Development of an Electromagnetic Simulation

Hale Barber

April 2024

1 Abstract

This paper covers the details of the creation of a numerical simulation of Maxwell's equations. The differential form of Maxwell's equations is used to formulate an algorithm to propagate electric and magnetic fields from any given starting point. In addition, the details of this algorithm's application using Rust for simulation and Python for visualization are covered. Examples of problems encountered with this simulation and some solutions identified using electromagnetic laws are provided. Finally, the program's usage is discussed, and the appendices provide a full user guide and the program's source code.

2 Simulation Design

2.1 Theoretical Basis

2.1.1 Mathematical Foundation

We will start by determining the evolution of the system from a given state, using the differential form of Maxwell's Equations.

$$\nabla \cdot \boldsymbol{E} = \frac{\rho}{\epsilon_0}$$

$$\nabla \cdot \boldsymbol{B} = 0$$

$$\nabla \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t}$$

$$\nabla \times \boldsymbol{B} = \mu_0(\boldsymbol{J} + \epsilon_0 \frac{\partial \boldsymbol{E}}{\partial t})$$

Gauss's Law does not involve dynamics, so it can be ignored for now. We will simply solve for the differentials of the fields, in terms of the time differential.

$$\partial \boldsymbol{B} = -\partial t(\nabla \times \boldsymbol{E})$$

$$\partial \boldsymbol{E} = \partial t(\frac{\nabla \times \boldsymbol{B}}{\mu_0 \epsilon_0} - \frac{\boldsymbol{J}}{\epsilon_0})$$

The boundary conditions are important to understand to properly simulate the fields. For the boundary of $t = 0$ we simply take in the value of the field at each position as our input. For the physical boundaries (i.e. $x = x_{max}$) we can approximate the derivative by assuming the exterior points have the same value as our boundary point. We will call this method using "fit" boundary conditions. We can also simply assume the points outside the boundaries have field values of 0. We will call using this "clip" boundary conditions.

2.1.2 Numerical Formulation

Now, we must turn these differentials and derivatives into discrete steps. A field value will be stored for every point in a lattice. For each point in the lattice, we can apply the above equations to calculate a discrete change in fields using a discrete change in time, a value for the curl of the field, and current density (provided by the simulation conditions). To calculate the curl of the field, we must consider how we calculate true, differential curl:

$$\nabla \times \boldsymbol{A} = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} \times \begin{bmatrix} A_x \\ A_y \\ A_z \end{bmatrix} = \hat{x}(\frac{\partial A_z}{\partial y} - \frac{\partial A_y}{\partial z}) + \hat{y}(\frac{\partial A_x}{\partial z} - \frac{\partial A_z}{\partial x}) + \hat{z}(\frac{\partial A_y}{\partial x} - \frac{\partial A_x}{\partial y})$$

If we can find the derivatives of a field in each spatial direction, we can find its curl, and thus how it should evolve over time. It is simple to find the discrete derivative in a given spatial direction; we can take the difference in field over the difference in that spatial dimension.

$$\frac{\partial \boldsymbol{A}}{\partial x} = \frac{\Delta \boldsymbol{A}}{\Delta x}$$

In the simulation this will be calculated by taking the difference between lattice points on either side of the point in question.

So, the program will, for each point in a lattice, calculate the spatial derivatives of each field using the points around it, then use these to calculate each field's curl. It will use the curl along with our given time differential to calculate the change in each field, which it will apply to find next time step's field values.

2.2 Program design

2.2.1 Simulation structure

For the main simulation I chose the Rust language. This was because Rust is a fast language (which I hadn't used before, and wanted to learn). In addition, I found a package that would let me use the GPU

Byte Value	Meaning
0	Same vector
1	Switch E/B vector
2	Next item
3	Next row
4	Next plane
5	Next frame

Table 1: Deliminator Byte Meanings

to perform the simulation more efficiently, but it did not work. I originally planned to perform the display in Rust as well, but the display package did not have the functionality I needed. Because of this, I decided to pipe the data to a Python program for display. I used a custom format for the data when it was in the pipe. The format consists of five bytes packets, where the first four bytes represent a floating point number, and the fifth byte is metadata, showing structure. The deliminator bytes follow the pattern shown in the above table (Tab. 1).

2.2.2 Output format

The Python display code uses a library called plotly, which is JavaScript-based, and displays in a browser during normal operation. It can also output to an HTML file.

