FRANKFURT UNIVERSITY OF APPLIED SCIENCES

MASTER THESIS

Finite Difference Time Domain Method Implementation in C++

Author:
Xhoni ROBO

First Supervisor:
Prof. Dr. Peter THOMA

Second 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

January 3, 2021

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

In this thesis, the author discusses the implementation of a Finite Difference Time Domain Method using the C++ programming language, to generate data for simulations of electromagnetic waves. This thesis aims to use terms that are as simple as possible, so that it can be understood easily by those who, like the author, have studied primarily computer science rather than physics. Three scenarios will be discussed here: the first with a one-dimensional domain, then a two-dimensional one, and lastly the three-dimensional implementation, which would have the most real-life applications. None of this would be possible however without the work of previous researchers and scientists. In particular, this project relies heavily on Maxwell's Equations for electromagnetics, which describe the behavior of all electromagnetic waves in a static environment. After first discretizing the domain, the formulas are adapted to the specific scenario, resulting in update equations that calculate the value of the next step of the simulation. While doing so, the thesis goes over transverse modes and electromagnetic vector curls. After each scenario is discretized properly, it is possible to proceed to the code implementation, where each of the equations is adapted and added inside a loop. The result is then processed with the help of an external program to visualize the simulation in a way that is more easily understandable to humans. In the end, an appendix holds useful information that one can use should they intend to further work on this project, or implement it into their own work.

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, which convinced me that such a subject would make an interesting thesis for me.

I would also like to thank 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 self-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 for their sacrifices, but hopefully, this is, at the very least, a step in the right direction.

Sincerely,
Xhoni Robo

Contents

Declaration of Authorship i

Abstract iii

Acknowledgements iv

1 Introduction 1

1.1 Motivation . . . 2

1.2 Electromagnetic Simulations . . . 3

1.2.1 Maxwell Equations . . . 4

1.2.2 Solving the Wave Equation . . . 6

1.3 Finite Difference Time Domain Method . . . 8

1.4 FDTD Implementation . . . 9

1.4.1 Application Requirements . . . 10

1.4.2 Computational Limitations and Inaccuracies Explained . . . 13

1.4.3 Discretization Error[25] . . . 15

2 FDTD - One-Dimensional Scenario 17

2.1 1D Discretization . . . 17

2.1.1 Spatial and Temporal Shift . . . 18

2.1.2 Electromagnetic Curls . . . 18

2.1.3 Stability and Energy Conservation . . . 23

2.2 C++ Implementation . . . 24

2.3 Data Visualization . . . 30

3 FDTD - Two-Dimensional Scenario 35

3.1 2D Discretization . . . 35
3.1.1 2D Transverse Modes . . . 35
3.1.2 2D TE Electromagnetic Curls . . . 36
3.2 C++ Implementation . . . 39
3.3 Data Visualization . . . 43

4 FDTD - Three-Dimensional Scenario **47**
4.1 3D Discretization . . . 48
4.1.1 3D Electromagnetic Curls . . . 49
4.2 C++ Implementation . . . 56
4.3 Data Visualization . . . 61

5 Conclusion **63**

A Appendix **66**
A.1 Tools Used . . . 66
A.1.1 Hardware . . . 66
A.1.2 Software . . . 67
A.2 Troubleshooting the implementation . . . 68
A.3 Integration into a bigger program . . . 69
A.4 Using a domain with different environments . . . 70
A.5 Using non uniform meshes . . . 72
A.6 General Improvements . . . 76
A.6.1 Performance Improvements . . . 76
A.6.2 Improvements for Symmetric Cases . . . 77
A.6.3 Using CUDA . . . 79
A.7 Full Code Files . . . 80
A.7.1 FDTD 1D . . . 80
A.7.2 FDTD 2D . . . 82
A.7.3 FDTD 3D . . . 86

Bibliography **91**

List of Figures

1.1 Faraday's Experiment 5
1.2 Magnetic Divergence 6
1.3 Code Result 14
1.4 Discretization Error 15

2.1 1D Spatial and Temporal Shift - TEM Mode 18
2.2 1D Curl around H_y 19
2.3 1D Curl around E_x 20
2.4 Leapfrog Time Scheme 21
2.5 Instability 23
2.6 1D Console Output 30
2.7 1D Excel - Text to Columns 31
2.8 1D Excel - Text to Columns 31
2.9 1D Electromagnetic Time Graph 32
2.10 1D Magnetic Time Snippet 33

3.1 2D TE Mode - H_z vector curl 36
3.2 2D TE Mode - E_x vector curl 38
3.3 2D TE Mode - E_y vector curl 39
3.4 2D TE Mode - E_x vector curl 44
3.5 2D Electric Field Simulation 45
3.6 2D Magnetic Field Simulation 45

4.1 3D Electric Discretization 48

4.2 3D Electric Discretization 49
4.3 3D H_y vector curl 51
4.4 3D E_z vector curl 52
4.5 3D H_x vector curl 54
4.6 3D Electric Field Simulation 62
4.7 3D Magnetic Field Simulation 62

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	$Z_0 = 376.730\,313\,668(57)\,\Omega$
Speed of Light	$c = 299\,792\,458\,\mathrm{m\,s^{-1}}$

1 Introduction

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[22].

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 standard C++. 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 [12]. 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 includes the code files which can be found in the GitHub repository[24]. As the base LaTeX template of this thesis was found online[29], 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.

1.1 Motivation

As humanity strives to better understand the world and universe around it, the physical limitations become more and more apparent. While people have made considerable progress in their 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 known of so far[30], or being able to capture an image of a black hole[17], such achievements would not have been possible if 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, a starting impulse will be eeded. For this project, the Gaussian pulse equation will be adapted to add an excitation to the simulation that would otherwise be static.

At the end, one can find the conclusion where uses for this application as well as further improvements are discussed. Anyone that would like to use this project as a basis for their own work should take a look at the Appendix (A) should they have any issues.

1.2 Electromagnetic Simulations

With the fast development of technology came new opportunities for gaining a better understanding of vast natural phenomena. This thesis 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[2]. Even if one 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 one cannot proceed without first going into the equations that govern large-scale electromagnetic phenomena in static mediums[26]: Maxwell Equations.

1.2.1 Maxwell Equations

As mentioned above, Maxwell Equations are believed to describe 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 left hand side is the EMF or voltage and the right hand side is the time derivative of 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 displacement current $\frac{\partial \vec{D}}{\partial t}$. The equation explains the effects of the magnetic field and the excitation current $\vec{J}$ on the electric field.

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

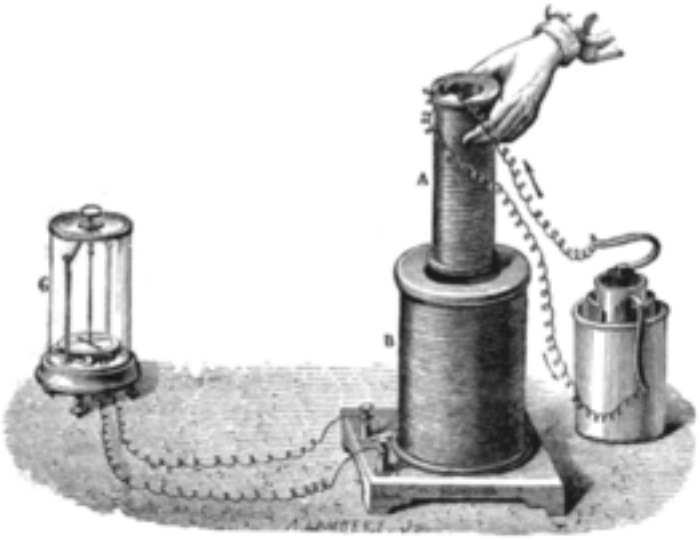


FIGURE 1.1: Faraday's Experiment,[23]
which resulted in the law that was later used by Maxwell in making his equation.

in one direction. That is why every magnet that is known so far, at least under normal conditions, of has two poles. To translate this into something more easily understandable, it basically means that, if one was to pick any subset of the area of a magnetic field, no matter what area was picked, it would have the same number of vector fields going inside and out. 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 fields. 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.

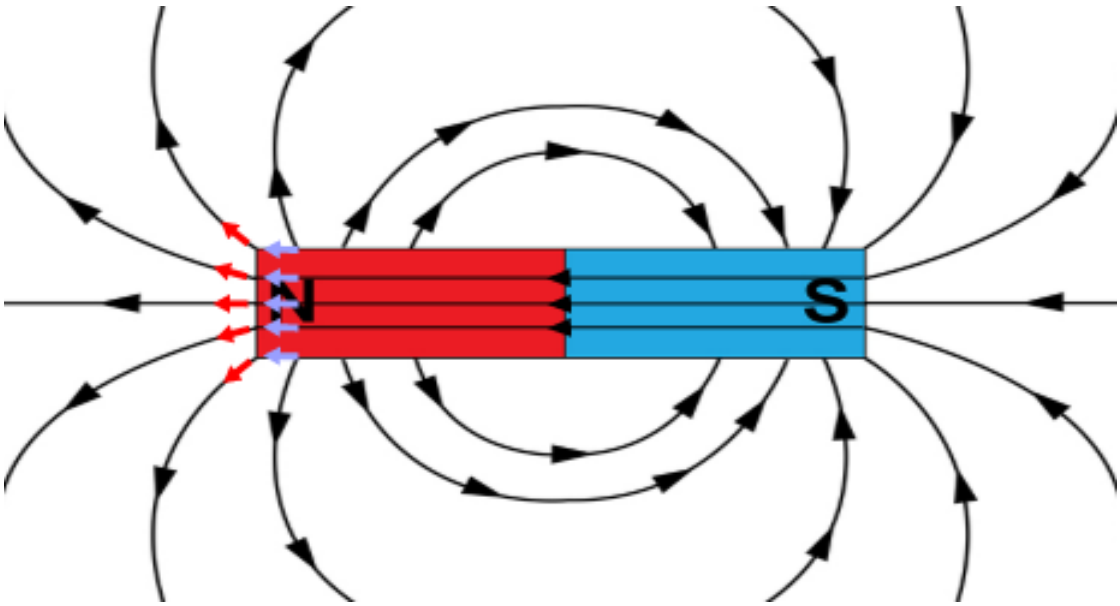


FIGURE 1.2: A crude representation of magnetic divergence through the use of a bar magnet (picture modified from [10]).

For the above equations, the following material relations 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 Maxwell equations above are shown in their derivative form, but they can also be shown as their integral form equivalent. There is no difference in implementation, regardless of which form is used.

1.2.2 Solving the Wave Equation

Using the formulas above, one can derive the electromagnetic wave equation, which is needed to proceed with the implementation further. Firstly for convenience, an assumption will be made that the environment is the vacuum of space. Secondly, it can also be assumed that there is no initial charge in this space nor any loss of energy. This means that $\epsilon(\vec{r}) = \epsilon_0$ and $\mu(\vec{r}) = \mu_0$. This

allows for the simplification of the above equations and gives the numerical solution for the wave equation (substitution of 1.2 in 1.1):

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

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 forward, this will be used as a basis to adapt the 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 one can get a system of equations that are further used to approximate each solution 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.[19]

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.[31] 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 there exists 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.[9] It is also worth noting that FIT can be used for deriving the update equations that will be used in the FDTD algorithm. 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.

1.3 Finite Difference Time Domain Method

FDTD was first proposed by Kane Yee in a 1966 paper[32] 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 half a mesh skip in each coordinate direction)
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. The next section 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.

