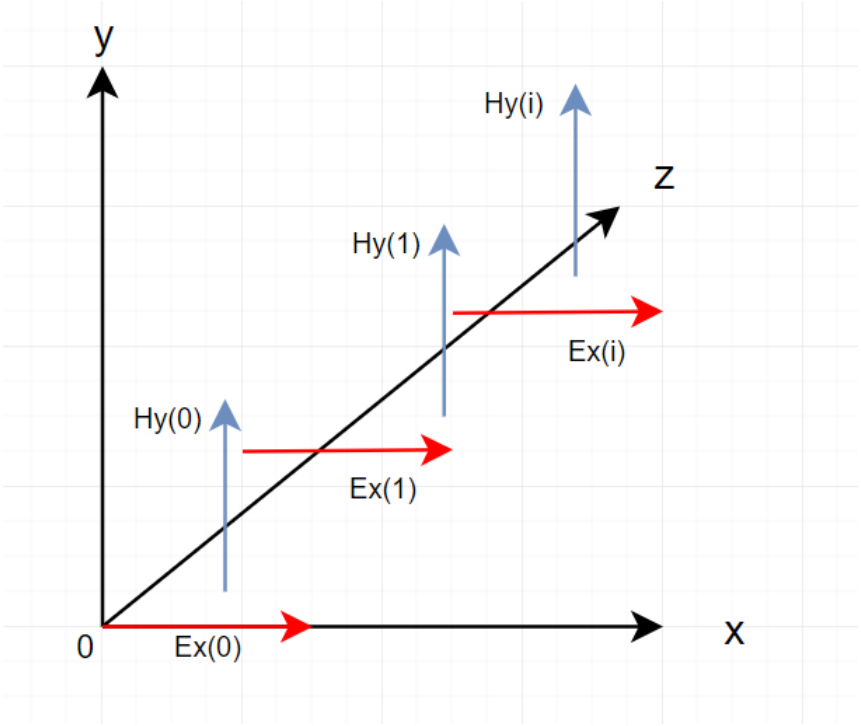


FIGURE 2.1: The spatial and temporal shift of a one-dimensional electromagnetic scenario.

2.1.1 Spatial and Temporal Shift

The first step of the FDTD discretization is a shift in spacetime of the electric and magnetic fields. This spatial shift is shown in Figure 2.1.

The electric vectors E are parallel to the x axis, while the magnetic vectors H are parallel to the y axis. The z axis in this case is the direction of the wave propagation.

2.1.2 Electromagnetic Curls

At first, the way that the vectors in Figure 2.1 are spaced out might seem odd. They look this way because they are part of each other's vector curl. We have two such curls in our one-dimensional scenario: one for the electric field vectors Ex,and one for the magnetic field vectors Hy. These curls are necessary for the update equations, because they explain the relationship between the

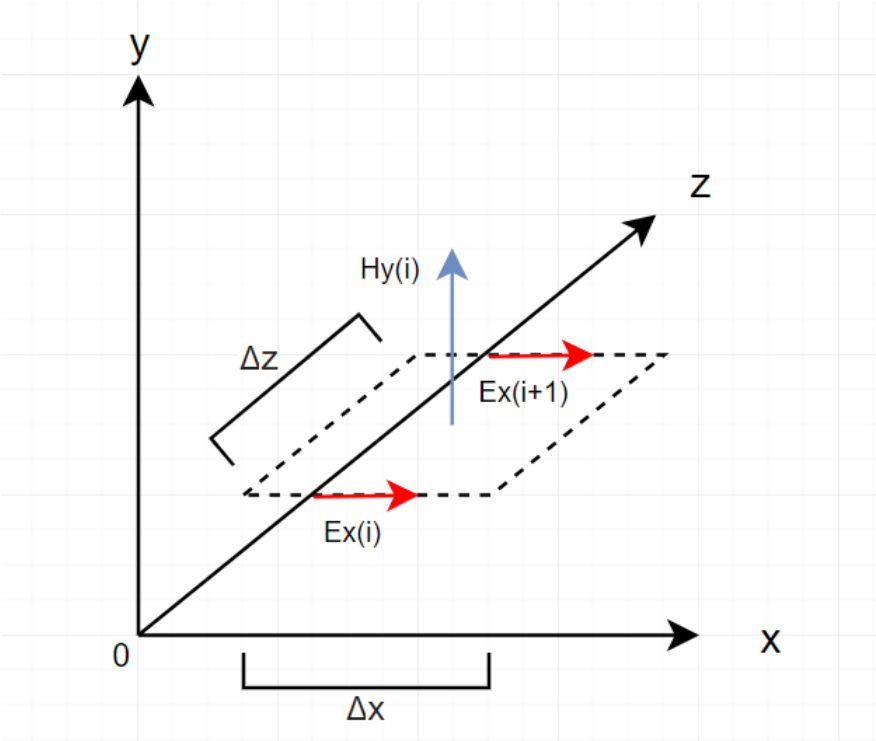


FIGURE 2.2: A graph showing the one-dimensional curl around the vector $H_y(i)$.

electric fields and the magnetic ones.

Figure 2.2 shows the curl around the magnetic vector $H_y(i)$. We can use this curl and plug it in to the original equation 1.14:

$$\oint E \cdot ds = E_x(i) \cdot \Delta x + E_z \cdot \Delta z - E_x(i+1) \cdot \Delta x - E_z \cdot \Delta z \tag{2.1}$$

Since we do not have an Ez vector in the one-dimensional scenario, $Ez \cdot \Delta z = 0$. Therefore we can simplify equation 2.1 to:

$$\oint E \cdot ds = E_x(i) \cdot \Delta x - E_x(i+1) \cdot \Delta x \tag{2.2}$$

On the left hand side of equation 1.14 we have:

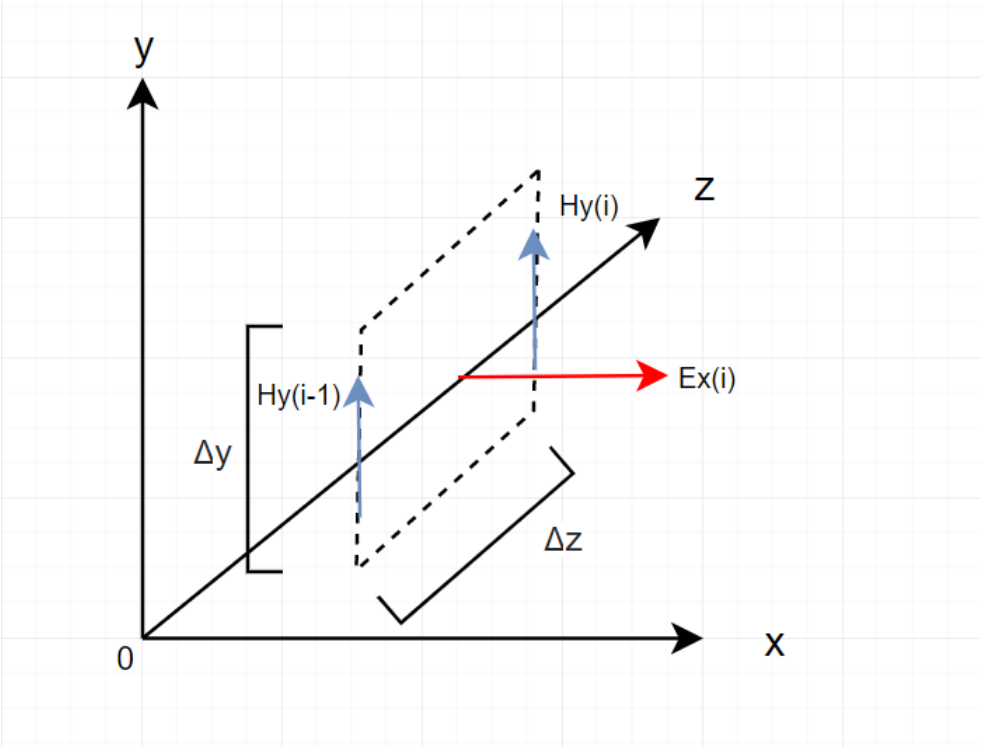


FIGURE 2.3: A graph showing the one-dimensional curl around the vector $E_x(i)$.

$$\int \mu \cdot H \cdot dA = \mu \int H \cdot dA = \mu \cdot H_y(i) \cdot \Delta y \cdot \Delta z \tag{2.3}$$

By combining 2.2 and 2.3, we get:

$$\Delta x(E_x(i) - E_x(i+1)) = -\frac{d}{dt}(\mu \cdot H_y(i) \cdot \Delta y \cdot \Delta z) \tag{2.4}$$

$$E_x(i) - E_x(i+1) = -\mu \cdot \Delta z \cdot \frac{dH_y(i)}{dt} \tag{2.5}$$

We can do the same thing for the curl of the electric vector E_x, shown in Figure 2.3, as follows (Similar to the previous scenario, $H_z = 0$):

$$\oint H \cdot ds = H_y(i) \cdot \Delta y - H_y(i-1) \cdot \Delta y = \Delta y(H_y(i) - H_y(i-1)) \tag{2.6}$$

$$\iint \epsilon \cdot E \cdot dA = \epsilon \cdot E_x(i) \cdot \Delta z \cdot \Delta y \tag{2.7}$$

Combining 2.6 and 2.7:

$$\Delta y(H_y(i) - H_y(i-1)) = \frac{d}{dt}(\epsilon \cdot E_x(i) \cdot \Delta z \cdot \Delta y) \tag{2.8}$$

$$H_y(i) - H_y(i-1) = \epsilon \cdot \Delta z \cdot \frac{d}{dt} E_x(i) \tag{2.9}$$

With those equations, we can now use the leapfrog time scheme to stagger our components along the time axis (Fig. 2.4).

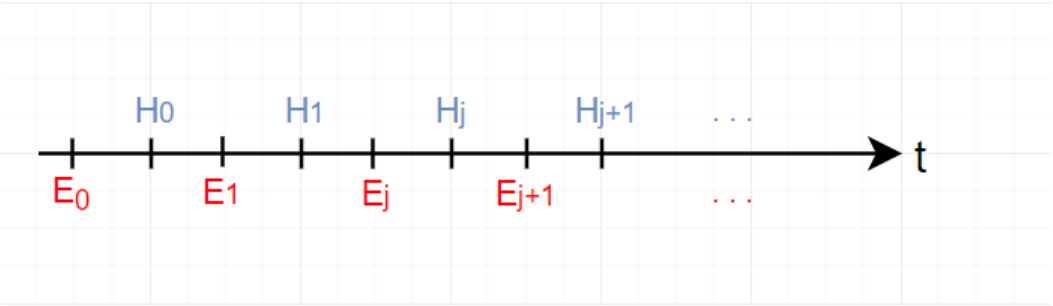


FIGURE 2.4: Staggering the components along the time axis t using the leapfrog time scheme

We will use the following indexing scheme:

$$E_{i,j} = E(i \cdot \Delta z, j \cdot \Delta t) \tag{2.10}$$

$$H_{i,j} = H((i + \frac{1}{2}) \cdot \Delta z, (j + \frac{1}{2}) \cdot \Delta t) \tag{2.11}$$

Using 2.10 and 2.11 we get the following time derivatives:

$$\left.\frac{dE_x}{dt}\right|_{\substack{t=(j+\frac{1}{2})\Delta t \\ z=i\cdot\Delta z}} = \frac{E_{x^{i,j+1}} - E_{x^{i,j}}}{\Delta t} \tag{2.12}$$

$$\left.\frac{dH_y}{dt}\right|_{\substack{t=j\cdot\Delta t \\ z=(i+\frac{1}{2})\Delta z}} = \frac{H_{y^{i,j}} - H_{y^{i,j-1}}}{\Delta t} \tag{2.13}$$

Going back to equation 2.5 we get:

$$E_x(i\cdot\Delta z) - E_x((i+1)\cdot\Delta z) = -\mu\cdot\Delta z\cdot\frac{dH_y((i+1/2)\Delta z)}{\Delta t} \tag{2.14}$$

Evaluating at $t = j\cdot\Delta t$ gives us:

$$E_{x^{i,j}} - E_{x^{i+1,j}} = -\mu\cdot\Delta z\cdot\frac{H_{y^{i,j}} - H_{y^{i,j-1}}}{\Delta t} \tag{2.15}$$

Finally, by solving for $H_{y^{i,j}}$ we get our update equation for the magnetic element:

$$H_{y^{i,j}} = H_{y^{i,j-1}} - \frac{\Delta t}{\mu\cdot\Delta z}(E_{x^{i,j}} - E_{x^{i+1,j}}) \tag{2.16}$$

In a similar way, we can get the update equation for the electric element by starting from the equation 2.9. The result is:

$$E_{x^{i,j+1}} = E_{x^{i,j}} + \frac{\Delta t}{\epsilon\cdot\Delta z}(H_{y^{i,j}} - H_{y^{i-1,j}}) \tag{2.17}$$

Finally, we are ready to proceed into the code implementation.

