Finite-Difference Time-Domain Simulation in 1D using Python

Yusuf Mahtab

January 24, 2020

Introduction to the theory

Before we begin, let us address the Maxwell equations in light of this task. We first write the vectors for the electric field and magnetic field strength where there are no sources, $\vec{E}$ and $\vec{H}$, respectively, in Cartesian form

$$\begin{aligned} \vec{E} &= (E_x, E_y, E_z) \\ \vec{H} &= (H_x, H_y, H_z) \end{aligned} \tag{1}$$

Then $\nabla \wedge \vec{E}$ can expanded as follows:

$$\begin{aligned} \nabla \wedge \vec{E} &= \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ E_x & E_y & E_z \end{vmatrix} \\ &= \begin{pmatrix} \frac{\partial E_z}{\partial y} - \frac{\partial E_y}{\partial z} \\ \frac{\partial E_x}{\partial z} - \frac{\partial E_z}{\partial x} \\ \frac{\partial E_y}{\partial x} - \frac{\partial E_x}{\partial y} \end{pmatrix} \equiv -\mu \begin{pmatrix} \frac{\partial H_x}{\partial t} \\ \frac{\partial H_y}{\partial t} \\ \frac{\partial H_z}{\partial t} \end{pmatrix} \end{aligned} \tag{2}$$

A similar approach to (2) gives

$$\nabla \wedge \vec{H} = \begin{pmatrix} \frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z} \\ \frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x} \\ \frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} \end{pmatrix} \equiv \epsilon \begin{pmatrix} \frac{\partial E_x}{\partial t} \\ \frac{\partial E_y}{\partial t} \\ \frac{\partial E_z}{\partial t} \end{pmatrix} \tag{3}$$

We are told that $\vec{E}$ and $\vec{H}$ only vary in the z-direction. This means we can set all derivatives with respect to x and y ($\frac{\partial}{\partial x}$ and $\frac{\partial}{\partial y}$) to 0. The above expansions then simplify to the following system of six equations

$$\begin{aligned} \frac{\partial E_x}{\partial t} &= -\frac{1}{\epsilon}\frac{\partial H_y}{\partial z} \qquad \frac{\partial E_y}{\partial t} = \frac{1}{\epsilon}\frac{\partial H_x}{\partial z} \qquad \frac{\partial E_z}{\partial t} = 0 \\ \frac{\partial H_x}{\partial t} &= \frac{1}{\mu}\frac{\partial E_y}{\partial z} \qquad \frac{\partial H_y}{\partial t} = -\frac{1}{\mu}\frac{\partial E_x}{\partial z} \qquad \frac{\partial H_z}{\partial t} = 0 \end{aligned} \tag{4}$$

Introducing the Yee algorithm

The Yee algorithm, also known as the finite-difference time-domain method (FDTD), was developed by Kane S. Yee [1]. The algorithm gives approximate solutions to the Maxwell equations over space and time. Specifically, it deals with Faraday's and Ampere's Laws which incorporate the curl of the electric and magnetic vector fields, as well as their spatial and temporal derivatives, to couple the fields together. When solving for the fields independently, we arrive at simple wave equations for $\vec{E}$ and $\vec{H}$, however the solutions are more complicated when the fields are coupled as they are in the Maxwell equations.

The Yee algorithm gives solutions by staggering $\vec{E}$ and $\vec{H}$ over space and time. This means that space (here we will consider z-direction only) and time are discretised using step sizes Δz and Δt and we only look at field values at points that are described by $a\Delta z$ and $b\Delta t$, where a and b are real coefficients. These points we will call *nodes*. To make things easier, we will align $\vec{E}$ with the z-axis, such that the $\vec{E}$ nodes appear in space at integer multiples of Δz and their location stays fixed over time. We then consider $\vec{H}$ to be both spatially and temporally offset from $\vec{E}$ by $\frac{1}{2}\Delta z$ and $\frac{1}{2}\Delta t$, respectively. The result is what is known as a Yee grid, a Cartesian grid where $\vec{E}$ and $\vec{H}$ are interweaved in time and space, one field appearing midway through the step of the other. A diagram has been included later to help the reader grasp the nature of the grid.