1.4 FDTD Implementation

Before beginning with the discretization, it was mentioned previously that Maxwell's Equations can be shown in both their differential form and their integral form. Using this form has the benefit that the equations are easier to work with and follow. Using integrals instead of derivatives, however, does mean that, at least for the derivation, FIT is being used instead of FDTD. Despite that, the final equations will be the same. Here is the integral form of Maxwell's Equations:

$$\oint \vec{E} \cdot d\vec{s} = -\frac{d}{dt} \iint \vec{B} \cdot d\vec{A} \tag{1.8}$$

$$\oint \vec{H} \cdot d\vec{s} = \iint \frac{d\vec{D}}{dt} \cdot d\vec{A} \tag{1.9}$$

$$\oiint \vec{D} \cdot d\vec{A} = \iiint \rho dV \tag{1.10}$$

$$\oiint \vec{B} \cdot d\vec{A} = 0 \tag{1.11}$$

The material relations will look as follows:

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

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

By plugging 1.13 and 1.12 into equations 1.8 and 1.9 respectively:

$$\oint \vec{E} \cdot d\vec{s} = -\frac{d}{dt} \iint \mu \cdot \vec{H} \cdot d\vec{A} \tag{1.14}$$

$$\oint \vec{H} \cdot d\vec{s} = \frac{d}{dt} \iint \epsilon \cdot \vec{E} \cdot d\vec{A} \tag{1.15}$$

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, the above functions will need to be implemented into a program that can generate approximate data on electromagnetic waves, which will be discussed in the next section.

1.4.1 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 standard C++, 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 material properties 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 (vacuum) ϵ_0 and μ_0. Also, these values are going to be constant throughout the simulation, meaning it will feature 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 one was 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 very short pulse, thus providing a near useless visualization once the data is put into an appropriate program.

Instead, the Gaussian pulse excitation can be used in the middle of the domain, and have it propagate throughout it. This is ideal because the simulation is using reflective boundaries, meaning the pulse will bounce back and forth between each boundary without any loss. If one was 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.

A Gaussian pulse will be used for this 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 this project would want to avoid the use of header files that are not included by default in C++ is that such files could make the program dependent on things such as 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, the application should support anything from one to three dimensions. While only a 3D simulation will be needed for most practical applications, 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, certain limitations that plague all computers, simply due to their nature, will have to be explained. Since these limitations cannot be bypassed as of yet, one can never achieve an exact, perfectly realistic simulation. Thus, it is good to keep them in mind while developing such applications.

1.4.2 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 or compiler of choice, and then passed down in the form of a lower level language. This process occurs more than once too, until one gets 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. This would be best explained with an example: summing fractions 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 a person knows is correct. However, when one runs 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";
}
```

one 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 the code, the total sum is not equal to 1.0, even though it should be.

What is happening is that a computer is being asked to store the infinite number $1/9 = 0.\overline{1}$ using a finite amounts of bit. Depending on the data type that is used, it can have anywhere from 8 bits of precision, to 2^8 bits. However, that is still a finite amount, and no matter what one does, 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 a good idea of what to expect. Even more so when considering that there is an error with even bigger margins that must be discussed.

1.4.3 Discretization Error[25]

In computer simulations, by far the most noteworthy type of error is the Discretization Error. This error occurs whenever a discretization technique is applied. In more conventional words, it occurs whenever a continuous function or phenomenon is calculated as a finite number of evaluations using a computer (Figure 1.4).

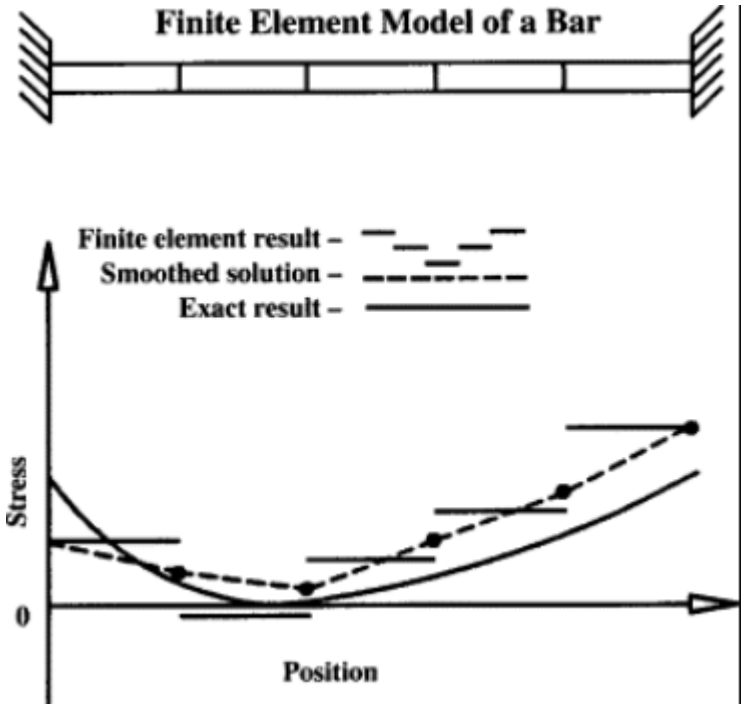


FIGURE 1.4: The difference between various estimations and the exact result[11]

This type of error is inevitable whenever approximation techniques are used.

In FDTD, it is the difference between the exact partial differential equation and the discretized algebraic equation used by the implementation. It can theoretically be calculated by taking the value of the exact solution and comparing it with the value received by the numerical approximation. In practice, however, doing so can prove difficult depending on the scenario. Such an error can also be observed by changing the domain mesh size. The larger the mesh, the larger the error would be.

Therefore a balance should be achieved between having a small enough number of mesh cells and having a short enough computational time. Another compromise would be a variable mesh size if the areas that need high precision are known beforehand. Keeping a coarse mesh in general and only refining it where needed could aid in reducing the error margin considerably. Alternatively, one could look at refinement methods. However, they too bear a computational cost, and will therefore not be used here.

2 FDTD - One-Dimensional Scenario

This chapter will go more in depth into developing an application that can generate electromagnetic data in a one-dimensional domain. Previously, a series of steps to implement FDTD and that a part of them depend on the particular implementation was mentioned. 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

The previous chapter concluded 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. Throughout this section the update equations that will be used in the loops will be derived. Before moving on, Transverse modes must first be briefly discussed. 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 this scenario, TEM mode will be used for discretization, meaning neither electric nor the magnetic field have field components in the direction of propagation.

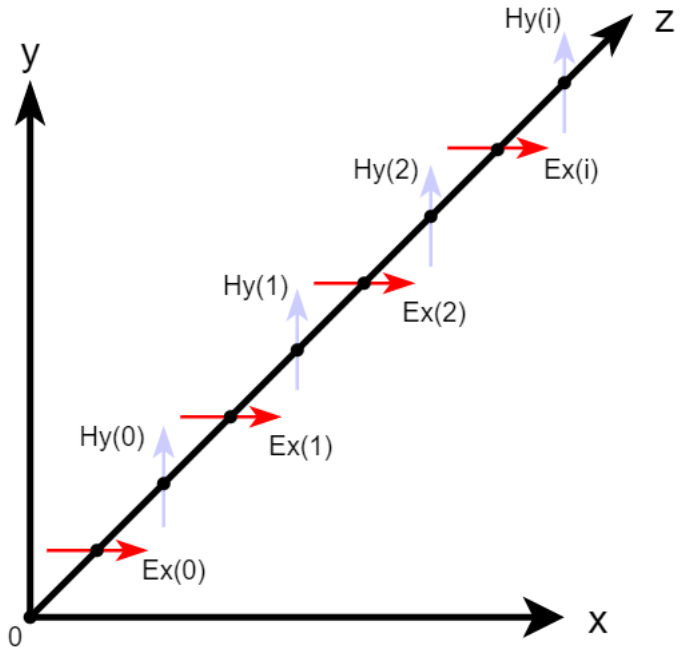


FIGURE 2.1: The spatial and temporal shift of a one-dimensional electromagnetic scenario.

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. There are two such curls in the 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 electric fields and the magnetic ones.

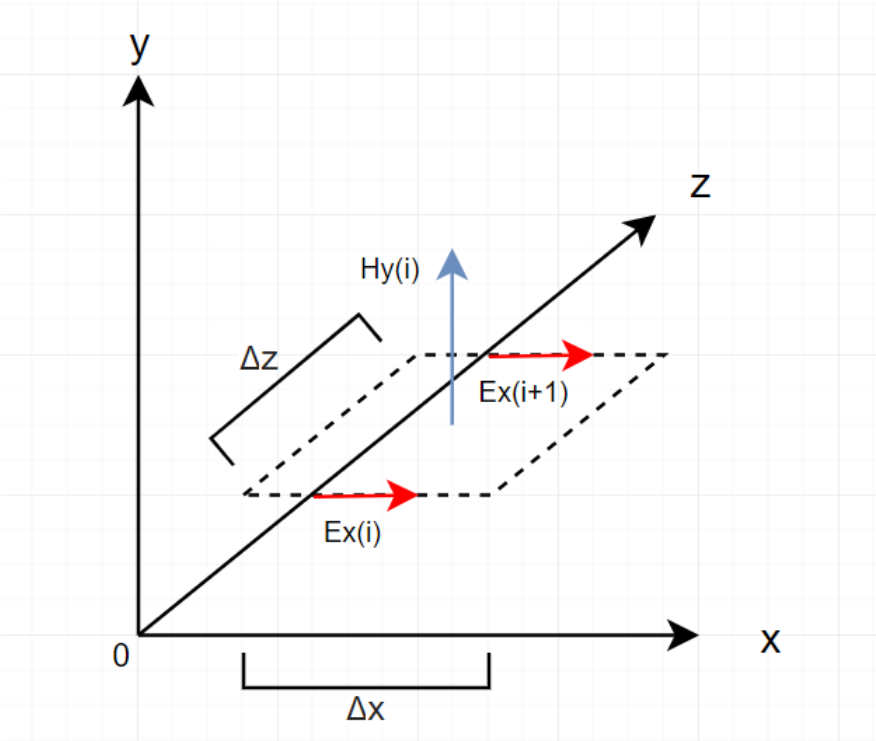


FIGURE 2.2: A graph showing the one-dimensional curl around the vector $H_y(i)$.

Figure 2.2 shows the curl around the magnetic vector $H_y(i)$. This curl can be used by plugging 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 there is no Ez vector in the one-dimensional TEM scenario, $Ez \cdot \Delta z = 0$. Therefore the equation 2.1 can be simplified 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:

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

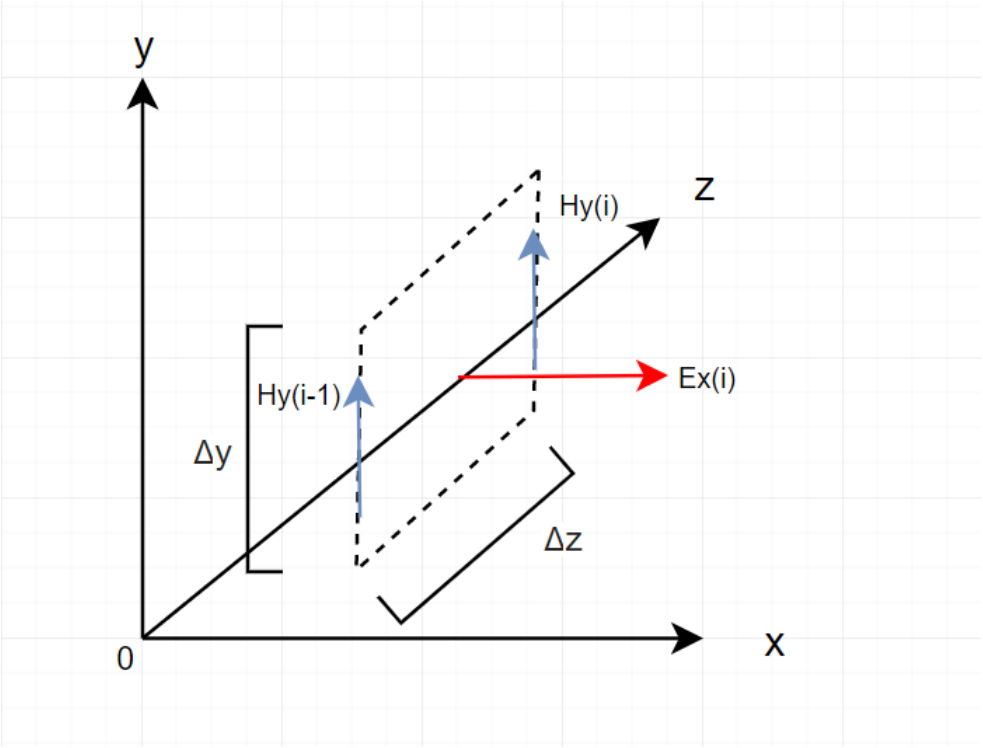


FIGURE 2.3: A graph showing the one-dimensional curl around the vector $E_x(i)$.

By combining 2.2 and 2.3:

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

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

The same thing can be done the 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 equations 2.5 and 2.9, one can now use the leapfrog time scheme to stagger these components along the time axis (Fig. 2.4).

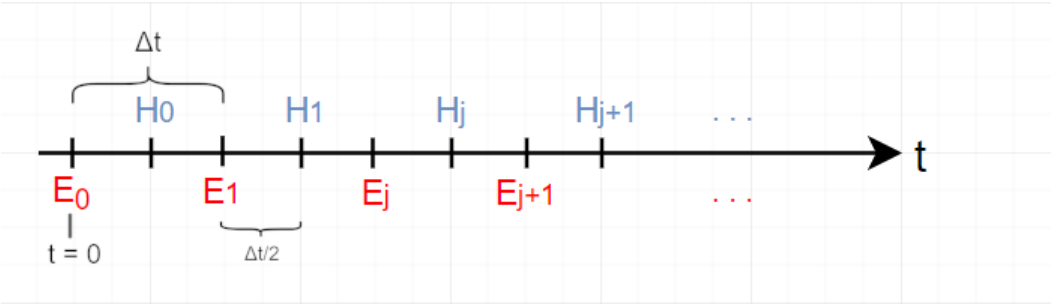


FIGURE 2.4: Staggering the components along the time axis t using the leapfrog time scheme. Here the index indicates the time sample of the field.

The following indexing scheme will be used:

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

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

Using 2.10 and 2.11 results in 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:

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

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

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

Finally, by solving for $H_{y^{i,j}}$ one can get the 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, the update equation for the electric element can be obtained by starting from the equation 2.9. The result is:

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

Before moving on, it is critical to discuss the stability of the simulation, which relies on the value of Δt.

2.1.3 Stability and Energy Conservation

Deciding the mesh size for a given domain is heavily based on the computational capabilities of the instruments that are being used, rather than the implementation itself. Therefore, it is fairly straightforward most of the time: one should make the mesh as small as possible, provided the implementation can finish computing within an amount of time that can be considered reasonable.

Unfortunately, this is not the case for the time step size, Δt. While it can be chosen arbitrarily, picking a number that is too large can cause the implementation to grow unstable (Figure 2.5).

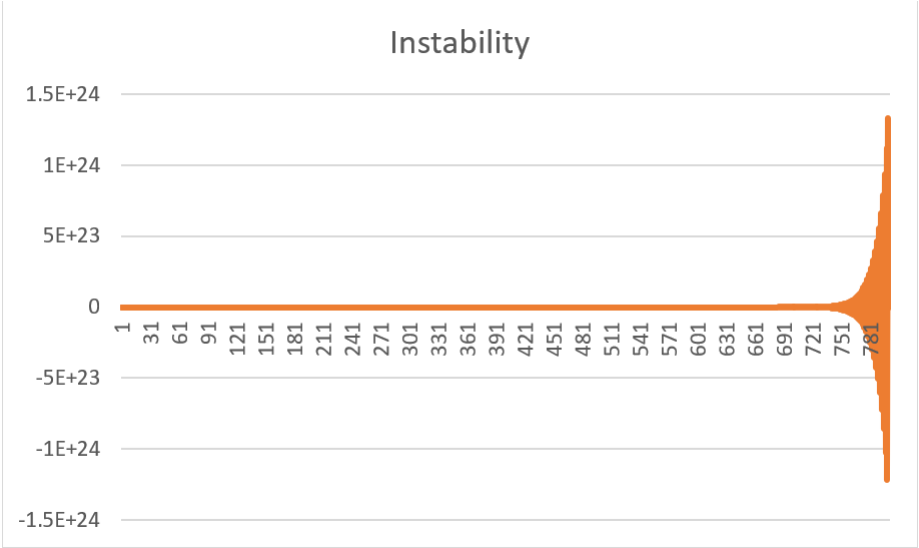


FIGURE 2.5: The instability that can occur when choosing a Δt number that is too high.

There is a max possible value for Δt called Δt_{max} that is derived from the Courant–Friedrichs–Lewy condition, which can be found using the following equation:

$$\Delta t_{max} = \sqrt{\epsilon \cdot \mu} \cdot \frac{C_{max}}{\sqrt{(\frac{1}{\Delta x})^2 + (\frac{1}{\Delta y})^2 + (\frac{1}{\Delta z})^2}} \tag{2.18}$$

The part with the deltas under the square root depends on the dimensions of the domain. Therefore, for this one dimensional scenario, equation 2.18 can be shortened to:

$$\Delta t_{max} = \sqrt{\epsilon \cdot \mu} \cdot \frac{C_{max}}{\sqrt{(\frac{1}{\Delta z})^2}} \tag{2.19}$$

C_{max} is a number that heavily depends on the method used after the main problem is discretized. In the case of explicit schemes such as FDTD, usually it is $C_{max} = 1$, meaning equation 2.19 can be simplified further:

$$\Delta t_{max} = \sqrt{\epsilon \cdot \mu} \cdot \frac{1}{\sqrt{(\frac{1}{\Delta z})^2}} \tag{2.20}$$

$$\Rightarrow \Delta t_{max} = \sqrt{\epsilon \cdot \mu} \cdot \Delta z \tag{2.21}$$

So long as $\Delta t \leq \Delta t_{max}$, the implementation will stay stable.

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 the implementation begins, one needs to prepare the coding environment. For this project, the author is using the Eclipse IDE for C++ Development[12]. After the creation of a new project, the code file will start off with an empty skeleton that features the *main()* method. Due to not needing any other classes, this is where all the code will be contained. Before that, one will need to include some standard C++ headers so that their methods, data types, and variables, can be used.