2.2 C++ Implementation

Calculating the update equations is by far the most difficult part of implementing FDTD. However, translating equations into code is not always straightforward. Before we begin the implementation, we need to prepare our coding environment. For this project, the author is using the Eclipse IDE for C++ Development[9]. After creating a new project, we will start off with an empty skeleton that features the *main()* method. Due to not needing any other classes, this is where we will place our code. Before that, we will need to include some packages so that we can use their methods, data types, and variables.

```
#define _USE_MATH_DEFINES

#include <iostream>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <cmath>
#include <vector>
#include <string>

using namespace std;
```

To shortly explain what each line does:

- **cmath, math.h**[2] - Packages that allow the use of various helpful math related commands and constants. Requires having `#define _USE_MATH_DEFINES` set in order for everything to work properly
- **iostream, stdio.h**[3] - Allows for the usage of input and output stream objects and commands. In the 1D implementation, we will print our data to the console by using `cout`

- **stdlib.h**[4] - Provides helpful data types, especially when dealing with vectors
- **vector**[5] - Arrays in C++ are not dynamic. Once populated, they can no longer be modified. That is why for this implementation we need to use vectors.
- **string**[6] - Includes the string datatype. A string is basically an array of characters. Not only can we use this to format our output more easily, it can also be very useful for printing debugging messages.

Next, we will need to initialize some variables. To begin with, we need to set the permittivity ϵ and permeability μ of the domain that we are going to simulate. For our scenario, we are going to use ϵ_0 and μ_0, which are the values for vacuum.

```
const double permitivity = 8.854e-12;  // vacuum permitivity
const double permeability = 1.256e-6;  // vacuum permeability
```

These numbers are normally infinite, therefore we are already introducing inaccuracies into our simulation. For greater precision, we could include more decimals, though the difference would not be easily visible. Please note that although the units are not mentioned, we are using SI units for all calculations. This is important, as using the wrong unit could make the simulation inaccurate at best, or utterly unstable at worst.

```
double L = 5;
int N = 200;
int iterNum = 800;
double deltaZ = L / N;
double deltaT = (deltaZ * sqrt(permitivity*permeability));
```

L is the length of the domain in SI units, namely meters. **N** is the number of steps, or in the one-dimensional scenario the number of times we have

sliced our domain. **iterNum** is the number of iterations. **deltaZ** (Δz) is the difference between the current step and the next one in meters. This can be set to anything, but we want to evenly split our domain and the best way to do so would be to set this to anything that is $x \cdot L/N$.

Special attention should be brought to **deltaT** (Δt). If this value is too big, the simulation will quickly grow unstable. While picking any arbitrary value would work, so long as it is small enough, we can choose a value that will always work through the equation below:

$$\Delta t = \Delta z \cdot \sqrt{\epsilon \mu} \tag{2.18}$$

This is the one-dimensional version of this equation. We will see how it changes later on when discussing the two-dimensional and three-dimensional scenarios. Now that our testing environment is ready, we will need to initialize the variables that will be used in our loops.

```
// variables needed for Gaussian Pulse excitation
double eps = 1e-3;
double Teps = 50 * deltaT;
double beta = -(pow((2/Teps), 2) * log(eps));



vector<double> E;
vector<double> H;
vector<double> tE;
vector<double> tH;
```

E and **H** are the electric and magnetic field vectors that will hold our data for each time step. For the one-dimensional scenario, in order to display the simulated data we will use what is called a time graph. A time graph is a graph of the values that an arbitrary point n has during the simulation. We

will choose the midpoint of the domain $N/2$ for our scenario, and the vectors **tE** and **tH** will contain the data of the electric and magnetic time graph respectively. We also initialize some helpful values that will be used by the Gaussian pulse excitation, with the equation 1.16 mentioned in chapter 1: **eps** (ε), **Teps** (T_ε), and **beta** (β), with equation 1.17 that was discussed in the same chapter.

We are done setting up the variables we will need for the simulation, therefore it is time to actually move on to the *main()* method where we will implement the FDTD algorithm. First, we will need to populate our magnetic and electric field vectors, since we have not done so yet. Since we want our simulation to have no initial charge in it, we will have to set everything to 0. Luckily there is a quick way to do so:

```
E.assign(N, 0);
H.assign(N, 0);
```

This will push N zeros to our vectors. Now we will need to start looping for

```
int i = 0; i < iterNum; i++
```

values. We will also start off our Gaussian pulse here, by applying it to the beginning of the electric vector like so:

```
double t = i * deltaT;
double gamma = Teps / 2;


E[0] = exp(-(beta * pow((t - gamma), 2)));
```

Afterwards, we use the update equations that we derived above to get the values for the magnetic and electric fields:

```
// loop for values
for (int z = 0; z < N-2; z++) {
        H[z] = H[z] - (deltaT / permeability / deltaZ) * (E[z] - E[z+1]);
```

```
}


for (int z = 1; z < N-1; z++) {
        E[z] = E[z] + (deltaT / permitivity / deltaZ) * (H[z] - H[z-1]);
}
```

Please note that the starting index of a vector is zero, therefore when adding N values to our vectors, the index of the last value is going to be $N-1$. In our loops, we need to be certain that we never surpass this number. For the magnetic vector H loop, we need to make sure that the loop stops when $z+1 = N-1 \Leftrightarrow z = N-2$. Alternatively, we could also populate $N+1$ values, and extend our domain one unit past our boundaries.

After the above update loops, but while still inside the main loop, we will store the middle values into the respective time graph vectors for each time step:

```
// time graph
tE.push_back(E[100]);
tH.push_back(H[100]);
```

As mentioned previously, we can pick any point for this and it would still work. This is also another reasons why vectors are incredibly useful: we did not populate **tE** or **tH** with any values previously, therefore they had a length of zero. However, using the *push_back()* method, we can dynamically add values dynamically, therefore allowing us to change the number of iterations at will.

After running the main loop **iterNum** times, we finally print our values to the console as a comma separated list:

```
cout << "\n\ntE\n";


// print E values
```

```
for (int n = 0; n < iterNum; n++) {
        cout << to_string(tE[n]) + ",";
}

cout << "\n\ntH\n";

// print H values
for (int n = 0; n < iterNum; n++) {
        cout << to_string(tH[n]) + ",";
}
```

The result is shown in figure 2.5.

```
tE
0.000000,0.000000,0.000000,0.000000,0.000000,

tH
0.000000,0.000000,0.000000,0.000000,0.000000,
```

FIGURE 2.5: The console output of our FDTD one-dimensional algorithm

Now that we generated the data, we can proceed by choosing how to visualize it.

2.3 Data Visualization

Generating the data is the difficult part. In order to visualize it into a form that is easily understandable by humans, we can use a variety of programs. An easy solution for this is Microsoft Office Excel[14]. While this software requires a license, there are viable alternatives to it that should be able to achieve the same results.

For starters, we will need to copy the comma separated list that is in our console output to the first column, in the second row for the electric data and the third row for the magnetic data. This will populate the first cell for both

rows. After that, we can go to **Data > Text to Columns** and then split our one cell list into multiple columns (2.6).

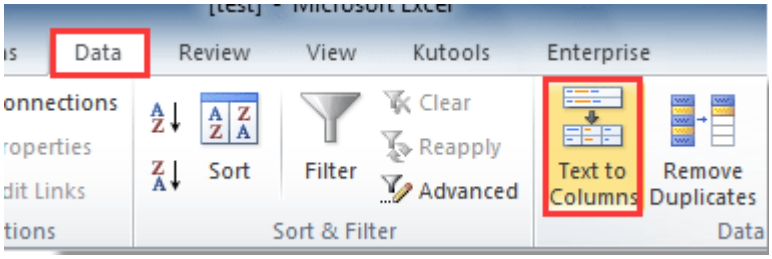


FIGURE 2.6: Transforming data from a single cell containing the whole list, into separate columns for each value.

After doing that for both electric and magnetic values, we can add a row of numbers above them for the timesteps. The result will look like so:

	A	B	C	D	E	F	G	H	I	J	K	L
1	0	1	2	3	4	5	6	7	8	9	10	11
2	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0

FIGURE 2.7: Transforming data from a single cell containing the whole list, into separate columns for each value.

With this, we can visualize the data however we desire. For the purposes of this demostration, we shall use line graphs. In Figure 2.8, we have colored the time graph of the electric field orange, and the magnetic field blue.

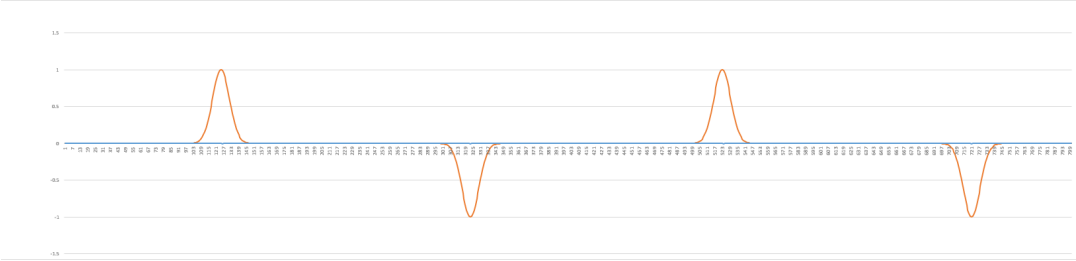


FIGURE 2.8: The time graph of the electromagnetic data generated by our 1D application.

Please note that both lines were shown in the same plane due to convenience. The waves themselves move as they did during the discretization we did

prior: electric waves move in the **X-Z** plane while the magnetic waves move in the **Y-Z** plane. Also, we can note that there is an enormous difference in the values of the electric and magnetic fields. We can easily tell that the electric wave is moving, but it is difficult to notice the magnetic wave movement. Here is one of those values zoomed in (Fig. 2.9):

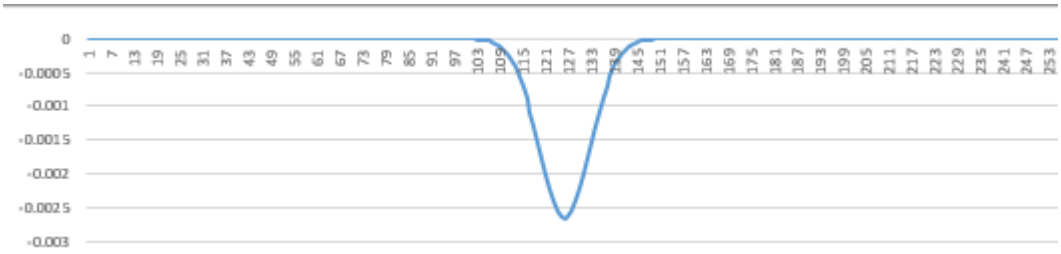


FIGURE 2.9: A snippet of the time graph shown in Figure 2.8 zoomed in

The reason for this difference being so big is due to the medium we chose for our domain: free space or otherwise known as vacuum. Also known as the impedance of free space, this value is commonly known as $Z_0 = 376.730\,313\,668(57)\,\Omega$. This is also the ratio between the electric field and the magnetic one in free space. The value can also be calculated by using the permittivity and permeability:

$$Z_0 = \frac{E}{H} = \sqrt{\frac{\mu_0}{\epsilon_0}} \tag{2.19}$$

With that done, let us take a look at the two-dimensional scenario.