2.2.3 Input format

For the the input into the program, I originally hard-coded the initial conditions, but later defined a JSON format to describe all user input to and starting conditions for a simulation. I chose to call these JSON files "manifests". They describes two things: constants and objects. The constant section defines the permeability and permittivity of the space, the size of the time step, the number of time steps to take, how many time steps to simulate for every one displayed, how many cells to simulate per each one displayed (per direction), and what strategy to use for the physical boundary conditions. The objects section is a list of objects. These objects can define starting fields within a point or plane, and define an antenna, with a given frequency and strength. Together, these describe most of the input conditions a user may want to specify. Unfortunately, the physical size of the simulation is not changeable, as the static typing of the array types used requires a constant, and thus hard-coded array size. Because of this, the simulation size is locked at a 30 by 30 by 30 grid in a one meter cube.

3 Challenges in simulation

3.1 Display speed

Generally, the process of simulation is much quicker than the process of display. This is because the simulation is in a fast language (Rust) and is highly optimized, whereas the display relies on the data being piped to a Python program using a JavaScript rendering library. Viewing frame rates sink if high spacial and temporal resolution is used. To combat this, I added the option to cull a portion of points in time or space before they are sent to the display program.

3.2 Time step resolution

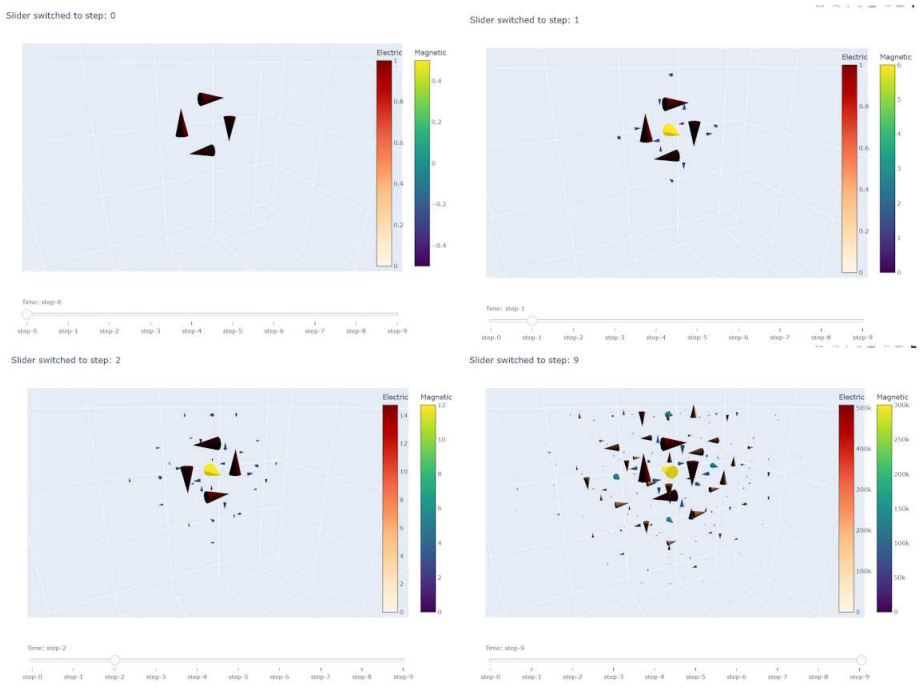

Figure 1: Simulation with too high of a time step. ($\mu_0 = 1, \epsilon_0 = 1, \Delta t = 0.1$)

High resolution is important for ensuring certain artifacts are not present. Let us take an example where a small loop of electric field is used as our starting point (Fig. 1). We know from Faraday's law, that such a loop means there must be a negative change in magnetic flux within the loop. Indeed, the simulation shows this, as after the first time step a magnetic field appears in the center of the loop, opposing the right hand normal of the loop. Notice its magnitude

is high, even for our simplified constants ($\mu_0 = 1, \epsilon_0 = 1$). Next time step, these loops in the magnetic field counteract the original electric field, as per Lenz's law. However, they overshoot, with the resulting field being in the opposite direction, at 17 times the magnitude. This overshoot builds up and makes the fields in any simulation oscillate wildly and grow to infinity.