```
#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**[3] - Header files 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**[4] - Allows for the usage of input and output stream objects and commands. In the 1D implementation, the program will print the data to the console by using `cout`
- **stdlib.h**[5] - Provides helpful data types and functions, and is part of **vector**.
- **vector**[6] - Arrays in standard C++ are not dynamic. Once populated, they can no longer be modified. That is why for this implementation vectors need to be used.
- **string**[7] - Includes the string datatype. A string is basically an array of characters. Not only can one use this to format the output more easily, it can also be very useful for printing debugging messages.

Next, some variables will need to be initialized. The permittivity ϵ and permeability μ of the domain that is going to be simulated need to be set first. For this scenario, the values for vacuum are going to be used, ϵ_0 and μ_0.

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

These numbers are normally infinite, therefore innacuracies are already being introduced into the simulation. For greater precision, one could include more decimals, though the difference for such a small case would still be negligible. It should be noted that although the units are not mentioned, all the calculations use standard SI units. 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 the domain was split. **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 it would be desirable to evenly split the domain while also covering it whole, and the best way to do so would be to set it to L/N (so long as the mes is uniform).

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, one can choose a value that will always work through the equation below as described above:

$$\Delta t = \Delta z \cdot \sqrt{\epsilon \mu} \tag{2.22}$$

This is the one-dimensional version of this equation. It will be seen how it changes later on when discussing the two-dimensional and three-dimensional scenarios. Now that the testing environment is ready, one will need to initialize the variables that will be used in these 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 the data for each time step. For the one-dimensional scenario, in order to display the simulated data a time graph will be used, which is a graph of the values that an arbitrary point n has during the simulation. A good choice for this scenario would be the midpoint of the domain $N/2$, and the vectors **tE** and **tH** will contain the data of the electric and magnetic time graph respectively. It is also prudent to 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.

Now that the setup of the variables that will be needed for the simulation is done, it is time to actually move on to the *main()* method where FDTD

algorithm will be implemented. First, the magnetic and electric field vectors need to be populated, since this was not done as of yet. Since the simulation is going to have no initial charge in it, every point in these vectors can be initialized to 0. Luckily there is a quick way to do so in C++ that does not require the use of loops:

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

This will push N zeros to the vectors. Now one will need to start looping for

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

values. It can also be helpful to start off the 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, by using the update equations that were 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]);
}
```

It should be noted that the starting index of a vector is zero, therefore when adding N values to these vectors, the index of the last value is going to be

$N-1$. In these loops, this number must never be surpassed. For the magnetic vector H loop, it would need to stop when $z+1 = N-1 \Leftrightarrow z = N-2$. Alternatively, one could also populate $N+1$ values, and extend the domain one unit past the boundaries.

After the above update loops, but while still inside the main loop, the middle values will be stored 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, one can pick any point for this and it would still work. This is also another reasons why vectors are incredibly useful: since the vectors **tE** or **tH** were not populated with any values previously, they had a length of zero. However, using the *push_back()* method, the values can be added dynamically, therefore allowing to change the number of iterations at will (Note: For performance reasons it might be better to reserve the whole vector for the time graphs on initialization. Otherwise the whole vector will be copied over and over again. For the current hardware and parameters this performance boost is negligible, however it might prove useful in less powerful machines).

After running the main loop **iterNum** times, the values are finally printed 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 2.6.

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

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

FIGURE 2.6: The console output of the FDTD one-dimensional algorithm

The time graphs can be considered to be the field values that a single cell has throughout the simulation. They provide an easier way to visualize moving simulations, due to being capable of showing all the data at once.

Now that the data was generated, it can finally be visualized.

2.3 Data Visualization

Generating the data was the difficult part. In order to visualize it into a form that is easily understandable by humans, a variety of programs can be used. An easy solution for this is Microsoft Office Excel[20]. While this software requires a license, there are viable alternatives to it that should be able to achieve the same results.

For starters, the comma separated list that is in the console output, comprising of the data inside the time graph vectors for the midpoint of both magnetic and electric fields, needs to be copied 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, one can go to **Data > Text to Columns** and then split the one cell list into multiple columns (2.7).

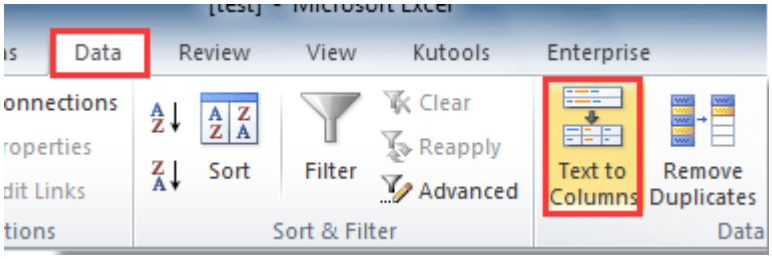


FIGURE 2.7: Transforming data from a single cell containing the whole list, into separate columns for each value.

After doing that for both electric and magnetic values, one 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 2.8: Transforming data from a single cell containing the whole list, into separate columns for each value.

With this, the data can be visualized however it is desired. For the purposes of this demonstration line graphs will be used. In Figure 2.9, the time graph of the electric field was colored orange, and the magnetic field blue.

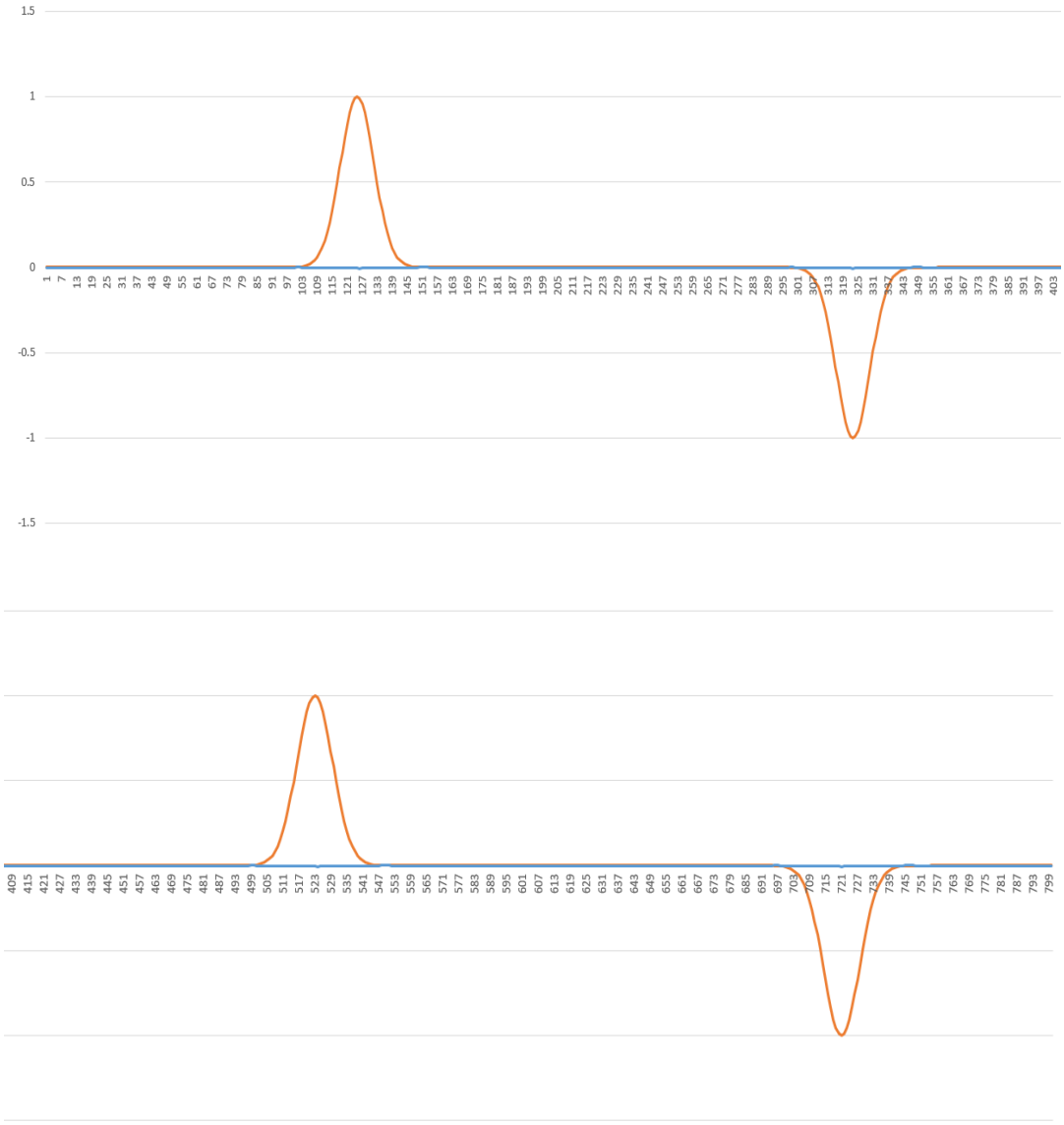


FIGURE 2.9: The time graph of the electromagnetic data generated by the 1D application.

It should be noted that both lines were shown in the same plane due to convenience. The waves themselves move as they did during the discretization done prior: electric waves move in the **X-Z** plane while the magnetic waves move in the **Y-Z** plane. Also, it can be noted that there is an enormous difference in the values of the electric and magnetic fields. One 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. 2.10):

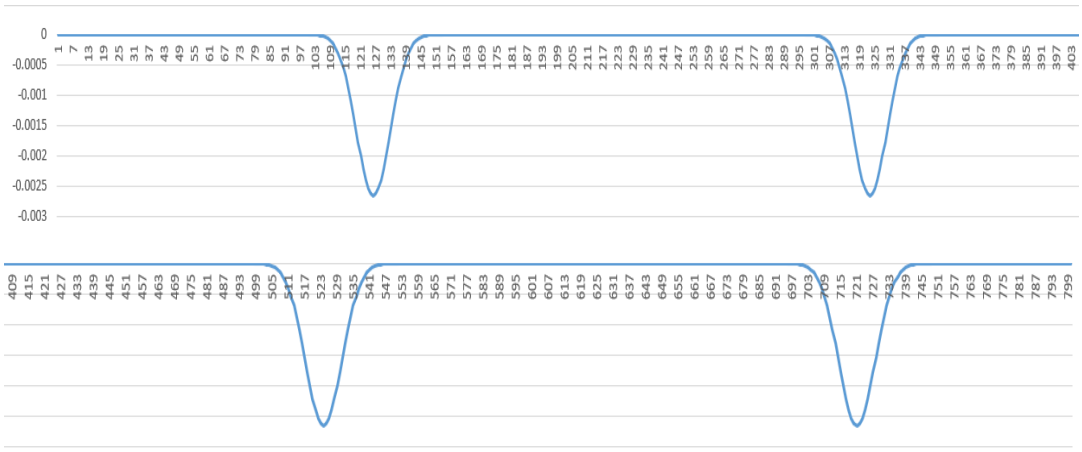


FIGURE 2.10: A snippet of the time graph shown in Figure 2.9 zoomed in

The reason for this difference being so big is due to the medium chosen for this 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:

$$Z_0 = \frac{E}{H} = \sqrt{\frac{\mu_0}{\epsilon_0}} \tag{2.23}$$

Something else worth noting is the distance between each wave. If the midpoint is chosen as the observation point for the values of the time graph, the

distance between the starting point of each wave will be precisely N, the size of the domain. If one decided to use a different point of observation, the distance between each wave would vary. However, the distance between the first wave and the fourth, the second and the fifth, and so on, would always be $N \cdot 2$.

Next comes the two-dimensional scenario.

3 FDTD - Two-Dimensional Scenario

Now that the one-dimensional scenario is complete, the following is going to be much easier, as for the most part the same logic is going to be used again. 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 the different transverse modes available for electromagnetic waves were mentioned briefly without going too much in depth as they were not so relevant in that scenario. In a one-dimensional case, it does not matter much which axis is picked for which field, so long as they are perpendicular to each other. In the two-dimensional scenario however, the mode that is choosen is going to dictate the curls that will be used.

3.1.1 2D Transverse Modes

For the two-dimensional scenarios, choosing a transverse mode means that the chosen field will have no vectors in the direction of propagation, meaning the z axis in this 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** - $E_x,\ E_y,\ H_z$
- **TM mode** - $E_z,\ H_x,\ H_y$

While each mode will have different vectors, the process is roughly the same. For this scenario TE mode will be used, meaning that $E_z, H_x, H_y = 0$.

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:

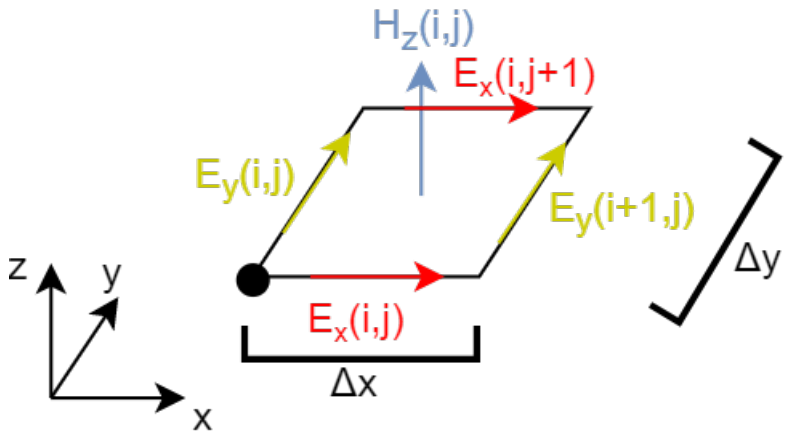


FIGURE 3.1: The curl around the magnetic field vector H_z.

Using a similar indexing scheme as done previously:

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

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

$$\Rightarrow E_x(i,j) \cdot \Delta x - E_x(i,j+1) \cdot \Delta x + E_y(i+1,j) \cdot \Delta y - E_y(i,j) \cdot \Delta y = -\frac{d}{dt}(\mu \cdot H_z(i,j) \cdot \Delta x \cdot \Delta y) \quad (3.2)$$

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 there is a uniform mesh size, meaning that $\Delta x = \Delta y = \Delta z = \Delta s$, one 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, it is known that:

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

Combining equations 3.4 and 3.5 results in the 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:

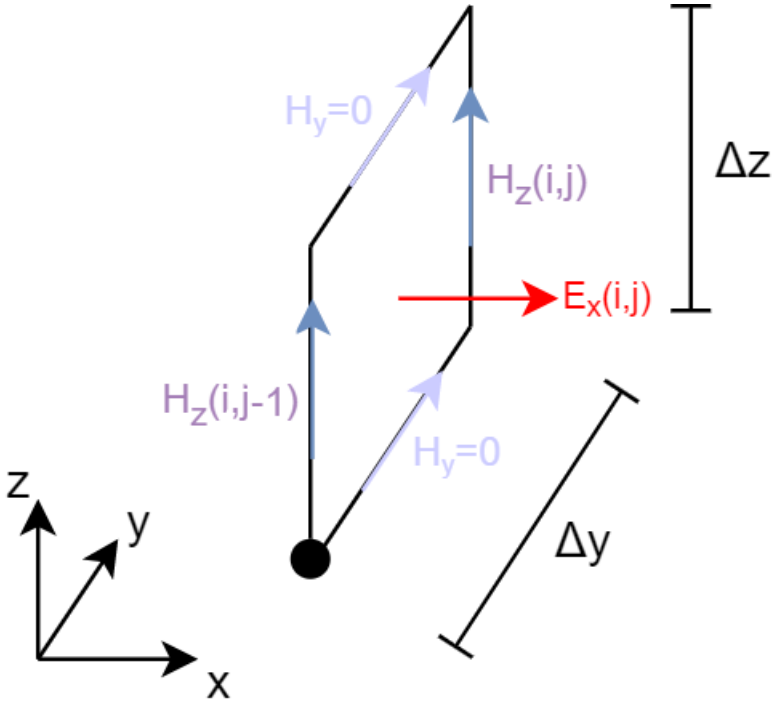


FIGURE 3.2: The curl around the electric field vector E_x.

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):

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

The code implementation can now begin.

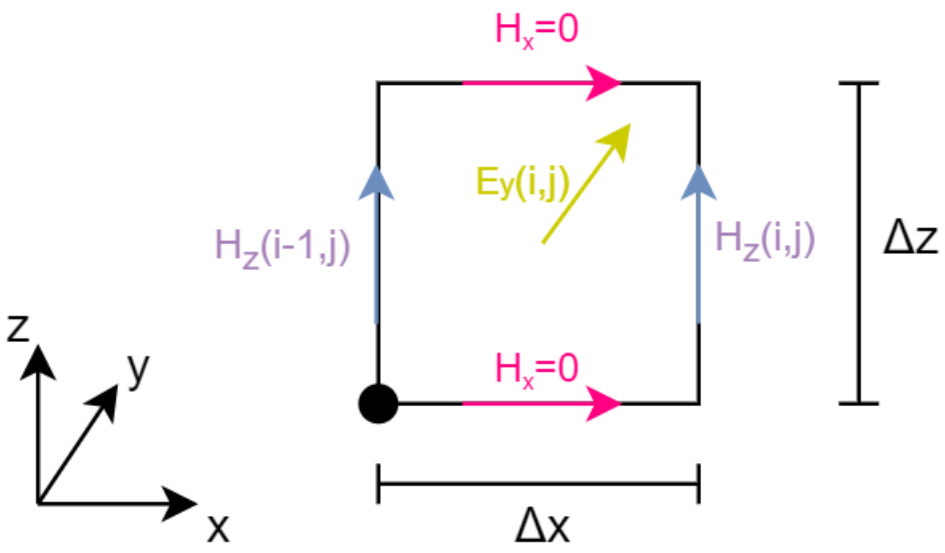


FIGURE 3.3: The curl around the electric field vector E_y.

3.2 C++ Implementation

Starting off with the imports, the ones that were already discussed will not be explained again. There are however new headers that will be needed:

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

For the two dimensional scenario, a new method of visualizing the data will be used. The end result will be an animation that shows how the waves propagate in real time. It will be explained more in depth once data visualization is described, but for now all that is needed to be known is that the output of the two-dimensional implementation is going to be CSV files that contain the data. The header files above will help with that:

- **fstream**[8] - Input output stream operations for files. Allows users to create and modify them.
- **io.h**[14] - Header file used by **fstream** among other standard library files.

Two new C++ functions to write the electromagnetic data into files need to be introduced:

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

        //        x,y,z,Ex,Ey
        //        i,j,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, const vector<vector<double>>
↪  &Hz){

        //        x,y,z,Hz
        //        i,j,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 < Hz[x].size(); y++) {
```

```
                        csvFile << to_string(x) + "," + to_string(y) +
                        ↪  ",0," + to_string(Hz[x][y]) + "\n";
                }
        }

        csvFile.close();
}
```