3 FDTD - Two-Dimensional Scenario

Now that we completed the one-dimensional scenario, the following is going to be much easier, as we are going to use the same logic for the most part. For the interest of saving time, redundant information that was covered previously is going to be omitted.

3.1 2D Discretization

In the previous chapter we mentioned the different transverse modes available for electromagnetic waves, but did not go too much in depth as they were not too relevant in that scenario. In a one-dimensional case, it does not matter much which axis we pick for which field, so long as they are perpendicular to which other (something that axes do by default). In the two-dimensional scenario however, the mode we choose is going to dictate our curls.

3.1.1 2D Transverse Modes

For the two-dimensional scenarios, choosing a transverse mode means that the field we chose will have no field in the direction of propagation, meaning the z axis in our case. That means that any vector moving along that axis will have a value of zero for that field. However, the other field can only

have values alongside that direction, meaning that it will be zero for the x and y axis instead. More specifically, each mode features the following field vectors:

- **TE mode** - $\vec{E}_x,\ \vec{E}_y,\ \vec{H}_z$
- **TM mode** - $\vec{E}_z,\ \vec{H}_x,\ \vec{H}_y$

While each mode will have us working with different vectors, the process is roughly the same. For our scenario, we will use TE mode, meaning that $\vec{E}_z, \vec{H}_x, \vec{H}_y = 0$.

3.1.2 2D TE Electromagnetic Curls

Similar to the one-dimensional scenario, each vector will have a curl around it, such as the H_z vector in figure 3.1:

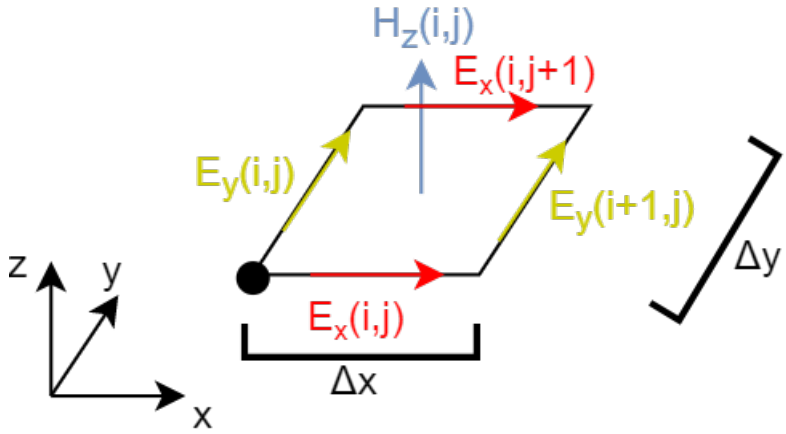


FIGURE 3.1: The curl around the magnetic field vector H_z.

Using a similar indexing scheme as we did previously, we have:

$$E_{i,j} = E(i \cdot \Delta x, j \cdot \Delta y) \tag{3.1}$$

$$\oint \vec{E} \cdot d\vec{s} = -\frac{d}{dt} \iint \mu \cdot \vec{H} \cdot d\vec{A}$$
$$\Rightarrow E_x(i,j) \cdot \Delta x - E_x(i,j+1) \cdot \Delta x + E_y(i+1,j) \cdot \Delta y - E_y(i,j) \cdot \Delta y$$
$$= -\frac{d}{dt}(\mu \cdot H_z(i,j) \cdot \Delta x \cdot \Delta y) \quad (3.2)$$

which leads to

$$\frac{d}{dt}H_z(i,j) = -\frac{1}{\mu} \cdot (\frac{E_x(i,j) - E_x(i,j+1)}{\Delta y} + \frac{E_y(i+1,j) - E_y(i,j)}{\Delta x}) \quad (3.3)$$

If we have a uniform mesh size, meaning that $\Delta x = \Delta y = \Delta z = \Delta s$, we can simplify equation 3.3 to:

$$\frac{d}{dt}H_z(i,j) = -\frac{1}{\mu \cdot \Delta s} \cdot ((E_x(i,j) - E_x(i,j+1) + E_y(i+1,j) - E_y(i,j)) \quad (3.4)$$

For the left hand side, we know that:

$$\frac{d}{dt}H_z(i,j) = \frac{H_z^{new}(i,j) - H_z^{prev}(i,j)}{\Delta t} \quad (3.5)$$

By combining equations 3.4 and 3.5, we get our update equation for the magnetic field vector H_z:

$$H_z^{new}(i,j) = H_z^{prev}(i,j) - \frac{\Delta t}{\mu \cdot \Delta s} \cdot$$
$$(E_x(i,j) - E_x(i,j+1) + E_y(i+1,j) - E_y(i,j)) \quad (3.6)$$

For the electric vectors E_x and E_y, the equations and curls are going to be roughly the same as the ones for the one-dimensional scenario. The curl for E_x can be seen in Figure 3.2:

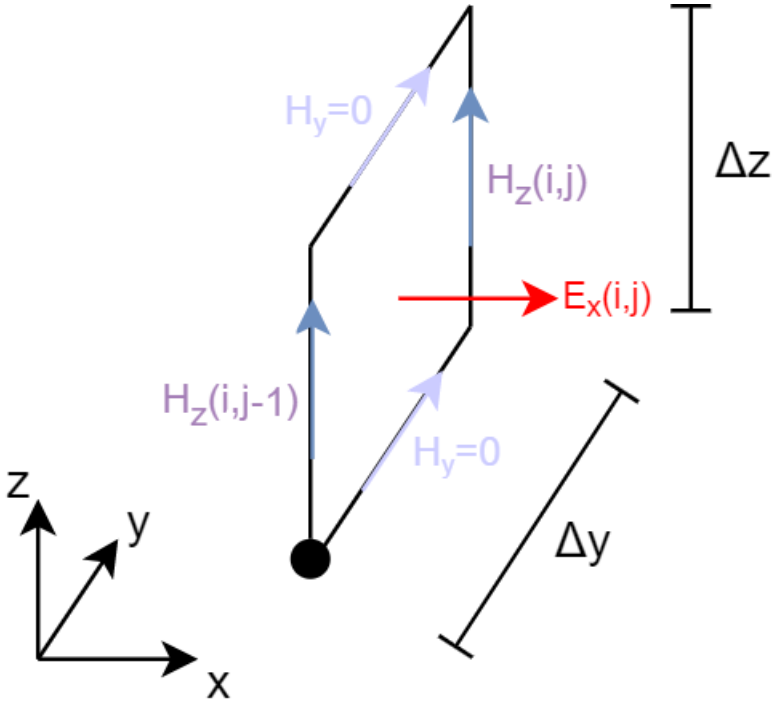


FIGURE 3.2: The curl around the electric field vector E_x.

resulting in the following update equation:

$$E_x^{new}(i,j) = E_x^{prev}(i,j) + \frac{\Delta t}{\epsilon \cdot \Delta s} \cdot (H_z(i,j) - H_z(i,j-1)) \tag{3.7}$$

In a similar fashion, the curls for E_y will be as follows (Figure 3.3),

with a fairly similar equation (note the signs):

$$E_y^{new}(i,j) = E_y^{prev}(i,j) - \frac{\Delta t}{\epsilon \cdot \Delta s} \cdot (H_z(i,j) - H_z(i-1,j)) \tag{3.8}$$

We can now proceed to the code implementation.

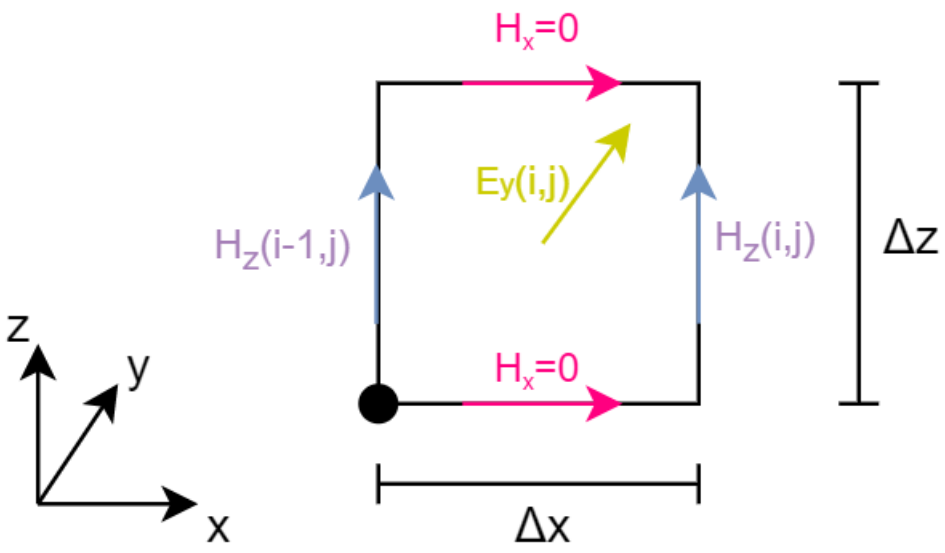


FIGURE 3.3: The curl around the electric field vector E_y.

3.2 C++ Implementation

Starting off with our imports, we are going to ignore the ones we already discussed. We have however, added three new imports:

```
#include <io.h>
#include <fstream>
```

For the two dimensional scenario, we are going to change the way we visualize our data. The end result will be an animation that shows us how the waves propagate in real time. We will get deeper into it once we discuss data visualization, but for now all we need to know is that the output of the two-dimensional implementation is going to be CSV files. Our packages above help us with that:

- **fstream**[7] - Input output stream operations for files. Allows us to create and modify them.
- **io.h**[11] - Header file used by **fstream** among other packages.