Before we continue, let us first explore the method of central finite-difference approximations to understand how the algorithm transforms the above-mentioned partial differential equations into numerical solutions.

The central finite-difference approximation

If we have some function $f(x)$, then for some finite interval h, we can expand $f(x+\frac{h}{2})$ and $f(x-\frac{h}{2})$ about x as follows

$$f\left(x+\frac{h}{2}\right) = f(x) + \frac{1}{1!}\frac{h}{2}\frac{df}{dx} + \frac{1}{2!}\left(\frac{h}{2}\right)^2\frac{d^2f}{dx^2} + \frac{1}{3!}\left(\frac{h}{2}\right)^3\frac{d^3f}{dx^3} + \dots \tag{5}$$

$$f\left(x-\frac{h}{2}\right) = f(x) - \frac{1}{1!}\frac{h}{2}\frac{df}{dx} + \frac{1}{2!}\left(\frac{h}{2}\right)^2\frac{d^2f}{dx^2} - \frac{1}{3!}\left(\frac{h}{2}\right)^3\frac{d^3f}{dx^3} + \dots \tag{6}$$

Subtracting (6) from (5), we have

$$f\left(x+\frac{h}{2}\right) - f\left(x-\frac{h}{2}\right) = h\frac{df}{dx} + \frac{1}{3}\left(\frac{h}{2}\right)^3\frac{d^3f}{dx^3} + \dots$$

We note that the terms which have not been explicitly shown are of order h^5 and higher, so we can simply use the notation $O(h^5)$ to collate these. Rearranging for the derivative of f, the above becomes

$$\begin{aligned}\frac{df}{dx} &= \frac{f\left(x+\frac{h}{2}\right) - f\left(x-\frac{h}{2}\right)}{h} - \frac{1}{3}\left(\frac{1}{2}\right)^3 h^2\frac{d^3f}{dx^3} + O(h^4)\\ &= \frac{f\left(x+\frac{h}{2}\right) - f\left(x-\frac{h}{2}\right)}{h} + O(h^2)\end{aligned} \tag{7}$$

Taking $h \ll 1$, the $O(h^2)$ term becomes sufficiently small to be treated as negligible. Thus, the final result is

$$\frac{df}{dx} \approx \frac{f\left(x+\frac{h}{2}\right) - f\left(x-\frac{h}{2}\right)}{h} \tag{8}$$

This is known as the ***central finite-difference approximation***. Using this result, we can find the derivative of a function at some point by looking at the value of the function in the local neighbourhood of the point, without actually evaluating the derivative there at all. This, of course, comes with a level of accuracy that is determined by the second-order nature of the approximation method.

Comparison with forward and backward approximation schemes

The central approximation is favoured over the forward and backward approximations due to its higher accuracy. Below we derive the latter approximations. We begin by expanding $f(x+\Delta x)$ using Taylor's theorem, where Δx is some finite number that is not an infinitesimal

$$\begin{aligned}f(x+\Delta x) &= f(x) + \Delta x f'(x) + \frac{(\Delta x)^2}{2!}f''(x) + \dots\\ &= f(x) + \Delta x f'(x) + O(\Delta x^2)\end{aligned} \tag{9}$$

We can rearrange the series expansion to read the following

$$\frac{f(x+\Delta x)-f(x)}{\Delta x} = f'(x) + O(\Delta x)$$

Thus, we can clearly see that $\frac{f(x+\Delta x)-f(x)}{\Delta x}$ can be used as an approximation for $\frac{df}{dx}$, with an error of order Δx. This is known as the first-order approximation.