These functions will be called every 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/";
```

Some changes need to be made to the code to adapt it to a two-dimensional environment. For starters, the vectors need to be initialized as two-dimensional. The electric field will also require an extra vector now for a total of two, one for the *x* axis and one for the *y* axis. In total, the following vectors need to be declared:

```
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 one would also need to add new variables for the deltas:

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

Since a uniform mesh is being used, meaning all the spatial steps have the same dimensions, only one of these deltas needs to be used (e.g. *deltaZ*).

In the Appendix (A) implementations for unequal mesh step sizes will be discussed further. For now, since only one delta will be used, Δt will need to be changed to:

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

Another difference is that the Gaussian pulse excitation is going to be applied in the center of the magnetic field this time. Due to the impedance of vacuum, it is prudent to promptly reduce the magnitude of this excitation, as otherwise the electric values would 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 result in a nice symmetric propagation towards the edges. In symmetrical scenarios one can save computation time by splitting the domain into equal parts. It is even more helpful with circular symmetry as the domain can effectively be split into as many parts as needed, and by calculating the wave propagation for only one part, those values can be then replicated for the other parts. This will also be discussed in the Appendix.

By translating the update equations into code, the main loop becomes:

```
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 each vector field is done, one can call the respective method to write the data into csv files. For organizational purposes, this data will be dumped into separate folders for each field. After waiting for code execution to finish, there will now be 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, it would help to use an open source program called Paraview[22]. It allows to build animated visualizations with the data that was generated. It is also the reason for the chosen naming scheme for the output files. By using the format *X.csv.n* they can all be imported as one object. This can be easily done by simply opening Paraview, selecting all of the data in one of the output folders, and just dragging and dropping it into the "Pipeline Browser" panel, pictured below (Figure 3.4).

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

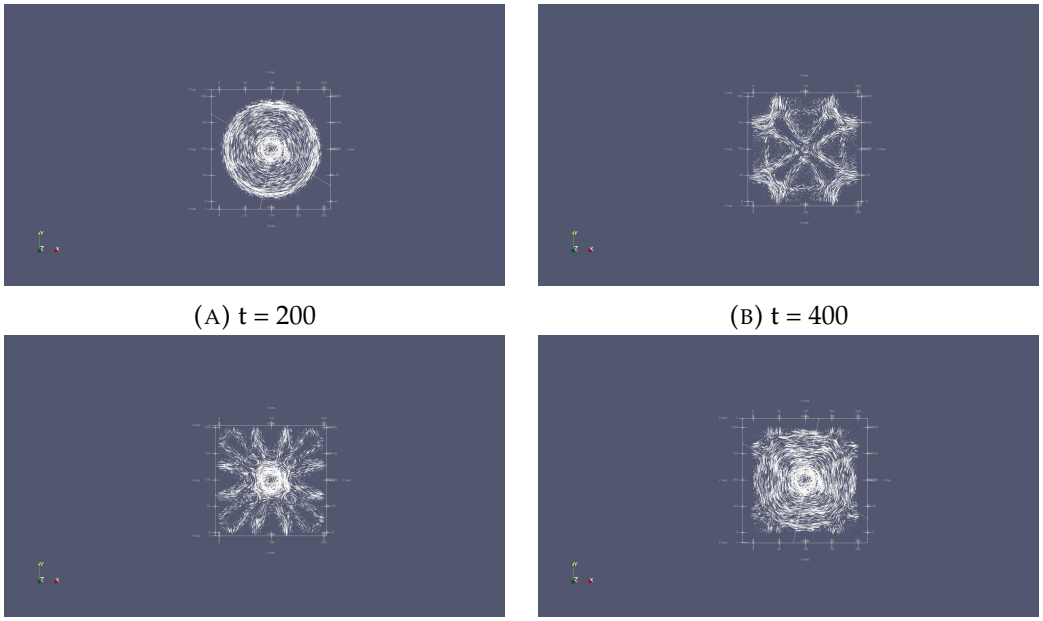
Having just the data by itself will not do much however. That is why one needs to apply filters to the data. The ones that will be used in this case are the following:

- **Table to Points** - Simply displays the CSV data as points.
- **Calculator** - Allows 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**, the vector components for the data needs to be defined. In the output files, one will notice that not only x and y have been specified, but also z, which contains only zeros. That needed to be done that since Paraview will not accept less than three columns here. It is also helpful to tick **Axes Grid** under the **Annotations** category. In the **Calculator**, one needs to add the following formula

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

which will build the vector data from the components. Lastly, for the **Glyph**, the **Result** array gotten from the **Calculator** needs to be used as both **Orientation** and **Scaling**. Depending on whether the data is visible or not,the scaling slider can be changed as desired. If done correctly, one will get a result similar to Figure 3.5.

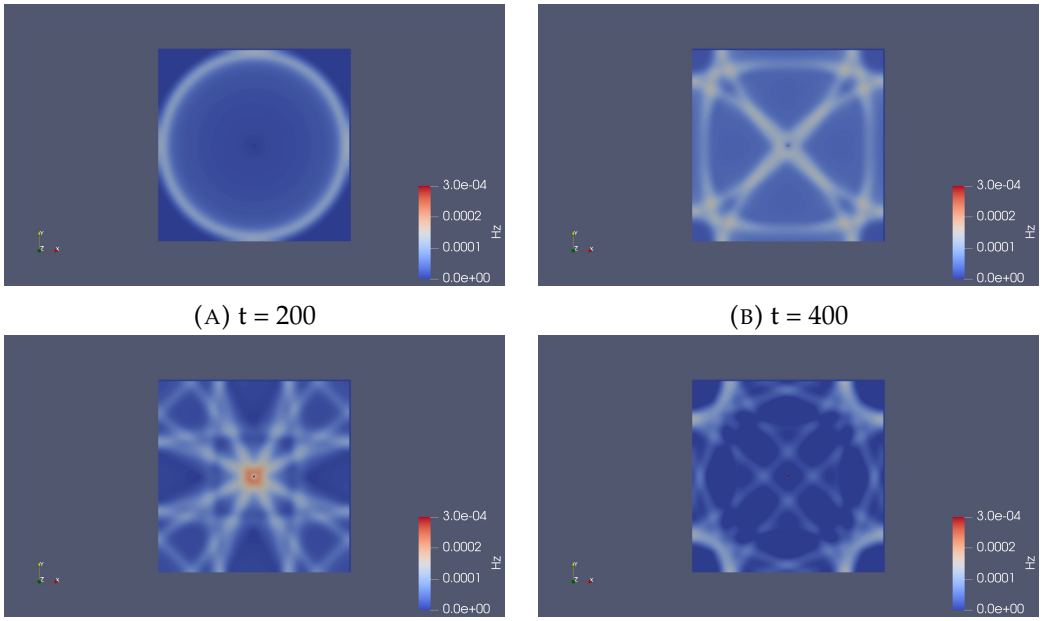


(A) t = 200 (B) t = 400

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

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

The above is for the vectorial electric data. For the scalar magnetic data it may be more simple to use **Table to Structured Grid**. With a few adjustments to colors and scaling, the following result can be achieved (Figure 3.6):



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

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

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

Moving forward, only the visualizing method for the electric data above can

be of use, as the **Table to Structured Grid** method is not applicable when using more than one vector component. With that done, the three-dimensional scenario can finally be discussed, being the most useful one for practical real life applications.

4 FDTD - Three-Dimensional Scenario

The three-dimensional scenario has 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 to deal with has doubled. This scenario is also where optimization really matters, considering that there will be $6 \cdot N^3$ amounts of data per time step. Nonetheless, if one focuses on just the most basic implementation, taking this step by step, and splitting the effort into small parts, it will be easy to move forward.

4.1 3D Discretization

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

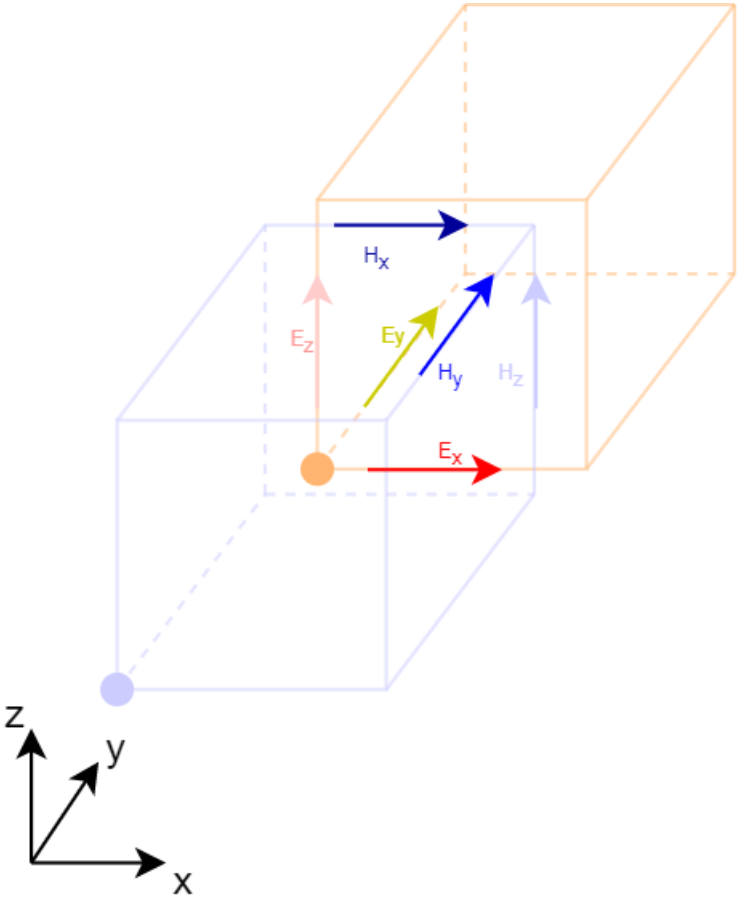


FIGURE 4.1: The figure shows the vectors for the electric field, with the respective magnetic vectors.

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