We are also going to introduce two new methods to write our electromagnetic data into files:

```
void writeEDataToCsvFile(string filename, vector<vector<double>> Ex,
↪  vector<vector<double>> Ey){

        //          "x","y",Ex,Ey
        //          0,0,Ex[x,y],Ey[x,y]

        ofstream csvFile(filename);
        csvFile << "x,y,z,Ex,Ey\n";

        for (unsigned x = 0; x < Ex[0].size(); x++) {
                for (unsigned y = 0; y < Ex[x].size(); y++) {
                        csvFile << to_string(x) + "," + to_string(y) +
                        ↪  ",0," + to_string(Ex[x][y]) + "," +
                        ↪  to_string(Ey[x][y]) + "\n";
                }
        }

        csvFile.close();
}

void writeHDataToCsvFile(string filename, vector<vector<double>> Hz){

        //          "x","y",Hz
        //          0,0,Hz[x,y]

        ofstream csvFile(filename);
        csvFile << "x,y,z,Hz\n";

        for (unsigned x = 0; x < Hz[0].size(); x++) {
                for (unsigned y = 0; y < Ex[x].size(); y++) {
                        csvFile << to_string(x) + "," + to_string(y) +
                        ↪  ",0," + to_string(Hz[x][y]) + "\n";
```

```
                }
        }


        csvFile.close();
}
```

We will be calling these methods every time we loop, to print a file that contains the respective field's data for that time step. The filename format should be *"X.csv.n"*, where *X* is the field the data is for (E for electric, H for magnetic), while *n* is the time step number. These files are going to get added in the the directory:

```
const string filePath = "./Out/";
```

We must also make some minor changes to the code to adapt it to a two-dimensional environment. For starters, we need to initialize our vectors as two-dimensional vectors. We also need two vectors for the electric field now, one for the *x* axis and one for the *y* axis.

```
vector<vector<double>> Ex(N, vector<double> (N, 0));
vector<vector<double>> Ey(N, vector<double> (N, 0));
vector<vector<double>> Hz(N, vector<double> (N, 0));
```

Normally we would also need to add new variables for our deltas:

```
double deltaX = L / N;
double deltaY = L / N;
double deltaZ = L / N;
```

Since we have a uniform mesh, meaning all the spatial steps have the same dimensions, we are only going to use one of them (*deltaZ*). In the Appendix (A) we are going to discuss how they can be implemented for unequal time step meshes. For now, since we are only using one of them, our Δt will need to be changed to:

```
double deltaT = (deltaZ * sqrt(permitivity*permeability)  * (1/sqrt(2)));
```

Another difference is that we are going to apply our Gaussian pulse excitation in the center of the magnetic field this time. Due to the impedance of vacuum, we are going to promptly reduce the magnitude of this excitation, as otherwise we would get electric values that will be far too large.

```
// reducing the magnitude since in free space
Hz[99][99] = exp(-(beta * pow((t - gamma), 2))) * 10e-4;
```

Starting from the center will give us a nice symmetric propagation towards the edges. In symmetrical scenarios we can save computation time by splitting the domain into equal parts. It is even more helpful with circular symmetry as we can effectively split it into as many parts as we want, calculate the wave propagation for only one part, then replicate those values for the other parts. This will also be discussed in the Appendix, but for now we will move on with our most basic scenario.

By translating the update equations into code, we get the following in our main loop:

```
for (int i = 0; i < N-1; i++) {
        for (int j = 1; j < N-1; j++) {
                Ex[i][j] = Ex[i][j] + (deltaT / permitivity / deltaZ) *
                ↪  (Hz[i][j] - Hz[i][j-1]);
        }
}

for (int i = 1; i < N-1; i++) {
        for (int j = 0; j < N-1; j++) {
                Ey[i][j] = Ey[i][j] - ((deltaT / permitivity / deltaZ) *
                ↪  (Hz[i][j] - Hz[i-1][j]));
        }
```

```
}
writeEDataToCsvFile((filePath + "E/E.csv." + to_string(i)), Ex, Ey);


// loop for values
for (int i = 0; i < N-2; i++) {
        for (int j = 0; j < N-2; j++) {
                Hz[i][j] = Hz[i][j] - ((deltaT / permeability / deltaZ) *
                ↪  (Ex[i][j] - Ex[i][j+1] + Ey[i+1][j] - Ey[i][j]));
        }
}
writeHDataToCsvFile((filePath + "H/H.csv." + to_string(i)), Hz);
```

After we are done with each vector field, we can call the respective method to write our data into csv files. For organizational purposes, we will dump the data into separate folders. After waiting for code execution to finish, we will now have two folders with data that is ready to be used for visualization.

3.3 Data Visualization

In order to visualize two-dimensional data in a meaningful manner, we are going to use a helpful open source program called Paraview[15]. It allows us to build real time simulations with the data we generated. It is also the reason we chose to name our files the way we did. By using the format *X.csv.n* we can import all of our data as one object. We can do so easily by simply opening Paraview, selecting all of our data, and just dragging and dropping it into the "Pipeline Browser" panel, pictured below (Figure 3.4).

Having just the data by itself will not do much however. That is why we need to apply filters to the data. They are:

- **Table to Points** - Simply displays the CSV data we got as points.

FIGURE 3.4: The curl around the electric field vector E_x.

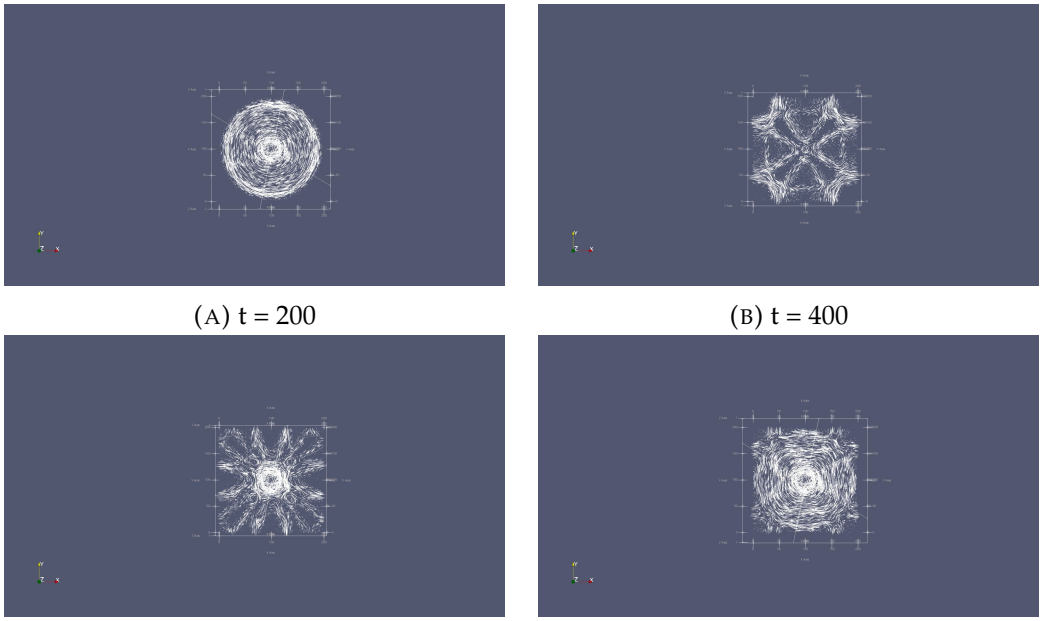
- **Calculator** - Allows us to manipulate the data and dictate which value goes where.
- **Glyph** - Transforms the data into lines which are better in visualizing vector fields

The order is important, as is the fact that each of these filters must be configured properly. For **Table to Points**, we must define the axes of our data. In our files, we have specified x and y, but also z, which contains only zeros. We needed to do that since Paraview will not accept an empty value here. It is also helpful to tick **Axes Grid** under the **Annotations** category. In the **Calculator**, we need to add the following formula

```
iHat*Ex+jHat*Ey
```

which will unify our point data. Lastly, for the **Glyph**, we need to use the **Result** array we got from our **Calculator** as both **Orientation** and **Scaling**. Depending on whether the data is visible or not, we can change the scaling slider until we get what we need. If done correctly, we will get a result similar to Figure 3.5.

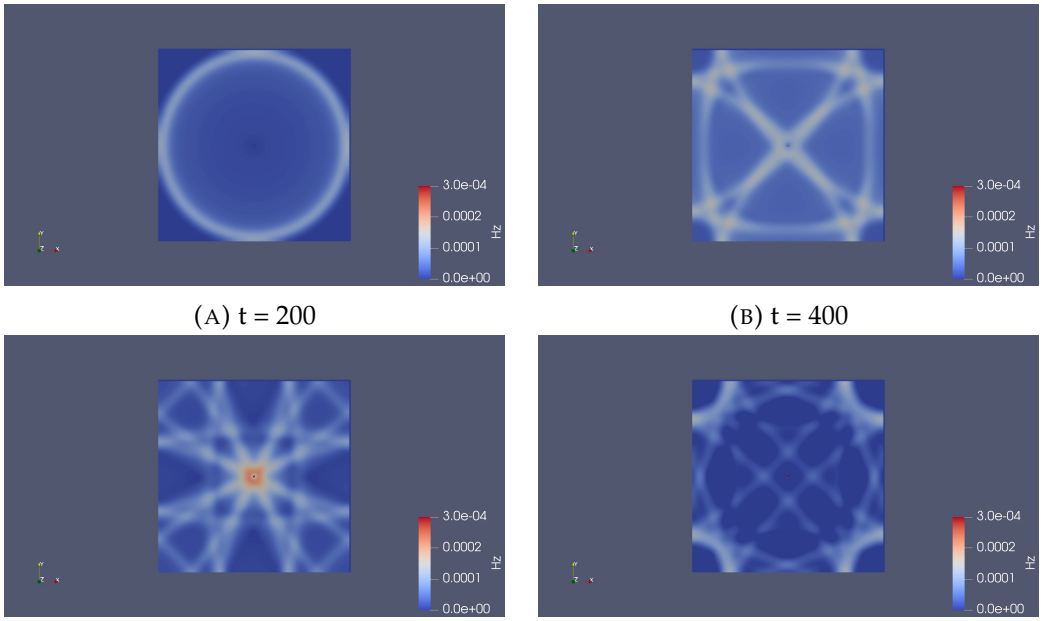
The above is for the electric data. For the magnetic data we can use a simple **Table to Structured Grid**, and it will be good just like that. With a few adjustments to colors and scaling, we can achieve the following result (Figure 3.6):



(A) t = 200 (B) t = 400

(C) t = 600 (D) t = 800

FIGURE 3.5: A simulation of the 2D electric field.



(A) t = 200 (B) t = 400

(C) t = 600 (D) t = 800

FIGURE 3.6: A simulation of the 2D magnetic field.

Moving forward, we are going to use only the method we used for visualizing electric field data, as the **Table to Structured Grid** method is not usable when we use more than one vector. With that done, we can finally move on to the three-dimensional scenario, which is the most useful one for practical real life applications.

4 FDTD - Three-Dimensional Scenario

For the three-dimensional scenario, we will see a lot of similarities with the previous versions. The methodology is again pretty much the same. It might seem a bit intimidating at first, when considering that the number of vector components we need to deal with has doubled. This scenario is also where optimization really matters, as we are dealing with N^3 amounts of data. Nonetheless, if we focus on just the most basic implementation, taking this step by step, and splitting the effort into small parts, we can easily move forward.

4.1 3D Discretization

The easiest way to explain the 3D discretization is with Figure 4.1.

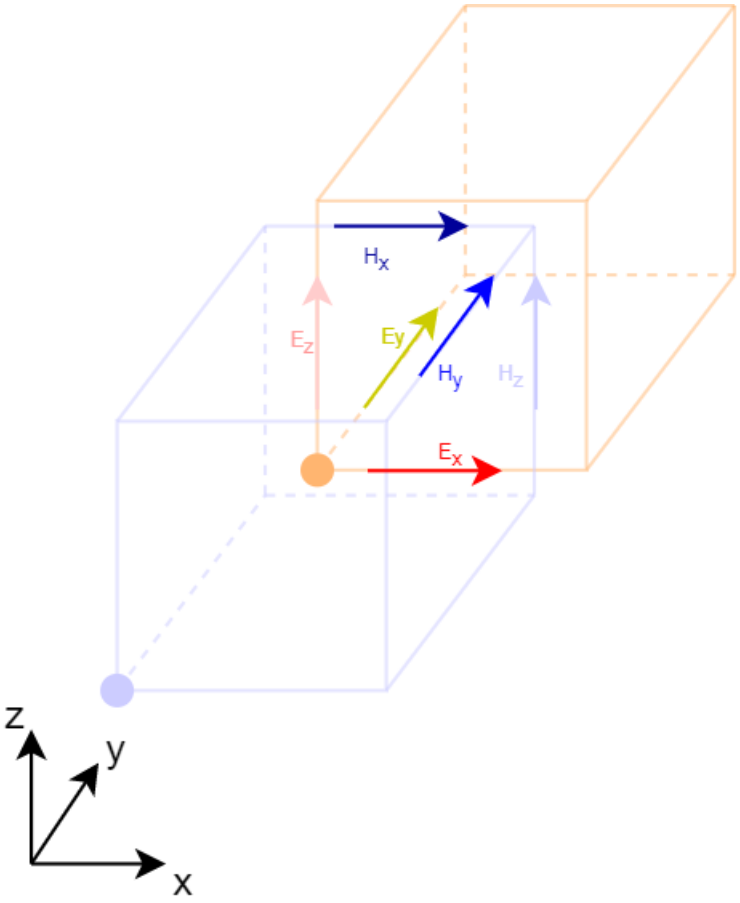


FIGURE 4.1: The figure shows the vectors for the electric field, with the respective magnetic vectors.