Now, consider $\Delta x > 0$, i.e. $\Delta x = h$, where h is a positive real number. Then we have the ***forward difference approximation*** which uses the interval $[x, x+h]$

$$\frac{df}{dx} \approx \frac{f(x+h)-f(x)}{h}$$

If instead we look at the interval $[x-h, x]$, then we have the ***backward difference approximation***. Here, we would consider $\Delta x < 0$, i.e. $\Delta x = -h$. Then

$$\begin{aligned} \frac{df}{dx} &\approx \frac{f(x-h)-f(x)}{-h} \\ &= \frac{f(x)-f(x-h)}{h} \end{aligned} \tag{10}$$

Both of these methods, as mentioned previously, have errors of $O(h)$. On the other hand, we can see from (7) that the central approximation has an error of $O(h^2)$, making it a second-order approximation. At sufficiently small enough h, the error for the central approximation is far better than that of the forward and backward approximations. Hence, it is generally favoured for its higher accuracy and is used in the Yee algorithm for this reason.

Using central difference approximations in the Yee algorithm

Having explored the method of central approximations in principle, we now seek to apply it to the Maxwell equations. We have already established that we have particular nodes in our Yee grid that we will be utilising, effectively the only points of interest in this entire scheme.

Consider the x component of the $\vec{E}$ field, E_x. The field has been set to be aligned with the z-axis, so we know that the E_x nodes will appear discretely on the z-axis at integer values of Δz. Similarly, the field is updated at integer values of Δt, i.e. after every time-step. This means we can locate nodes in a space-time coordinate system using discrete coordinates.

For a given node $E_x(x,t)$, we can specify its location using the indices n and i, corresponding to the spatial and temporal step of the node, respectively, so that the nodes are now represented through the following notation

$$\begin{aligned} E_x(z,t) = E_x(n\Delta z, i\Delta t) = E_x\Big|_n^i \\ H_y(z,t) = H_y(n\Delta z, i\Delta t) = H_y\Big|_n^i \end{aligned} \tag{11}$$

Using the central finite-difference approximation, the derivatives in the first equation in (4) that relate E_x and H_y can be replaced with finite-differences

$$\frac{\partial E_x}{\partial t} = -\frac{1}{\epsilon}\frac{\partial H_y}{\partial z}$$

$$\implies \frac{E_x\left(z, t+\frac{\Delta t}{2}\right) - E_x\left(z, t-\frac{\Delta t}{2}\right)}{\Delta t} = -\frac{1}{\epsilon}\frac{H_y\left(z+\frac{\Delta z}{2}, t\right) - H_y\left(z-\frac{\Delta z}{2}, t\right)}{\Delta z}$$

$$\implies \frac{E_x\Big|_n^{i+\frac{1}{2}} - E_x\Big|_n^{i-\frac{1}{2}}}{\Delta t} = -\frac{1}{\epsilon}\frac{H_y\Big|_{n+\frac{1}{2}}^{i} - H_y\Big|_{n-\frac{1}{2}}^{i}}{\Delta z}$$

We shift the spatial indices by $\frac{1}{2}$ so that E_x lies at integer indices, then solve for $E_x\Big|_n^{i+1}$ to arrive at our first update equation

$$E_x\Big|_n^{i+1} = E_x\Big|_n^{i} - \frac{\Delta t}{\epsilon \Delta z}\left[H_y\Big|_{n+\frac{1}{2}}^{i+\frac{1}{2}} - H_y\Big|_{n-\frac{1}{2}}^{i+\frac{1}{2}}\right] \tag{12}$$

Notice the H_y nodes now have half-integer indices, corresponding to the fact that $\vec{H}$ is offset from $\vec{E}$ by $\frac{\Delta z}{2}$ and $\frac{\Delta t}{2}$, so its n^{th} node appears a half-step after the n^{th} E_x node. Below is a diagram of what the grid would look like in the neighbourhood of the point $(n\Delta z, [i+\frac{1}{2}]\Delta t)$:

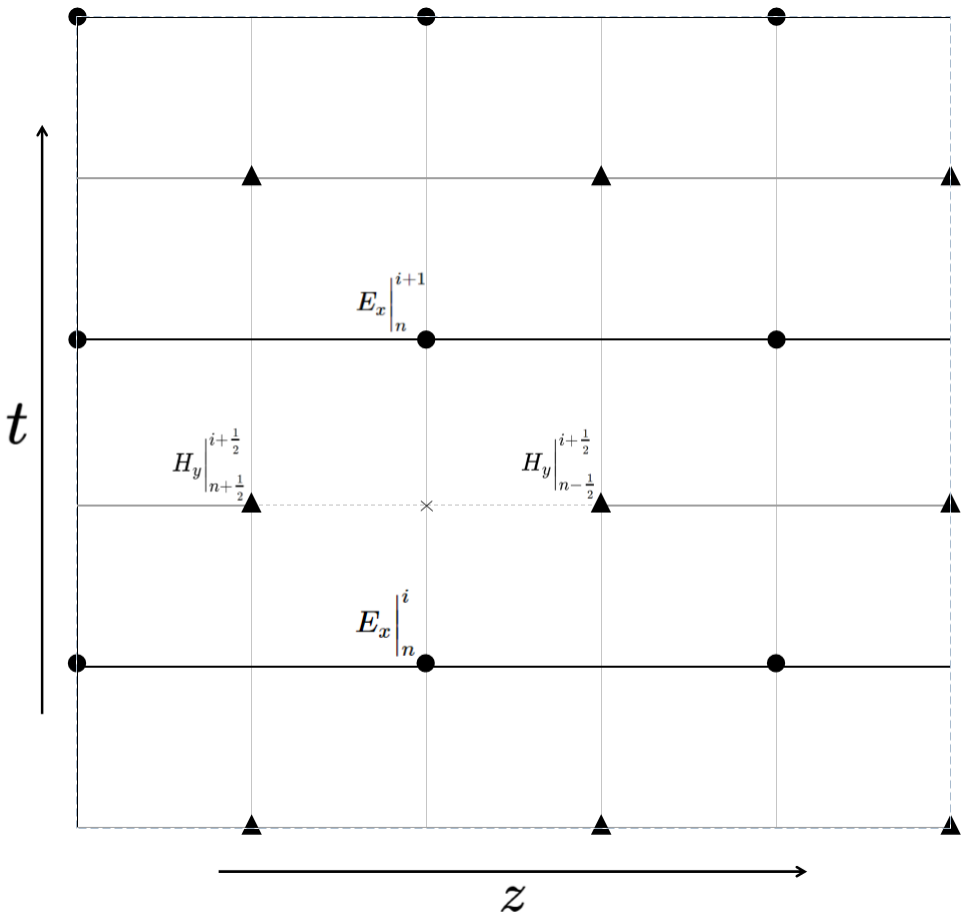

Figure 1: Yee grid showing electric and magnetic field nodes in a spacetime coordinate system. The update equations are initially formulated about the point $(n\Delta z, [i+\frac{1}{2}]\Delta t)$, indicated by the $\times$.

As we can see, our update equation allows us to use the previously stored value of the E_x node at position $n\Delta z$, as well as the neighbouring H_y values, to find the new value of the node after one timestep. Since time is a global parameter, we can apply

the update equation to all nodes across z and effectively update the electric field across the entire grid.

Following a similar process for the other differential equations in (4), excluding those involving z-components, we can derive a total of 4 update equations, split into 2 pairs of coupled difference equations

$$\begin{aligned} E_x\Big|_n^{i+1} &= E_x\Big|_n^i - \frac{\Delta t}{\epsilon \Delta z}\left[H_y\Big|_{n+\frac{1}{2}}^{i+\frac{1}{2}} - H_y\Big|_{n-\frac{1}{2}}^{i+\frac{1}{2}}\right] \\ H_y\Big|_{n+\frac{1}{2}}^{i+\frac{1}{2}} &= H_y\Big|_{n+\frac{1}{2}}^{i-\frac{1}{2}} - \frac{\Delta t}{\epsilon \Delta z}\left[E_x\Big|_{n+1}^{i} - E_x\Big|_{n}^{i}\right] \\ E_y\Big|_n^{i+1} &= E_y\Big|_n^i + \frac{\Delta t}{\epsilon \Delta z}\left[H_x\Big|_{n+\frac{1}{2}}^{i+\frac{1}{2}} - H_x\Big|_{n-\frac{1}{2}}^{i+\frac{1}{2}}\right] \\ H_x\Big|_{n+\frac{1}{2}}^{i+\frac{1}{2}} &= H_x\Big|_{n+\frac{1}{2}}^{i-\frac{1}{2}} + \frac{\Delta t}{\epsilon \Delta z}\left[E_y\Big|_{n+1}^{i} - E_y\Big|_{n}^{i}\right] \end{aligned} \tag{13}$$