Despite the fact that the image above is fairly cluttered with information, only bits of it are needed at a time. A keen eye will notice in the next section that the electromagnetic curls for the three-dimensional scenario can be derived in the same way that was used for the two-dimensional scenario.

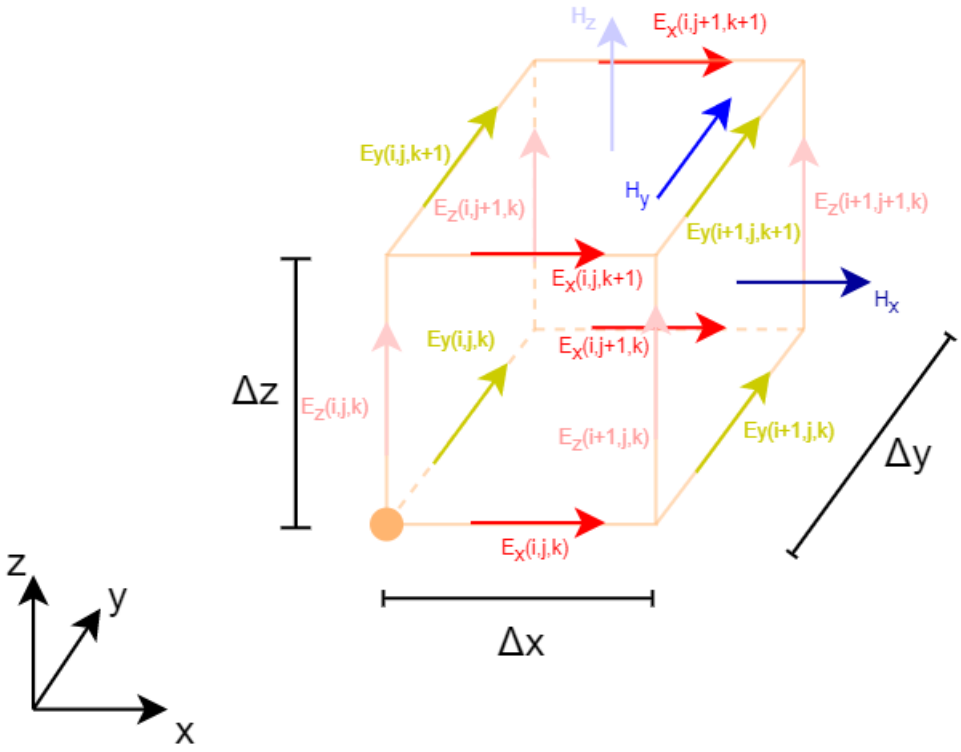


FIGURE 4.2: The figure shows the vectors for the electric field, with the respective magnetic vectors.

4.1.1 3D Electromagnetic Curls

In the previous chapter, the vector curl for H_z was derived, resulting in the update equation for the magnetic field. Luckily the curl is exactly the same in the three-dimensional scenario, however the formula needs to be revised 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, the same process as before can be repeated.

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

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

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

Using a uniform mesh size again, $\Delta x = \Delta y = \Delta z = \Delta s$, equation 4.3 can be simplified 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, one can obtain the update equation for the H_y curl in Figure 4.3

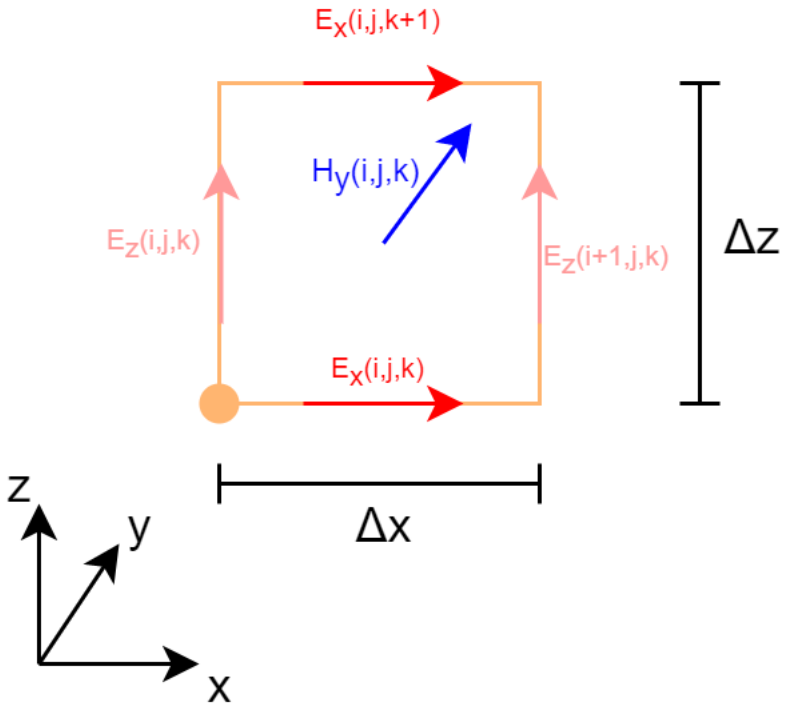


FIGURE 4.3: The curl around the magnetic field vector H_y.

It can be noted 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.

Once more, as the case was in the previous chapter, the methodology here is the same. One must however now account for not only the added dimension, but also the fact that previously one of the magnetic vectors was zero. Starting off from the initial Maxwell equation:

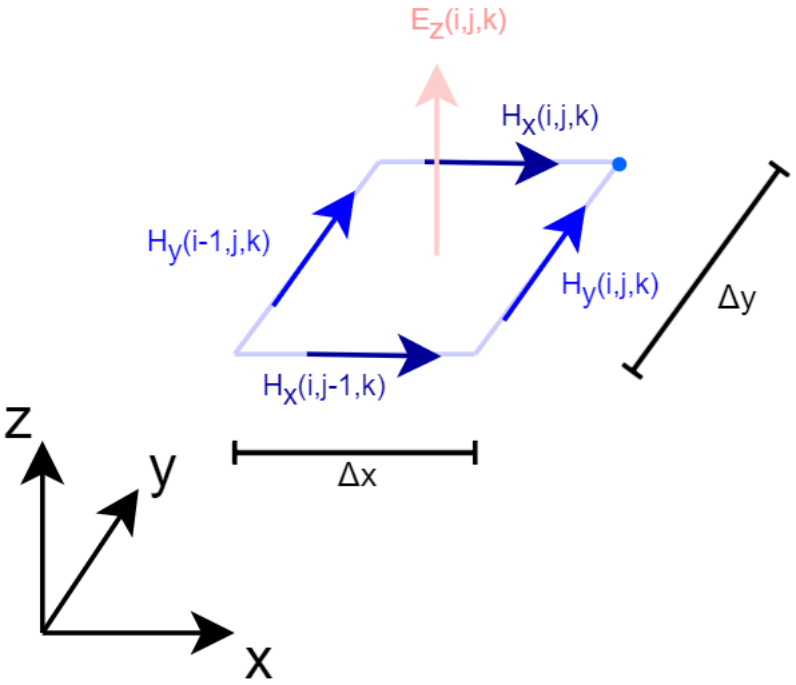


FIGURE 4.4: The curl around the electric field vector E_z.

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

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

Again taking into account the uniform mesh:

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

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:

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

One could proceed to manually do the curls for the remaining field vectors, however with a keen eye a pattern can be noticed in the equations above, 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 $a_1 = y$, 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 the calculation is being done 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

the equation for the electric curl is being calculated, the sign will be $+$. Otherwise it will be $-$. The updated index, meaning the one that will have the ± 1, will be the index that belongs to neither the vector that the equation is for, nor the vector to which the curl belongs to.

The H_x vector curl, shown in Figure 4.5, can be used to test this pattern.

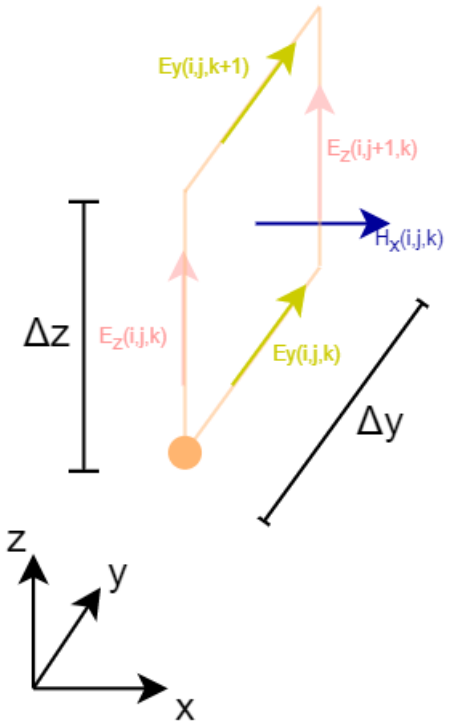


FIGURE 4.5: The curl around the magnetic field vector H_x.

From the image, the following can be deduced:

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 this is the curl for H_x, that means the z index is being increased by one, meaning that C_1 needs to be replaced 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 the result would be:

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

On the right hand side, the $-$ sign the second half of the equation can be flipped 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:

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

Therefore, the 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, one can derive the update equations for the vectors that have not been determined yet: E_x (4.19) and E_y (4.20).

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

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

Next is the code implementation.

4.2 C++ Implementation

Starting from the two-dimensional implementation, few changes need to be made in order to adapt the program for three dimensions. If the previous code is used as a basis, then all the header files that are neeeded as well as the basic code structure are already there. One 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 the amount of update loops increased from N^2 to N^3, but now there are also six three-dimensional C++ vectors instead of three two-dimensional ones. It is recommended to start from a small number and go up for there. The hardware that is available to the author of this project can handle around this much, so these values will stay as they are 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.

Now that there are three dimensions, the *deltaT* equation will also need to be updated:

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

Furthermore, six three-dimensional C++ vectors are needed for the 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 the functions that create CSV files for the data need to be modified. Now that there is an equal number of vectors for both the electric and the magnetic wave, only one function is needed to handle both fields:

```
void writeDataToCsvFile(string filename, const
↪  vector<vector<vector<double>>> &Vx, const
↪  vector<vector<vector<double>>> \&Vy, const
↪  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 the main loop, one can choose to apply the Gaussian Pulse excitation to either an electric field vector or the magnetic one. Both work, however when using the magnetic field for the excitation, it may be necessary to lower the magnitude if using vacuum as a medium. Depending on the material used in the domain, this may be unnecessary. Once more, applying this excitation in the middle of the vector results in a nice symmetrical wave propagation.