As seen previously, the magnetic field is staggered in every axis by half the step size. The figure also gives a hint as to how the vector components are laid out in our domain. If we were to take only the electric field into the consideration, we can show where each electric vector belongs (Figure 4.2).

Despite the fact that the image above is fairly cluttered with information, we only need to use bits of it at a time. We will notice in the next section that the electromagnetic curls for the three-dimensional scenario can be derived in the same way that we used for the two-dimensional scenario.

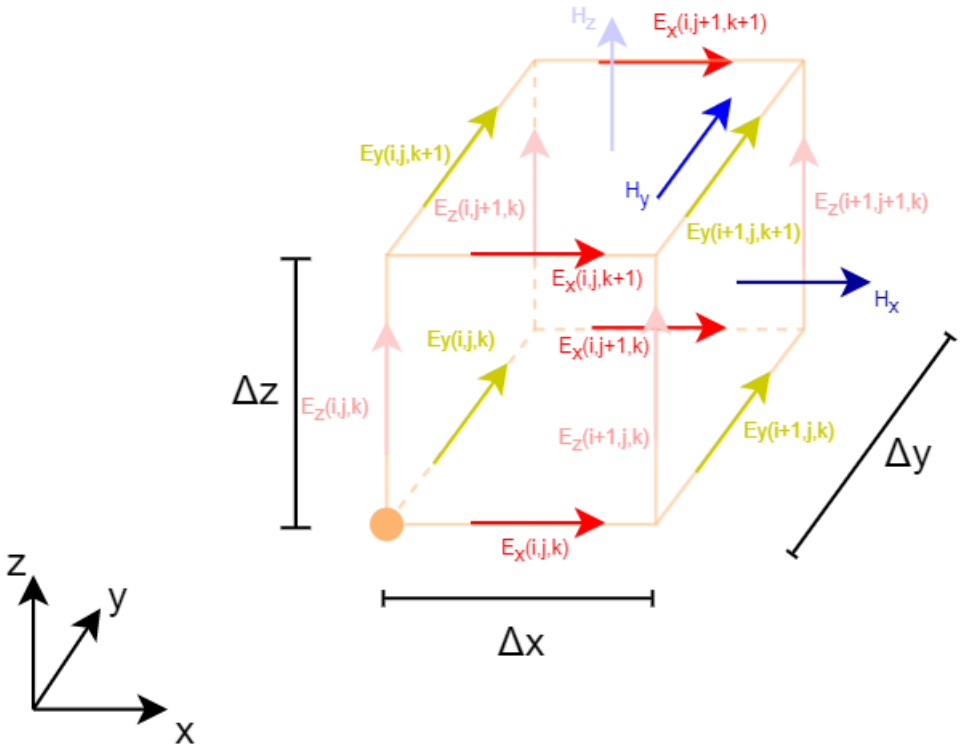


FIGURE 4.2: The figure shows the vectors for the electric field, with the respective magnetic vectors.

4.1.1 3D Electromagnetic Curls

In the previous chapter, we derived the vector curl for H_z and from there arrived at the update equation. Luckily the curl is exactly the same in the three-dimensional scenario, however we must revise our formula to account for the added dimension, starting by changing the indexing scheme to the following:

$$E_{i,j,k} = E(i \cdot \Delta x, j \cdot \Delta y, k \cdot \Delta z) \tag{4.1}$$

From there, we repeat the same process as before.

$$\oint \vec{E} \cdot d\vec{s} = -\frac{d}{dt} \iint \mu \cdot \vec{H} \cdot d\vec{A}$$

$$\Rightarrow E_x(i,j,k) \cdot \Delta x - E_x(i,j+1,k) \cdot \Delta x + E_y(i+1,j,k) \cdot \Delta y - E_y(i,j,k) \cdot \Delta y = -\frac{d}{dt}(\mu \cdot H_z(i,j,k) \cdot \Delta x \cdot \Delta y) \quad (4.2)$$

$$\frac{d}{dt} H_z(i,j,k) = -\frac{1}{\mu} \cdot \left(\frac{E_x(i,j,k) - E_x(i,j+1,k)}{\Delta y} + \frac{E_y(i+1,j,k) - E_y(i,j,k)}{\Delta x}\right) \quad (4.3)$$

Using a uniform mesh size again, $\Delta x = \Delta y = \Delta z = \Delta s$, we can simplify equation 4.3 to:

$$\frac{d}{dt} H_z(i,j,k) = -\frac{1}{\mu \cdot \Delta s} \cdot ((E_x(i,j,k) - E_x(i,j+1,k) + E_y(i+1,j,k) - E_y(i,j,k)) \quad (4.4)$$

For the left hand side:

$$\frac{d}{dt} H_z(i,j,k) = \frac{H_z^{new}(i,j,k) - H_z^{prev}(i,j,k)}{\Delta t} \quad (4.5)$$

And finally:

$$H_z^{new}(i,j,k) = H_z^{prev}(i,j,k) + \frac{\Delta t}{\mu \cdot \Delta s} \cdot (E_x(i,j+1,k) - E_x(i,j,k) - E_y(i+1,j,k) + E_y(i,j,k)) \quad (4.6)$$

Using the same method, we can get the update equation for the H_y curl in Figure 4.3

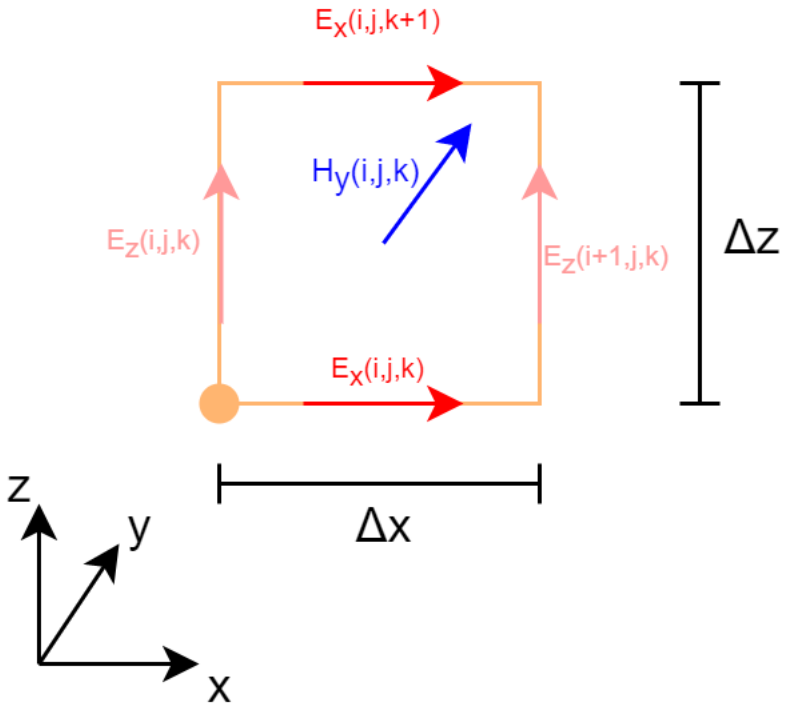


FIGURE 4.3: The curl around the magnetic field vector H_y.

We see that the magnetic vector $H_y(i,j,k)$ is affected by the electric vectors $E_x(i,j,k)$, $E_x(i,j,k+1)$, $E_z(i,j,k)$, and $E_z(i+1,j,k)$. The resulting update equation will then be:

$$H_y^{new}(i,j,k) = H_y^{prev}(i,j,k) + \frac{\Delta t}{\mu \cdot \Delta s} \cdot \\ (E_z(i+1,j,k) - E_z(i,j,k) - E_x(i,j,k+1) + E_x(i,j,k)) \quad (4.7)$$

Before continuing with the magnetic field, let's quickly take a look at the electric vector E_z curl (Figure 4.4). The reasons for this will become apparent shortly.

Again, as we did in the previous chapter the methodology is the same. We must however now account for not only the added dimension, but also the fact that previously one of our magnetic vectors was zero. Starting off from the initial Maxwell equation:

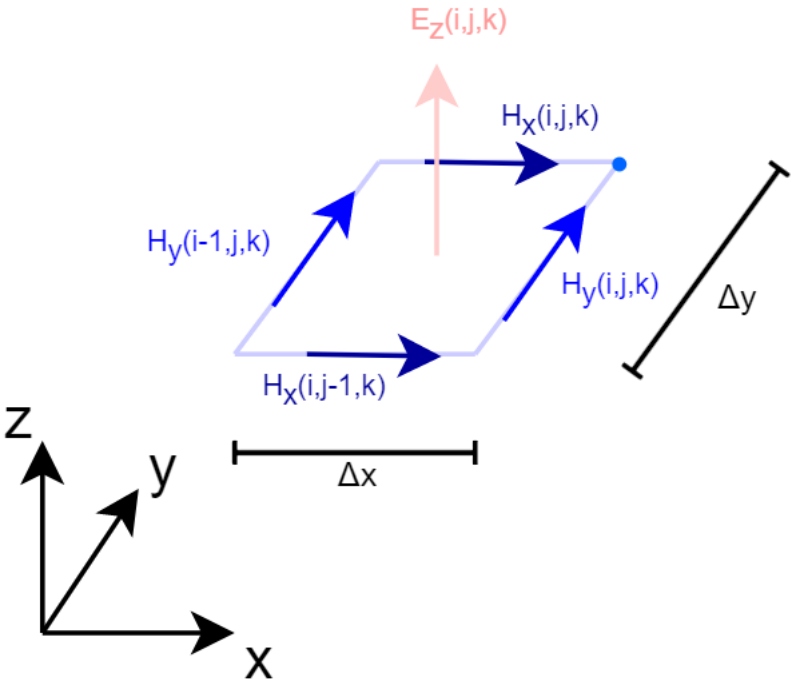


FIGURE 4.4: The curl around the electric field vector E_z.

$$\oint H \cdot ds = \frac{d}{dt} \iint \epsilon \cdot E \cdot dA \tag{4.8}$$

$$\Rightarrow H_x(i,j-1,k) \cdot \Delta x + H_y(i,j-1,k) \cdot \Delta y - H_x(i,j-1,k) \cdot \Delta x - H_y(i,j-1,k) \cdot \Delta y = \epsilon \cdot \frac{d}{dt} E_z(i,j,k) \cdot \Delta x \cdot \Delta z \tag{4.9}$$

Again taking into account the uniform mesh:

$$\frac{dE_z(i,j,k)}{dt} = \frac{1}{\epsilon \Delta s}(H_x(i,j-1,k) + H_y(i,j-1,k) - H_x(i,j-1,k) - H_y(i,j-1,k)) \tag{4.10}$$

On the left hand side:

$$\frac{dE_z(i,j,k)}{dt} = \frac{E_z^{new}(i,j,k) - E_z^{prev}(i,j,k)}{\Delta t} \tag{4.11}$$

Finally, by combining equation 4.10 and 4.11:

$$E_z^{new}(i,j,k) = E_z^{prev}(i,j,k) + \frac{\Delta t}{\epsilon \cdot \Delta s} \cdot (H_x(i-1,j,k) - H_x(i,j,k) - H_y(i,j,k-1) + H_y(i,j,k)) \quad (4.12)$$

We could proceed to manually do the curls for the remaining field vectors, however with a keen eye we can spot a pattern in the equations above, which is shown in formula 4.13.

$$V_{a_1}^{new}(i,j,k) = V_{a_1}^{prev}(i,j,k) + \frac{\Delta t}{\alpha \cdot \Delta s} \cdot (T_{a_2}(C_1) - T_{a_2}(i,j,k) - T_{a_3}(C_2) + T_{a_3}(i,j,k)) \quad (4.13)$$

where:

- **V** is the vector field type, which can be either **E** or **H**
- **T** is the opposite of **V**, meaning if $V = E$, then $T = H$
- $\boldsymbol{a_1, a_2, a_3}$ symbolize the axes, which need to follow a specific order: $x, y, z, x, y, z, x...$. As an example, if we have $a_1 = y$, then $a_2 = z$ and $a_3 = x$
- $\boldsymbol{\alpha}$ is either ϵ when calculating for an electric field vector, or μ for a magnetic one.
- $\boldsymbol{C_1, C_2}$ are the indexes of the vectors that are shifted by one in the direction of an axis, which depend on two things: which axis the vector is parallel to, and whether we are calculating for an electric field or a magnetic one.