3.3 Grid Artifacts

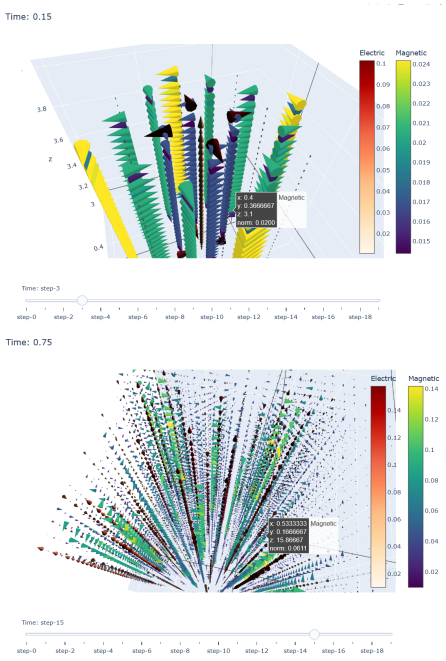

Figure 2: Antenna simulation showing grid artifacts at small and large scale. ($\mu_0 = 1, \epsilon_0 = 1, \Delta t = 0.005$)

The grid nature of the simulation causes some artifacts. For example loops of electric and magnetic field are typically squares, and propagate in a square grid. Perhaps a fine enough grid could stop this, but this would likely only be possible if the program was moved to the GPU and/or extremely optimized. Grid sizes could be significantly improved by using parallelism even without the GPU, because the new values for each cell are independent of each other. In addition, artifacts seem to persist at a larger scale (Fig. 2), so even an extremely fine grid might not solve the problem. However, a finer grid would also involve a larger diameter antenna, and maybe these artifacts would disappear with an antenna of a scale larger than that

of the grid.

3.4 Edge Artifacts

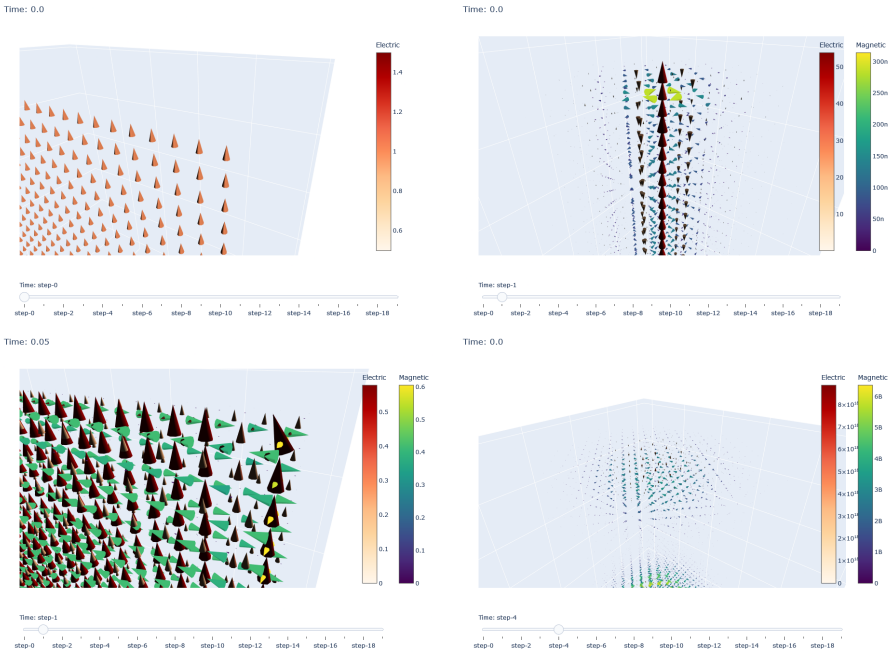

Figure 3: Errors due to edges. Left: minor perturbations, electric plane. Right: complete simulation failure, realistic antenna, high time step. Both: Edges set to "clip".

The boundary conditions can also cause artifacts, ranging from minor to extreme. A "fit" boundary condition can cause some feedback effects that cause perturbations throughout the field. In addition, low time step can also lead to extreme edge artifacts (Fig. 3, right). These perturbations typically do not cause many problems, but they can sometimes mask more subtle effects.