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

Only one of the field vectors is needed to apply the excitation, however which one is chosen will affect the order of the 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, the electric loops would need to come first. It bears repeating that the loop indexes should not go out of bounds for the magnetic field equations.

With that said, after running the above program one should have 200 files (indexed from 0 to 199) that can dropped in Paraview as before for visualization.

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

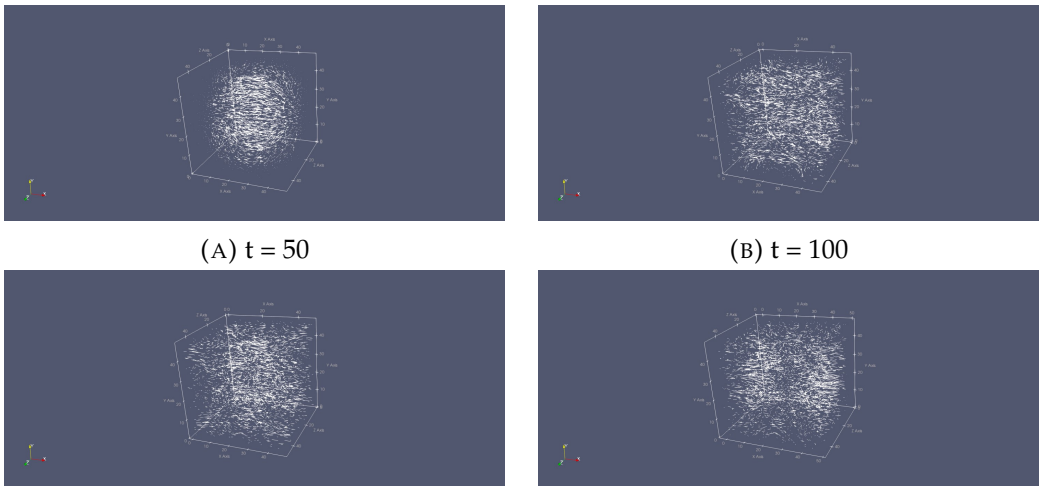
```
iHat*Vx+jHat*Vy+kHat*Vz
```

It should be noted that Vx,Vy, and Vz, are taken directly from the first line of the CSV files. If that header is different, it needs to be reflected here. In this case, the first line is *"x,y,z,Vx,Vy,Vz"*.

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

The same can be done for the magnetic data as well by following the same steps. It should be noted that this data will need to be scaled quite a bit more than its electric counterpart, as it would be quite hard to see otherwise due to using vacuum as the medium.

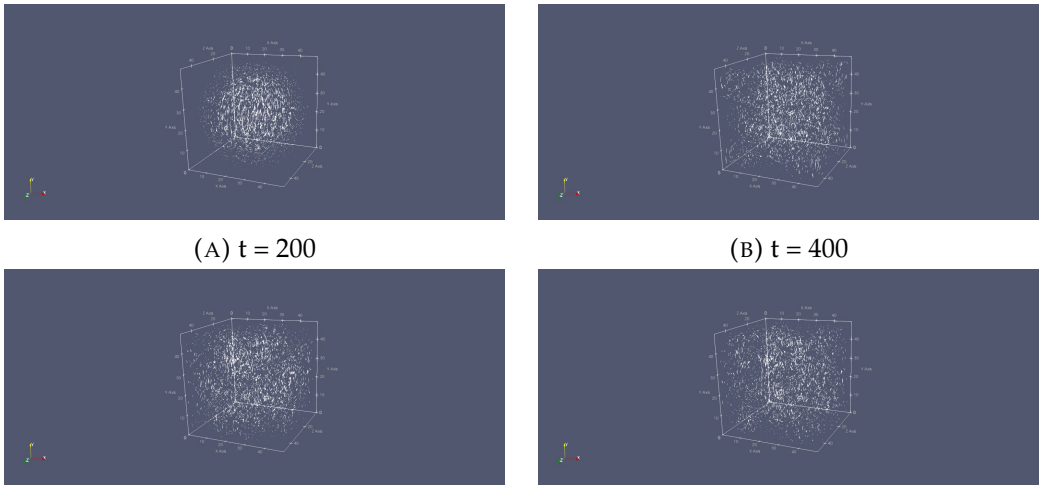
With that, the simulation for a three-dimensional scenario, and also this project as a whole, is complete.



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

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

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



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

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

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

5 Conclusion

Through the help of many scientists and researchers, this project was able to simulate electromagnetic wave data for three scenarios using a simple implementation that is easy to adapt and expand upon. At first, it explained the value of simulations in research, but also in various industries. It showed examples of capturing the movement of a light particle[30], or being able to capture an image of a black hole[17], 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 humans were able to know the existence of black holes to begin with[13]. And it was computers that compiled multiple small images into one. Indeed the development of technology has allowed not only the achievement of such feats, but also the creation of simulations of various phenomena from scratch, which imitate nature closely, but not quite perfectly.

Unless humanity is able to create computers that can achieve infinite precision, something that right now is not possible, three will always be 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, this error can be minimized to a point where it is negligible.

For electromagnetic waves specifically, simulating their behavior gives 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. These equations were adapted in a FDTD implementation to create three C++ programs, capable of simulating electromagnetic phenomena in one-dimensional, two-dimensional, and three-dimensional cases.

First, the numerical solution to the Wave Equation was derived. With that, it is possible to choose from many algorithms as to how to proceed. FDTD was chosen here due to how simple it is and how it can be implemented in a realistic amount of time by only one person[9].

Initially called the Yee algorithm because it was proposed by Kane Yee, FDTD 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, the domain is discretized and formulas that are called update equations are derived, allowing for the next step of the respective field to be calculated.

This process was applied to all three scenarios, making changes where needed to account for dimensions. Electromagnetic curls were used to derive the update equations, which could then be used in the code implementations. After

the programs generated the necessary data, the output was used to visualize these simulations in real time, by using a program called Paraview[22].

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. The Appendix (A) will go over some possible improvements. There one 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

Below is a short list of hardware and software used throughout the duration of this project.

A.1.1 Hardware

The project was developed and run on a laptop with the following relevant system properties:

- **Operating System** - Windows 10 x64 bit version
- **BIOS Version/Date** - American Megatrends Inc. N.1.54, 9/17/2019
- **Processor (CPU)** - Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz, 2592 Mhz, 6 Core(s), 12 Logical Processor(s)
- **Graphics Card (GPU)** - NVIDIA GeForce RTX2070 with Max-Q Design
- **Hard Drive** - 2TB Solid State Drive (SSD)

While the hardware is not dated by any means, a more powerful computer will be able to achieve higher computational speeds allowing for the generation of more data, particularly in the three-dimensional scenario where the bottleneck becomes obvious. In order of importance, this implementation relies on the CPU first and foremost for the calculations, having a fast hard disk such as an SSD in order to write the files quickly (the program does *iterNum*

file operations and the files can become quite large in size), and a GPU, which Paraview uses to visualize the data.

This order of importance can be changed by using the GPU for the implementation, something that is highly advised and is explained further in Section A.6.3.

A.1.2 Software

The following software was used in the implementation of the FDTD algorithm and the writing of this thesis:

- **Eclipse IDE for C/C++ Developers**[12] - C++ development environment that offers helpful features such as code completion, etc. Base installation may not come with a compiler by default.
- **GNU Compiler Collection (GCC)**[27] - Required to compile C++ code, in case the IDE does not have a compiler by default.
- **Paraview**[22] - Open-source, multi-platform data analysis and visualization application
- **TeXstudio**[33] - Program that is useful for writing LaTeXdocuments easily.
- **Online Diagram Editor**[34] - Free online editor for diagrams. Used to create images for this thesis, such as the vector curl figures
- **GNU Image Manipulation Program (GIMP)**[28] - Free open source image processing program

All of the applications above are free and offer their full range of capabilities without the need for a premium membership. They are not hard requirements for this implementation to work, with maybe the exception of the compiler. If one has access to an alternative they feel more comfortable

using, they could likely switch to that with little to no changes required in the implementation itself.

A.2 Troubleshooting the implementation

Unlike normal C++ implementations, troubleshooting this application is not straightforward at all. One can often be in a position where everything looks as if it is okay, yet the application still will not work. It can be easy to get stuck with a program that terminates with an error, or have the implementation work but the values are incredibly out of proportion. Debugging is also arduous, as there are multiple loops with thousands of data points, meaning that one needs to know where to look in order to even have the possibility to fix the issue. This section will help guide through the problems that the author faced throughout the development of this project.

First, it is helpful to use the console output to properly log the crucial parts of the implementation. This can be as simple or as advanced as one may desire, however it will prove immensely useful in tracking the cause of the issue. An example of logging would be to print to the console the mid point of a vector for each loop. While these console operations can increase computation time and put a strain on the hardware, once the debugging is done, they can be commented out until needed again. If one wants to take it a step further, they can use an open source C++ library or header file that offers logging support.

Second, as discussed in Section 2.1.3, it is crucial that an appropriate time step is chosen, else the simulation will become unstable and can even terminate improperly. Therefore, the first thing that should be checked is that the time step is small enough. If using the formula in Section 2.1.3, there should never be an issue with the time step, so long that it is implemented properly.

Another issue for instability would be an excitation that is implemented improperly. For the Gaussian Pulse, one needs to choose an appropriate ε value. In the main loop, the value of t needs to be updated constantly as well. It can be as simple as `double t = i * deltaT;`, where i is the current iteration. If the resulting values seem too high still, it can then be divided by an arbitrary number, though ideally this change would apply better to the value assigned to the vector, e.g. `Hz[99][99] = exp(-(beta * pow((t - gamma), 2))) * 10e-4;`. Also, it is important that for whichever vector one chooses to apply the excitation to, the opposite should be picked to start the loops. In this case, the E vector loops should be first.

Final problem could be an incorrect update equation. something as minor as a wrong sign can quickly cause an instability. Double checking the vector curls to make sure that they are done correctly can help in finding out where the issue is. Alternatively, if the equations are correct, yet the program still terminates erroneously, one should check that the indexes of the loops are not out of bounds.

A.3 Integration into a bigger program

This implementation is condensed purposefully into one file, and it relies on only standard C++ header files, specifically so that it can be imported to other machines and adapted to a larger scale project. This can be done with very few adjustments.

Starting off, a good idea would be to copy the code to a clean C++ file and into a class. Classes are the basics of Object Oriented programming, which is the best way to handle large projects where bad code separation can cause problems further down the line. It also helps in saving memory due to the

fact that the variables will only be initialized when the class is created, which is precisely when they need to be used.

The main method can then be changed to a normal function to call. The variables can either be initialized on class creation, or passed as arguments to the method. If the first option is chosen, then each variable should have a setter function. The values that need to be set for the implementation to work are *permittivity, permeability, L, N, iterNum, deltaX, deltaY, deltaZ, deltaT, eps*. The rest are derived from these values. These can either be initialized in the constructor, or in an initializer function in case one wants to only have one instance of the class, which is highly recommended. The rest of the implementation can stay as is. In case all three scenarios, i.e. for each dimension, are needed, it would be better to implement a separate class for each.

A.4 Using a domain with different environments

This implementation uses one medium throughout the whole domain for the sake of simplicity. The medium is characterized purely by the values of permittivity and permeability. These can either be chosen out of known materials, or entered manually, and they can be either absolute values, or relative compared to the values of vacuum.