The last point deserves a deeper explanation in order to be elaborated properly. C_1 and C_2 are indexes of type (i,j,k), where one of the indexes is ± 1. If

we are calculating the equation for the electric curl, the sign will be $+$. Otherwise it will be $-$. The updated index, meaning the one that will have the ± 1, will be the index that does not belong to either the vector we are trying to figure out the equation for, or the vector to which the curl belongs to.

Let us use the H_x vector curl, shown in Figure 4.5, to test our pattern.

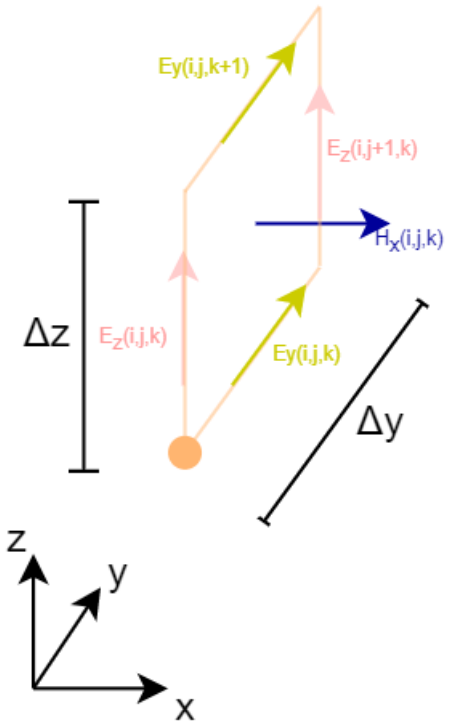


FIGURE 4.5: The curl around the magnetic field vector H_x.

We have the following:

1. $V = H$
2. $T = E$
3. $a_1 = x$, meaning $a_2 = y$ and $a_3 = z$
4. $\alpha = \mu$
5. C_1 in this case are the indexes for E_y. Since we are doing the curl for H_x, that means we are adding one to the z index, meaning that we need to replace C_1 with $i, j, k + 1$

6. Similarly, C_2 would be replaced by $i, j+1, k$

The resulting equation would then be:

$$H_x^{new}(i,j,k) = H_x^{prev}(i,j,k) + \frac{\Delta t}{\mu \cdot \Delta s} \cdot (E_y(i,j,k+1) - E_y(i,j,k) - E_z(i,j+1,k) + E_z(i,j,k)) \quad (4.14)$$

To confirm that the equation is the same, after doing the curls we would get:

$$H_x^{new}(i,j,k) = H_x^{prev}(i,j,k) - \frac{\Delta t}{\mu \cdot \Delta s} \cdot (E_y(i,j,k) + E_z(i,j,k) - E_y(i,j,k+1) - E_z(i,j,k)) \quad (4.15)$$

On the right hand side, we can flip the $-$ sign on the second half by multiplying the curl in the brackets by -1:

$$H_x^{new}(i,j,k) = H_x^{prev}(i,j,k) + \frac{\Delta t}{\mu \cdot \Delta s} \cdot -1 \cdot (E_y(i,j,k) + E_z(i,j,k) - E_y(i,j,k+1) - E_z(i,j,k)) \quad (4.16)$$

Now, by working only with the curl part, we have:

$$\begin{aligned} &-1 \cdot (E_y(i,j,k) + E_z(i,j,k) - E_y(i,j,k+1) - E_z(i,j,k)) \\ &\Leftrightarrow (-E_y(i,j,k) - E_z(i,j,k) + E_y(i,j,k+1) + E_z(i,j,k)) \\ &\Leftrightarrow (E_y(i,j,k+1) - E_y(i,j,k) - E_z(i,j+1,k) + E_z(i,j,k)) \end{aligned} \quad (4.17)$$

Therefore, our final equation is:

$$H_x^{new}(i,j,k) = H_x^{prev}(i,j,k) + \frac{\Delta t}{\mu \cdot \Delta s} \cdot (E_y(i,j,k+1) - E_y(i,j,k) - E_z(i,j+1,k) + E_z(i,j,k)) \quad (4.18)$$

which is equal to equation 4.14.

By using either method, we can derive the update equations for the vectors we have not done yet: E_x (4.19) and E_y (4.20).

$$E_x^{new}(i,j,k) = E_x^{prev}(i,j,k) + \frac{\Delta t}{\mu \cdot \Delta s} \cdot (H_y(i,j,k-1) - H_y(i,j,k) - H_z(i,j-1,k) + H_z(i,j,k)) \quad (4.19)$$

$$E_y^{new}(i,j,k) = E_y^{prev}(i,j,k) + \frac{\Delta t}{\mu \cdot \Delta s} \cdot (H_z(i-1,j,k) - H_z(i,j,k) - H_x(i,j,k-1) + H_x(i,j,k)) \quad (4.20)$$

We can now move on to the implementation.

4.2 C++ Implementation

Compared to the two-dimensional implementation, we do not have to make too many changes in order to adapt our program for three dimensions. If we use the previous code as a basis, then we already have the packages we need as well as the basic code structure. We can immediately take a look at which environment variables will need changing.

```
int N = 50;
int iterNum = 200;
```

It is highly recommended to drastically reduce the size of the domain as well as the number of iterations. The reason is that not only have we increased the amount of update loops from N^2 to N^3, but we also have six three-dimensional vectors instead of three two-dimensional ones. It is recommended to start from a small number and go up for there. The hardware that we have available can handle around this much, so we will keep these values for now. It is a good thing to note that the resulting files can be too big for Paraview to handle, so that is another limitation there.

We also need to update our *deltaT* equation now that we have three dimensions:

```
double deltaT = (deltaZ * sqrt(permitivity*permeability)  * (1/sqrt(3)));
```

Also, we need six three-dimensional vectors for our electric and magnetic fields, all initialized with N zeros in each entry:

```
vector<vector<vector<double>>> Ex(N, vector<vector<double>>(N,
↪  vector<double>(N, 0)));
vector<vector<vector<double>>> Ey(N, vector<vector<double>>(N,
↪  vector<double>(N, 0)));
vector<vector<vector<double>>> Ez(N, vector<vector<double>>(N,
↪  vector<double>(N, 0)));
vector<vector<vector<double>>> Hx(N, vector<vector<double>>(N,
↪  vector<double>(N, 0)));
vector<vector<vector<double>>> Hy(N, vector<vector<double>>(N,
↪  vector<double>(N, 0)));
vector<vector<vector<double>>> Hz(N, vector<vector<double>>(N,
↪  vector<double>(N, 0)));
```

Next we need to modify the functions that create CSV files or our data. Now that we have an equal number of vectors for both the electric and the magnetic wave, we only need one function that handles both vectors:

```
void writeDataToCsvFile(string filename, vector<vector<vector<double>>> Vx,
↪  vector<vector<vector<double>>> Vy, vector<vector<vector<double>>> Vz){


        //          x,y,z,Vx,Vy,Vz
        //          0,0,Vx[x,y,z],Vy[x,y,z],Vz[x,y,z]


        ofstream csvFile(filename);
        csvFile << "x,y,z,Vx,Vy,Vz\n";


        for (unsigned  x = 0; x < Vx[0][0].size(); x++) {
                for (unsigned  y = 0; y < Vy[x][0].size(); y++) {
                        for (unsigned  z = 0; z < Vz[x][y].size(); z++) {
                                csvFile << x << "," << y << "," << z << ","
                                ↪  << Vx[x][y][z] << "," << Vy[x][y][z] <<
                                ↪  "," << Vz[x][y][z] << "\n";
                        }
                }
        }


        csvFile.close();
}
```

For our main loop, we can apply the Gaussian Pulse excitation to either an electric field vector or the magnetic one. Both work, but we need to keep in mind that when using the magnetic field for the excitation, we need to lower the magnitude if using vacuum as a medium. Depending on the material used in the domain, this may be unnecessary. Again, we apply this excitation in the middle of our vector so that we have a nice symmetrical wave propagation.

```
Ex[24][24][24] = exp(-(beta * pow((t - gamma), 2)));
```

We only need one vector, and this will also affect the order of our update

loops. If using an electric vector for the excitation, the loop order is as follows:

```
// loop for values
for (int i = 0; i < N-1; i++) {
        for (int j = 0; j < N-2; j++) {
                for (int k = 0; k < N-2; k++) {
                        Hx[i][j][k] = Hx[i][j][k] + (deltaT / permeability
                        ↪ / deltaZ) * (Ey[i][j][k+1] - Ey[i][j][k] -
                        ↪ Ez[i][j+1][k] + Ez[i][j][k]);
                }
        }
}

for (int i = 0; i < N-2; i++) {
        for (int j = 0; j < N-1; j++) {
                for (int k = 0; k < N-2; k++) {
                        Hy[i][j][k] = Hy[i][j][k] + (deltaT / permeability
                        ↪ / deltaZ) * (Ez[i+1][j][k] - Ez[i][j][k] -
                        ↪ Ex[i][j][k+1] + Ex[i][j][k]);
                }
        }
}

for (int i = 0; i < N-2; i++) {
        for (int j = 0; j < N-2; j++) {
                for (int k = 0; k < N-1; k++) {
                        Hz[i][j][k] = Hz[i][j][k] + (deltaT / permeability
                        ↪ / deltaZ) * (Ex[i][j+1][k] - Ex[i][j][k] -
                        ↪ Ey[i+1][j][k] + Ey[i][j][k]);
                }
        }
}

writeDataToCsvFile((filePath + "H/H.csv." + to_string(i)), Hx, Hy, Hz);
```

```
for (int i = 0; i < N-1; i++) {
        for (int j = 1; j < N-1; j++) {
                for (int k = 1; k < N-1; k++) {
                        Ex[i][j][k] = Ex[i][j][k] + (deltaT / permitivity /
                        ↪ deltaZ) * (Hy[i][j][k-1] - Hy[i][j][k] -
                        ↪ Hz[i][j-1][k] + Hz[i][j][k] );
                }
        }
}

for (int i = 1; i < N-1; i++) {
        for (int j = 0; j < N-1; j++) {
                for (int k = 1; k < N-1; k++) {
                        Ey[i][j][k] = Ey[i][j][k] + (deltaT / permitivity /
                        ↪ deltaZ) * (Hz[i-1][j][k] - Hz[i][j][k] -
                        ↪ Hx[i][j][k-1] + Hx[i][j][k]);
                }
        }
}

for (int i = 1; i < N-1; i++) {
        for (int j = 1; j < N-1; j++) {
                for (int k = 0; k < N-1; k++) {
                        Ez[i][j][k] = Ez[i][j][k] + (deltaT / permitivity /
                        ↪ deltaZ) * (Hx[i][j-1][k] - Hx[i][j][k] -
                        ↪ Hy[i-1][j][k] + Hy[i][j][k]);
                }
        }
}

writeDataToCsvFile((filePath + "E/E.csv." + to_string(i)), Ex, Ey, Ez);

}
```

When using a magnetic vector for the excitation, we would need to run the electric loops first. And again, we need to keep in mind that our loop indexes do not go out of bounds for the magnetic field equations.