3.5 Divergence Drift

There is no check on the simulation to ensure it remains in line with Gauss's law. This law states that, in a vacuum, the divergence in the electric and magnetic fields should both be zero at every point. The simulation relies on the user to give an input that is in line with Gauss's law, and does not perform any kind of calibration to bring the simulation back into line if it drifts from Gauss's law. We can

track this drift by adding up the absolute value of the divergence for every point in both the electric and magnetic fields. The divergence can be numerically calculated using the same discrete derivatives as the curl. In a simulation ($\mu_0 = 1, \epsilon_0 = 1, \Delta t = 0.0025, steps = 4000$) the divergence was found to start at zero and exponentially climb, staying in a reasonable range for a long time, before eventually becoming very large. Around step 900, the simulation had significant field values reach the corners of the simulation area. After this, the field values grew throughout the whole simulation without bound. The simulation divergence appeared to follow an exponential. From step 900 to step 1000, the divergence grew by 0.202 (105%), and from step 700 to 800, the divergence grew by .049 (103%). This implies that the divergence approximately doubles every 100 steps. However, this breaks down in the early stages. From step 10 to 110, the divergence grew by 4065%.

Perhaps this divergence drift could be combated by using some kind of algorithm to rebalance the fields to maintain a zero divergence every so often, but such an algorithm is beyond the scope of this paper.

4 Applications and Conclusion

In its current form, the simulation is mainly useful as a teaching tool. It could be used to demonstrate electromagnetic waves, and the general propagation of electromagnetism. In this role, it is excellent at building an intuition for electromagnetic phenomena in vacuum. A more fully featured application, including a full UI, better tools for adding objects, more types of more complicated objects, cleaner configuration settings for those objects, and interpolation for better viewing of the output could be used for advanced circuit simulation, where an idealized circuit model would not work. Such a simulation could also be used for modeling antenna or light interactions. To reasonably accomplish this, the simulation speed, and especially the visualization speed would need to be improved. This could be done in a number of ways, particularly parallelizing simulation or moving it to the GPU. In addition, I could use a more efficient graphics engine built into the simulation program, instead of using two programs and piping the data between. Such improvements would allow a much greater spatial and temporal resolution. Enough of these optimizations and features could eventually bring this simulation to the level of enterprise software, for use in the advanced electronics industry.

Appendix A: Source Code

The source code can be found at *github.com/Hydro111/maximillion*, in addition to a compiled release. Note that the compiled release still requires Python 3, and the Python package plotly. Python 3 can be installed at *python.org/downloads*. To install plotly, run the command `pip install plotly` in your command line of choice after installing Python.

Appendix B: Usage Guide

B.1 Tutorial

The release on GitHub contains the following things:

- Simulation code compiled for Windows x_86 64-bit (`simulation.exe`)
- Display code (`display.py`)
- Example manifests (`manifests.zip`)
- Scripts to run the simulation and display (`simulate.bat`, `simulatestore.bat`)

To use the simulation, you will need all of these (excluding the manifests) in one directory. Then, you will need to open a terminal (`Win + R` , type in cmd and hit OK) and use it to run `simulate.bat` or `simulatestore.bat` (depending on whether you want to write the simulation display to HTML). It will prompt you for the manifest. Type in the path to the JSON manifest you want to simulate. The simulation will eventually open in your default browser.

B.2 Examples

B.2.1 Basic simulation

Using...

```
C:.
|---release
|    |---simulate.exe
|    |---display.py
|    |---simulate.bat
|    |---simulatestore.bat
|
|---manifests
     |---electric_plane.json
```

Running...

```
C:\>.\release\simulate.bat
Manifest filename? .\manifests\electric_plane.json
```

... Produces a simulation of an electric plane.

B.2.2 Basic simulation using piped input

Using...

```
C:.
|---release
|     |---simulate.exe
|     |---display.py
|     |---simulate.bat
|     |---simulatestore.bat
|
|---manifests
      |---electric_plane.json
```

Running...

```
C:\>echo .\manifests\electric_plane.json| .\release\simulate.bat
```

... Produces a simulation of an electric plane.

B.2.3 Basic simulation creating HTML

Using...

```
C:.
|---release
|     |---simulate.exe
|     |---display.py
|     |---simulate.bat
|     |---simulatestore.bat
|
|---manifests
      |---electric_plane.json
```

Running...

```
C:\>echo .\manifests\electric_plane.json| .\release\simulatestore.bat
```

... Produces a simulation of an electric plane and creates an `out.html` file with the simulation stored in it.