As an example, water has a relative permittivity of around 80 times that of vacuum in room temperatures[1], and a relative permeability of 0.999992. There is a point to be made for pure metals, which are said to have "infinite" permittivity. In reality however, it is much more complicated and will therefore not be discussed over the scope of this thesis. One could also choose these values arbitrarily, if they desire to test purely theoretical applications. The biggest limitation for that is that these values need to satisfy the inequality $\frac{1}{\sqrt{\epsilon\mu}} \leq c$, where c is the speed of light in vacuum.

In order to have more than our environment in this implementation, the only change required is to switch from using constants to using vectors for permittivity and permeability. These vectors should be of size N, meaning the domain size, and they should have as many dimensions as the domain does. Basically put, each singular mesh will have its own values. The example below shows how this is done:

```
vector<double> permitivity;
vector<double> permeability;

int e1 = 50;
int e2 = 100;
int e3 = 25;
int e4 = 25;

int N = e1 + e2 + e3 + e4;

for (int i = 0; i < e1; i++) {
        permitivity.push_back(e1Permittivity);
        permeability.push_back(e1Permeability);
}

for (int i = 0; i < e2; i++) {
        permitivity.push_back(e2Permittivity);
        permeability.push_back(e2Permeability);
}

for (int i = 0; i < e3; i++) {
        permitivity.push_back(e3Permittivity);
        permeability.push_back(e3Permeability);
}

for (int i = 0; i < e4; i++) {
        permitivity.push_back(e4Permittivity);
```

```
        permeability.push_back(e4Permeability);
}
```

This is a simple example for the one-dimensional scenario. *e1,e2,e3,e4* are the environments and their respective sizes. For multiple dimensions, these would need to be split for each axis, i.e. *e1x,e1y,e1z*, etc. The loops would then need to change to:

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

As one might imagine, this implementation would put a higher strain on the hardware, especially in three-dimensional cases.

A.5 Using non uniform meshes

Much in the same way that the different environments were implemented, one can also choose to implement variable mesh sizes. This is a helpful way to reduce computational requirements in cases where one knows the points where a fine mesh is not necessary. If one desires to implement the most basic nonuniform mesh size, then similar to the section above they would have to start by turning the variables into vectors:

```
vector<double> deltaZ;
```

What makes it slightly more complicated is the fact that not only do the meshes rely on the size of the domain, they also dictate the number of total steps. If one doubles the step size, then the number of total steps needed to pass through the domain halves. It might seem daunting at first, but by taking it step by step it is possible to implement it in a simple way.

First, we need to make sure that the sum of all the steps is equal to the size of the domain. Basically:

$$L = \sum_{1}^{i} \Delta z_i \cdot numSteps_i$$

For ease of this implementation, this example will leave the size of the domain L to be dynamically set at the end, much in the same way the number of steps N was set in the previous section. That means that *deltaZ* and the number of total steps *numSteps* for each section can be set freely.

```
int numSteps1 = 90;
int numSteps2 = 10;
int numSteps3 = 25;
int numSteps4 = 75;

double L = deltaZ1 * numSteps1 + deltaZ2 * numSteps2 + deltaZ3 * numSteps3
↪  + deltaZ4 * numSteps4;
int N = numSteps1 + numSteps2 + numSteps3 + numSteps4;

for (int i = 0; i < numSteps1; i++) {
        deltaZ.push_back(deltaZ1);
}

for (int i = 0; i < numSteps2; i++) {
        deltaZ.push_back(deltaZ2);
}

for (int i = 0; i < numSteps3; i++) {
        deltaZ.push_back(deltaZ3);
}

for (int i = 0; i < numSteps4; i++) {
        deltaZ.push_back(deltaZ4);
```

```
}
```

If using the Courant–Friedrichs–Lewy condition to update the time step, then that would need to be amended in the above code as well:

```
// the rest of the variables remains the same

vector<double> deltaT;

for (int i = 0; i < numSteps1; i++) {
        deltaZ.push_back(deltaZ1);
        deltaT.push_back(deltaZ1 * sqrt(permitivity*permeability);
}

for (int i = 0; i < numSteps2; i++) {
        deltaZ.push_back(deltaZ2);
        deltaT.push_back(deltaZ2 * sqrt(permitivity*permeability);
}

for (int i = 0; i < numSteps3; i++) {
        deltaZ.push_back(deltaZ3);
        deltaT.push_back(deltaZ3 * sqrt(permitivity*permeability);
}

for (int i = 0; i < numSteps4; i++) {
        deltaZ.push_back(deltaZ4);
        deltaT.push_back(deltaZ4 * sqrt(permitivity*permeability);
}
```

However, for the time step only one value can be used throughout the whole simulation. As such, the most ideal way to handle this is to pick the smallest time step value in the vector *deltaT*, calling it `deltaTmin`. This can either be implemented in the loops above, which is recommended in case time complexity is prioritized and removes the need for a *deltaT* vector, or in its own loop, possibly contained in a function, for coding clarity (helpful in case the vector *deltaT* is used in other parts of the code).

```
double deltaTmin;


for (unsigned i = 0; i < deltaT.size(); i++) {
        if (deltaT[i] < deltaTmin) {
                deltaTmin = deltaT[i];
        }
}
```

Then, the update loop would become:

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

While this is the simplest way to achieve this goal, it is definitely not the best. Furthermore, such an implementation may be useless in a moving field, as the mesh is static while the wave itself is moving throughout the domain. However, this can be used as a basis to implement an adaptive mesh. Better yet, one can choose a library that already implements this in a more efficient way, e.g. libMesh[16]. Should it be desirable to have both the multi-environment domain and a nonuniform or adaptive mesh, care should be taken that the number of steps is the same for both.

A.6 General Improvements

Due to the decision to not use OS specific tools or external C++ libraries, the implementation is far from being at top efficiency. Below, certain possible improvements will be discussed. These will refer to the program itself, therefore system tweaks and hardware changes, whether internal or external, will not be considered.

A.6.1 Performance Improvements

The biggest bottleneck for such implementations will always be the amount of computations that can be done at a time. While different machines will have different capabilities, it is imperative to strive for an implementation that is as efficient as possible. This is done by basically attempting to reduce the total number of computations.

A helpful way to quantify this is the so called *Big O* notation, a mathematical notation used to describe the behavior of an algorithm when the number of arguments goes toward infinity. In Computer Science, it is used as a classification of algorithms based on their run time or spatial requirements[21]. The current implementation is $O(n^d)$, where d is the number of dimensions. This can also be read as: "the program will finish in polynomial or algebraic time".

Unfortunately, the FDTD algorithm itself basically requires n^3 operations for each vector component. Therefore, one improvement would be to use a coarser mesh, thereby having less computations to make. However, it can also be upgraded by removing the need to have three nested *for* loops. The best way to do so would be to use a library for matrix operations, which might improve the performance of each operation. Alternatively, another possible way to improve the algorithm is to add support for symmetric cases, though this can be applied to matrices as well.

A.6.2 Improvements for Symmetric Cases

When a simulation is symmetric, it means that both the geometry of the domain and the wave itself is symmetrical. This removes the need to compute the whole domain, choosing instead to split it into equal parts based on the symmetry lines. This logic can be applied to both the wave itself, and the domain as a whole.

For ease of demonstration, the one-dimensional scenario will be used again as an example. A single wave is symmetrical down the middle of it, as that is where it peaks thanks to the type of excitation that was applied. As such, a smarter algorithm could use that fact to its advantage by skipping the next values where the wave would go back to zero. Using the following code snippet as an example:

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


E[99] = exp(-(beta * pow((t - gamma), 2)));  //Gaussian excitation, alpha =
 ↪  1, Teps = 50*deltaT, eps = 1e-3, t = i * deltaT
```

It is known that this wave will travel forwards, be reflected from the boundary without any (intentional) loss in energy with negative values, then be reflected back once it reaches the beginning of the domain. The duration of this is $2N$ steps. This information can be used to cut the computation time in half, as one would only need to calculate one half of the domain, while the other can be mirrored.

With the Gaussian pulse, symmetry can also be applied to the two and three-dimensional implementations as well, so long as the excitation is applied at the center of the domain. Since this pulse will be circular, it will be propagating symmetrically towards the edges of the domain. Therefore, the domain can be split not only into equal quadrants but also into finer equal parts, since a circular simulation can have infinite symmetry lines so long as they go directly through the middle. Symmetry can also be combined with matrices, reducing run time even further.

All in all, that is the most one can hope to achieve in terms of reducing computation time by improving the algorithm. So far, the program used the CPU to complete its calculations. While that is acceptable, it pales in comparison to the computational speed that is possible to achieve by using the GPU instead, which was created for the sole purpose of fast parallel calculations. For C++ and NVIDIA graphic cards, this can be done by using CUDA.

A.6.3 Using CUDA

CUDA is a helpful application that supports multiple operating systems. It gives users the ability to use the GPU for C++ programs instead of the CPU. The benefit to this is that a GPU will have far more cores than a CPU ever will, allowing for thousands of calculations to be able to run in parallel. Even the most high end processors can have a maximum of around eight cores, while even the more dated GPUs can reach a hundred or more.

The reason is that while CPUs are meant to handle a large variety of tasks with high precision, they are limited in their parallel execution capabilities, which is where the GPU is needed. On the other hand, graphic cards could never fully replace processors, as though they may be able to perform the same operations multiple times, they lack the flexibility required to run large programs that the CPU has.

CUDA has an easy installation and many tutorials on their main website, as well as an active community. However, as it requires an NVIDIA GPU, an alternative to it would be QUDA[18] or OpenCL[15], which can be used regardless of the graphic card.

A.7 Full Code Files

Below are the full code files used in this project.

A.7.1 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)));  //Gaussian
                ↪  excitation, alpha = 1, Teps = 50*deltaT, eps = 1e-3, t
                ↪  = i * deltaT

                // 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]);
                }



                // 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.7.2 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>

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)  * (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, const vector<vector<double>> &Ex,
 ↪  const 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, const 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 < Hz[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;

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);
```

```
        }



}
```

A.7.3 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>


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, const
↪  vector<vector<vector<double>>> &Vx, const
↪  vector<vector<vector<double>>> &Vy, const
↪  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();
}

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)));

                // 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);

    }


}
```

Bibliography

[1] Donald G Archer and Peiming Wang. "The dielectric constant of water and Debye-Hückel limiting law slopes". In: *Journal of physical and chemical reference data* 19.2 (1990), pp. 371–411.

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

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

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

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

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

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

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

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

[10] George Doublys. *KS3 Physics: 4. Electromagnetism*. 2020 (accessed December 21, 2020). URL: https://medium.com/@ghduoblys/ks3-physics-4-electromagnetism-c8fb668fba93.

[11] Julian Dow. "A Unified Approach to the Finite Element Method and Error Analysis Procedures". In: 1st Edition (1998), p. xiv.

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

[13] 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.

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

[15] The Khronos® Group Inc. URL: https://www.khronos.org/opencl/.

[16] Benjamin S Kirk et al. "libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations". In: *Engineering with Computers* 22.3-4 (2006), pp. 237–254.

[17] 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.

[18] lattice. URL: http://lattice.github.io/quda/.

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

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

[21] Austin Mohr. "Quantum computing in complexity theory and theory of computation". In: *Carbondale, IL* (2014).

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

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

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

[25] Christopher Roy. "Review of discretization error estimators in scientific computing". In: *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*. 2010, p. 126.

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

[27] GCC Team. URL: https://gcc.gnu.org/.

[28] The GIMP Team. URL: https://www.gimp.org/.

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

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

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

[32] Kane Yee. "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media". In: *IEEE Transactions on antennas and propagation* 14.3 (1966), pp. 302–307.

[33] Benito van der Zander et al. URL: https://www.texstudio.org/.

[34] Zygomatic. URL: https://www.diagrameditor.com/.