With that said, after running the program we should have 200 files (indexed from 0 to 199) that we can grab and place in Paraview as we did before.

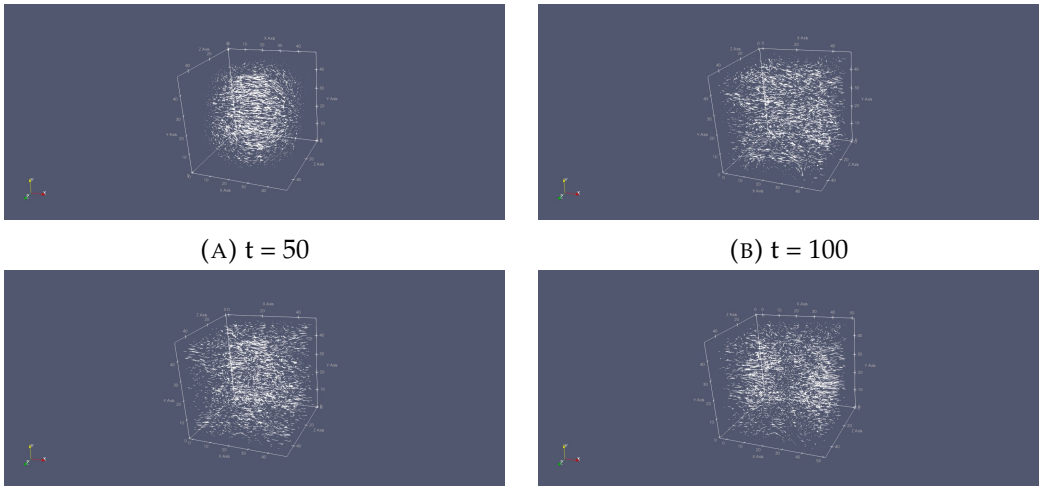
4.3 Data Visualization

The process for three-dimensional data is going to be almost exactly the same as the two-dimensional scenario in the previous chapter. The main caveat here is to make sure to update the equation for the **Calculator** filter by clicking it in the **Pipeline Browser** and on the properties page type

```
iHat*Vx+jHat*Vy+kHat*Vz
```

. Please note that Vx,Vy, and Vz, are taken directly from the header of our CSV files. If that header is different, we need to reflect that here. In our case, we are using *"x,y,z,Vx,Vy,Vz"*.

After setting it up and playing around with the configurations a bit, we can get the following simulation for our electric data (Figure 4.6).

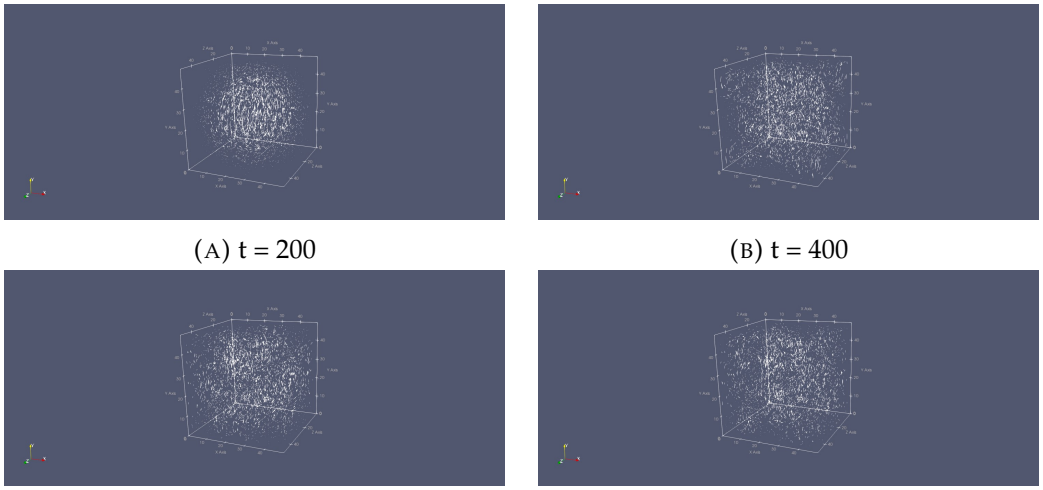


(A) t = 50 (B) t = 100

(C) t = 150 (D) t = 200

FIGURE 4.6: A simulation of the 3D electric field.

Following the same steps, we can do the same for the magnetic data as well. Please note that we scale this data up a bit, as it would be quite hard to see otherwise due to using vacuum as a medium.



(A) t = 200 (B) t = 400

(C) t = 600 (D) t = 800

FIGURE 4.7: A simulation of the 3D magnetic field.

And with that, we have completed our simulation for a three-dimensional scenario.

5 Conclusion

Through the help of many scientists and researchers, we were able to simulate electromagnetic wave data for three scenarios using a simple implementation that is easy to adapt and expand upon. At first, we explained the value of simulations in research, but also in various industries. We showed examples of capturing the movement of a light particle[20], or being able to capture an image of a black hole[12], something that NASA achieved quite recently.

In both examples above, these achievements would not be possible without having a basic idea of what to look for. It is thanks to Einstein and his theory of relativity that we were able to know the existence of black holes to begin with[10]. And it was computers that compiled multiple small images into one. Indeed the development of technology has allowed us to not only achieve such feats, but also create simulations of various phenomena from scratch, which imitate nature closely, but not quite perfectly.

Unless we are able to create computers that can achieve infinite precision, something that right now is not possible, we will always have a degree of error introduced into simulations that rely on infinite values. Even finite values will have to be truncated, provided they are large enough. Despite that, by using algorithms such as FDTD, we can minimize this error to a point where it is negligible.

For electromagnetic waves specifically, simulating their behavior gives us far more data than observation can and far easier. This holds especially true

when considering theoretical scenarios, such as comparing the electromagnetic properties of two different kinds of metals, or how these waves would behave in a vacuum. Such a simulation is only possible through knowledge of the basics, which in the case of electromagnetic phenomena would be Maxwell's Equations.

These equations were derived by James Maxwell from the work of previous fellow physicists: Gauss and his laws for electricity and magnetism, Faraday and his law of induction, and Ampère with his circuital law. Maxwell took these laws and created equations that are believed to govern all electromagnetic phenomena. We adapted these equations in our FDTD implementation to create three C++ programs, capable of simulating electromagnetic phenomena in one-dimensional, two-dimensional, and three-dimensional cases.

We began by first deriving the numerical solution to the Wave Equation. With that, we could choose from many algorithms as to how to proceed. We chose FDTD due to how simple it is and how it can be implemented in a realistic amount of time by only one person[8].

FDTD, initially called the Yee algorithm because it was proposed by Kane Yee, was modified by fellow researchers to become a staple of simulations for the wave equation. It can be implemented by following a number of steps, the first one being the transformation of Maxwell's Equations into finite differences. After that, we discretize the domain and formulate what are called update equations: formulas that allow us to derive the next step of the respective field.

We followed this process for all three scenarios, making changes where they are needed to account for dimensions. We used electromagnetic curls to derive the update equations, which we could then use in our implementations. After the programs generated the necessary data, we used it to visualize our simulations in real time, by using a program called Paraview[15].

With that, this project is officially over. However, the implementation is fairly simplistic. It was left this way on purpose so that it can be modified easily. In the Appendix (A) we will go over some possible improvements. There we can also find the tools used during this project, how to adapt this implementation to fit into another application, and how to change the program so that it can support domains that are formed of more than one material. Included is also a short guide to troubleshooting possible issues with the implementation, as well as the full code for each scenario.

Hopefully, this will prove helpful to anyone that will want to work further on this project, or to integrate it into their own.

A Appendix

A.1 Tools Used

A.1.1 Hardware

A.1.2 Software

A.2 Troubleshooting the implementation

A.3 Integration into a bigger program

A.4 Using a domain with different environments

A.5 General Improvements

A.5.1 Performance Improvements

A.5.2 Improvements for Symmetric Cases

A.5.3 Using CUDA

A.6 Full Code Files

A.6.1 FDTD 1D

```
#define _USE_MATH_DEFINES
```

```
#include <iostream>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <cmath>
#include <vector>
#include <string>

using namespace std;

const double permitivity = 8.854e-12;
const double permeability = 1.256e-6;

double L = 5;
int N = 200;
int iterNum = 800;
//double deltaX = L / N;
//double deltaY = L / N;
double deltaZ = L / N;
double deltaT = (deltaZ * sqrt(permitivity*permeability));

// variables needed for Gaussian Pulse excitation
double eps = 1e-3;
double Teps = 50 * deltaT;
double beta = -(pow((2/Teps), 2) * log(eps));

vector<double> E;
vector<double> H;
vector<double> tE;
vector<double> tH;

int main()
{
        E.assign(N, 0);
        H.assign(N, 0);
```

```
for(int i = 0; i < iterNum; i++) {

        double t = i * deltaT;
        double gamma = Teps / 2;

        E[0] = exp(-(beta * pow((t - gamma), 2)));

        // loop for values
        for (int z = 0; z < N-1; z++) {
                H[z] = H[z] - (deltaT / permeability /
                 ↪  deltaZ) * (E[z] - E[z+1]);
        }

        for (int z = 1; z < N-1; z++) {
                E[z] = E[z] + (deltaT / permitivity /
                 ↪  deltaZ) * (H[z] - H[z-1]);
        }

        // time graph
        tE.push_back(E[100]);
        tH.push_back(H[100]);
}
cout << "\n\ntE\n";

// print E values
for (int n = 0; n < iterNum; n++) {
        cout << to_string(tE[n]) + ",";
}

cout << "\n\ntH\n";

// print H values
for (int n = 0; n < iterNum; n++) {
```

```
                cout << to_string(tH[n]) + ",";
            }
        }
```

A.6.2 FDTD 2D

```
#define _USE_MATH_DEFINES

#include <iostream>
#include <stdio.h>
#include <io.h>
#include <math.h>
#include <stdlib.h>
#include <cmath>
#include <vector>
#include <string>
#include <fstream>
#include <cstdarg>

using namespace std;

const double permitivity =
↪  8.854e-12;
↪  vacuum permitivity
const double permeability = 1.256e-6;
↪
↪  // vacuum permeability

double L = 5;
int N = 200;
int iterNum = 800;
double deltaX = L / N;
double deltaY = L / N;
double deltaZ = L / N;
```

```
double deltaT = (deltaZ * sqrt(permitivity*permeability)  *
↪  (1/sqrt(2))); // 1/C * 1/sqrt2 * deltaZ

// variables needed for Gaussian Pulse excitation
double eps = 1e-3;
double Teps = 50 * deltaT;
double beta = -(pow((2/Teps), 2) * log(eps));



vector<vector<double>> Ex(N, vector<double> (N, 0));
vector<vector<double>> Ey(N, vector<double> (N, 0));
vector<vector<double>> Hz(N, vector<double> (N, 0));

const string filePath = "./Out/";

void writeEDataToCsvFile(string filename, vector<vector<double>>
↪  Ex, vector<vector<double>> Ey){

        //        "x","y",Ex,Ey
        //        0,0,Ex[x,y],Ey[x,y]

        ofstream csvFile(filename);
        csvFile << "x,y,z,Ex,Ey\n";

        for (int x = 0; x < Ex[0].size(); x++) {
                for (int y = 0; y < Ex[x].size(); y++) {
                        csvFile << to_string(x) + "," +
                        ↪  to_string(y) + ",0," +
                        ↪  to_string(Ex[x][y]) + "," +
                        ↪  to_string(Ey[x][y]) + "\n";
                }
        }

        csvFile.close();
}
```

```
void writeHDataToCsvFile(string filename, vector<vector<double>>
↪  Hz){

        //          "x","y",Hz
        //          0,0,Hz[x,y]

        ofstream csvFile(filename);
        csvFile << "x,y,z,Hz\n";

        for (int x = 0; x < Hz[0].size(); x++) {
                for (int y = 0; y < Ex[x].size(); y++) {
                        csvFile << to_string(x) + "," +
                        ↪  to_string(y) + ",0," +
                        ↪  to_string(Hz[x][y]) + "\n";
                }
        }

        csvFile.close();
}

int main()
{
        for(int i = 0; i < iterNum; i++) {

                double t = i * deltaT;
                double gamma = Teps / 2;

                // reducing the magnitude since in free space
                Hz[99][99] = exp(-(beta * pow((t - gamma), 2))) *
                ↪  10e-4;  //TO-DO: Gaussian excitation, alpha =
                ↪  1, Teps = 50*deltaT, eps = 1e-3, t = i * deltaT

                for (int i = 0; i < N-1; i++) {
                        for (int j = 1; j < N-2; j++) {
```

```
                                Ex[i][j] = Ex[i][j] + (deltaT /
                                ↪  permitivity / deltaZ) *
                                ↪  (Hz[i][j] - Hz[i][j-1]);
                        }
                }

                for (int i = 1; i < N-2; i++) {
                        for (int j = 0; j < N-1; j++) {
                                Ey[i][j] = Ey[i][j] - ((deltaT /
                                ↪  permitivity / deltaZ) *
                                ↪  (Hz[i][j] - Hz[i-1][j]));
                        }
                }

                writeEDataToCsvFile((filePath + "E/E.csv." +
                ↪  to_string(i)), Ex, Ey);

                // loop for values
                for (int i = 0; i < N-1; i++) {
                        for (int j = 0; j < N-1; j++) {
                                Hz[i][j] = Hz[i][j] - ((deltaT /
                                ↪  permeability / deltaZ) *
                                ↪  (Ex[i][j] - Ex[i][j+1] +
                                ↪  Ey[i+1][j] - Ey[i][j]));
                        }
                }


                writeHDataToCsvFile((filePath + "H/H.csv." +
                ↪  to_string(i)), Hz);

        }
}
```

A.6.3 FDTD 3D

```
#define _USE_MATH_DEFINES

#include <iostream>
#include <stdio.h>
#include <io.h>
#include <math.h>
#include <stdlib.h>
#include <cmath>
#include <vector>
#include <string>
#include <fstream>
#include <cstdarg>

using namespace std;

const double permitivity = 8.854e-12;
const double permeability = 1.256e-6;

double L = 5;
int N = 50;
int iterNum = 200;
double deltaX = L / N;
double deltaY = L / N;
double deltaZ = L / N;
double deltaT = (deltaZ * sqrt(permitivity*permeability)  *
↪  (1/sqrt(3))); // 1/C * 1/sqrt2 * deltaZ

// variables needed for Gaussian Pulse excitation
double eps = 1e-3;
double Teps = 50 * deltaT;
double beta = -(pow((2/Teps), 2) * log(eps));
```

```
vector<vector<vector<double>>> Ex(N, vector<vector<double>>(N,
↪  vector<double>(N, 0)));
vector<vector<vector<double>>> Ey(N, vector<vector<double>>(N,
↪  vector<double>(N, 0)));
vector<vector<vector<double>>> Ez(N, vector<vector<double>>(N,
↪  vector<double>(N, 0)));
vector<vector<vector<double>>> Hx(N, vector<vector<double>>(N,
↪  vector<double>(N, 0)));
vector<vector<vector<double>>> Hy(N, vector<vector<double>>(N,
↪  vector<double>(N, 0)));
vector<vector<vector<double>>> Hz(N, vector<vector<double>>(N,
↪  vector<double>(N, 0)));

const string filePath = "./Out/";

void writeDataToCsvFile(string filename,
↪  vector<vector<vector<double>>> Vx,
↪  vector<vector<vector<double>>> Vy,
↪  vector<vector<vector<double>>> Vz){
        ofstream csvFile(filename);
        csvFile << "x,y,z,Vx,Vy,Vz\n";

        for (unsigned  x = 0; x < Vx[0][0].size(); x++) {
                for (unsigned  y = 0; y < Vy[x][0].size(); y++) {
                        for (unsigned  z = 0; z < Vz[x][y].size();
                        ↪  z++) {
                                csvFile << x << "," << y << "," <<
                                ↪  z << "," << Vx[x][y][z] << ","
                                ↪  << Vy[x][y][z] << "," <<
                                ↪  Vz[x][y][z] << "\n";
                        }
                }
        }

        csvFile.close();
```

```
}

int main()
{
        for(int i = 0; i < iterNum; i++) {

                double t = i * deltaT;
                double gamma = Teps / 2;

                // reducing the magnitude since in free space
                Ex[24][24][24] = exp(-(beta * pow((t - gamma), 2)))
                ↪  * 10e-4;  //TO-DO: Gaussian excitation, alpha =
                ↪  1, Teps = 50*deltaT, eps = 1e-3, t = i * deltaT
                //Ey[9][9][9] = exp(-(beta * pow((t - gamma), 2)))
                ↪  * 10e-4;
                //Ez[9][9][9] = exp(-(beta * pow((t - gamma), 2)))
                ↪  * 10e-4;

                // loop for values
                for (int i = 0; i < N-1; i++) {
                        for (int j = 0; j < N-2; j++) {
                                for (int k = 0; k < N-2; k++) {
                                        Hx[i][j][k] = Hx[i][j][k] +
                                        ↪  (deltaT / permeability
                                        ↪  / deltaZ) *
                                        ↪  (Ey[i][j][k+1] -
                                        ↪  Ey[i][j][k] -
                                        ↪  Ez[i][j+1][k] +
                                        ↪  Ez[i][j][k]);
                                }
                        }
                }

                for (int i = 0; i < N-2; i++) {
                        for (int j = 0; j < N-1; j++) {
```

```
                        for (int k = 0; k < N-2; k++) {
                                Hy[i][j][k] = Hy[i][j][k] +
                                ↪  (deltaT / permeability
                                ↪  / deltaZ) *
                                ↪  (Ez[i+1][j][k] -
                                ↪  Ez[i][j][k] -
                                ↪  Ex[i][j][k+1] +
                                ↪  Ex[i][j][k]);
                        }
                }
        }

        for (int i = 0; i < N-2; i++) {
                for (int j = 0; j < N-2; j++) {
                        for (int k = 0; k < N-1; k++) {
                                Hz[i][j][k] = Hz[i][j][k] +
                                ↪  (deltaT / permeability
                                ↪  / deltaZ) *
                                ↪  (Ex[i][j+1][k] -
                                ↪  Ex[i][j][k] -
                                ↪  Ey[i+1][j][k] +
                                ↪  Ey[i][j][k]);
                        }
                }
        }

        writeDataToCsvFile((filePath + "H/H.csv." +
        ↪  to_string(i)), Hx, Hy, Hz);

        for (int i = 0; i < N-2; i++) {
                for (int j = 1; j < N-2; j++) {
                        for (int k = 1; k < N-2; k++) {
```

```
                                Ex[i][j][k] = Ex[i][j][k] +
                                ↪  (deltaT / permitivity /
                                ↪  deltaZ) * (Hz[i][j][k]
                                ↪  - Hz[i][j-1][k] -
                                ↪  Hy[i][j][k] +
                                ↪  Hy[i][j][k-1]);
                        }
                }
        }

        for (int i = 1; i < N-2; i++) {
                for (int j = 0; j < N-2; j++) {
                        for (int k = 1; k < N-2; k++) {
                                Ey[i][j][k] = Ey[i][j][k] +
                                ↪  (deltaT / permitivity /
                                ↪  deltaZ) * (Hx[i][j][k]
                                ↪  - Hx[i][j][k-1] -
                                ↪  Hz[i][j][k] +
                                ↪  Hz[i-1][j][k]);
                        }
                }
        }

        for (int i = 1; i < N-2; i++) {
                for (int j = 1; j < N-2; j++) {
                        for (int k = 0; k < N-2; k++) {
                                Ez[i][j][k] = Ez[i][j][k] +
                                ↪  (deltaT / permitivity /
                                ↪  deltaZ) * (Hy[i][j][k]
                                ↪  - Hy[i-1][j][k] -
                                ↪  Hx[i][j][k] +
                                ↪  Hx[i][j-1][k]);
                        }
                }
        }
```

```
                writeDataToCsvFile((filePath + "E/E.csv." +
                ↪  to_string(i)), Ex, Ey, Ez);
            }
        }
```

Bibliography

[1] JB Cao et al. "First results of low frequency electromagnetic wave detector of TC-2/Double Star program". In: (2005).

[2] cplusplus.com. URL: http://www.cplusplus.com/reference/cmath/.

[3] cplusplus.com. URL: http://www.cplusplus.com/reference/iostream/.

[4] cplusplus.com. URL: http://www.cplusplus.com/reference/cstdlib/.

[5] cplusplus.com. URL: http://www.cplusplus.com/reference/vector/.

[6] cplusplus.com. URL: http://www.cplusplus.com/reference/string/.

[7] cplusplus.com. URL: https://www.cplusplus.com/reference/fstream/fstream/.

[8] David B Davidson. *Computational electromagnetics for RF and microwave engineering*. Cambridge University Press, 2010.

[9] Eclipse. URL: https://www.eclipse.org/downloads/packages/release/kepler/sr2/eclipse-ide-cc-developers.

[10] Christopher Eling and Ted Jacobson. "Spherical solutions in Einstein-aether theory: static aether and stars". In: *Classical and Quantum Gravity* 27.4 (2010), p. 049802. DOI: 10.1088/0264-9381/27/4/049802. URL: https://doi.org/10.1088/0264-9381/27/4/049802.

[11] GCC. URL: https://gcc.gnu.org/onlinedocs/gcc-4.6.2/libstdc++/api/a00782_source.html.

[12] Elizabeth Landau. "Black Hole Image Makes History; NASA Telescopes Coordinated Observations". In: *Nasa.gov* (2019). Ed. by Sarah Loff. URL: https://www.nasa.gov/mission_pages/chandra/news/black-hole-image-makes-history.

[13] Daryl L Logan. "First course in finite element method, si". In: *Mason, OH: South-Western, Cengage Learning* (2011).

[14] Microsoft. URL: https://www.microsoft.com/en/microsoft-365/excel.

[15] ParaView. URL: https://www.paraview.org/.

[16] Arthur William Poyser. *Magnetism and electricity: A manual for students in advanced classes*. Longmans, Green and Company, 1918.

[17] Xhoni Robo. *XhRobo/Finite-Difference-Time-Domain-Method-Implementation-in-C-*. URL: https://github.com/XhRobo/Finite-Difference-Time-Domain-Method-Implementation-in-C-.

[18] Julius Adams Stratton. *Electromagnetic theory*. Vol. 33. John Wiley & Sons, 2007.

[19] Vel, Steve R. Gunn, and Sunil Patel. "Masters/Doctoral Thesis LaTeX Template". In: *LaTeX Templates* (). URL: https://www.latextemplates.com/template/masters-doctoral-thesis.

[20] Andreas Velten et al. "Femto-photography: capturing and visualizing the propagation of light". In: (2013).

[21] T Weiland. "A discretization method for the solution of Maxwell's equations for six-component fields.-Electronics and Communication,(AEÜ), Vol. 31". In: (1977).