These equations are the crux of the Yee algorithm. They allow updating of $\vec{E}$ halfway between the starting and ending points for a $\vec{H}$ step. Similarly, $\vec{H}$ is updated midway through an $\vec{E}$ step. The algorithm's explicit nature has allowed it to quickly become extremely popular in use across academia as it avoids the use of complicated simultaneous equations and matrix inversions [4].

Implementation

Indexing

Before we begin translating mathematics to code, we first need to set some basics about how the program will handle data. As mentioned previously, nodes for the $\vec{H}$-field have non-integer indices within the scheme. Computers only deal with integer values when indexing, so we will have to perform some mapping before we begin the main approach. We will assume that E_x nodes appear to the 'left' of H_y nodes with the same index, where 'left' represents the direction in which z is decreasing. With this arrangement, the n^{th} H_y value represents the node situated in-between the n^{th} and $(n+1)^{th}$ E_x node. In simple terms, E_x has its nodes mapped from $x \longmapsto x$ and H_y has the mapping $x + \frac{1}{2} \longmapsto x$ applied to its nodes.

Courant number and stability conditions

We note that our update equations (13) contain terms involving the step sizes Δz and Δt. In the case of $\vec{E}$, this term takes the form of $\dfrac{\Delta t}{\epsilon \Delta z}$. This can be used to derive the following result

$$\begin{aligned} \frac{\Delta t}{\epsilon \Delta z} &= \frac{\Delta t}{\epsilon_0 \epsilon_r \Delta z} \\ &= \frac{\Delta t}{\epsilon_0 \epsilon_r \Delta z}\frac{\sqrt{\epsilon_0 \mu_0}}{\sqrt{\epsilon_0 \mu_0}} = \frac{\Delta t}{\sqrt{\epsilon_0}\epsilon_r \Delta z}\frac{\sqrt{\mu_0}}{\sqrt{\epsilon_0 \mu_0}} \\ &= \frac{\Delta t}{\sqrt{\epsilon_0}\epsilon_r \Delta z} c\sqrt{\mu_0} \\ &= \frac{c\Delta t}{\epsilon_r \Delta z}\frac{\sqrt{\mu_0}}{\sqrt{\epsilon_0}} \\ &= \frac{c\Delta t}{\Delta z}\frac{Z_0}{\epsilon_r} \end{aligned} \tag{14}$$

where Z_0 is the impedance of free space and ϵ_r, the relative permittivity, is unity in free space. It is also noted that since we are simulating electromagnetic radiation, our solutions would be travelling at c in a vacuum. Therefore we expect our solution to travel $c\Delta t$ across space in one time-step. This should not be larger than the spatial step-size, Δz, we have set for our grid, as

each node should only be affecting its neighbouring nodes after each time-step. We can write this relationship in terms of the Courant number, C, which has a maximum value of unity in this method:

$$C = \frac{c\Delta t}{\Delta z} \leq 1$$

This can then be incorporated into (14) to give

$$\frac{\Delta t}{\epsilon \Delta z} \leq \frac{Z_0}{\epsilon_r}$$

As it turns out, the most stable and accurate solutions are resulted when C is maximised to unity, meaning the term above becomes a constant (since ϵ_r is unity in free space). Thus, we have

$$\frac{\Delta t}{\epsilon \Delta z} = Z_0 \approx 377 \tag{15}$$

and similarly

$$\frac{\Delta t}{\mu \Delta z} = \frac{1}{Z_0} \approx \frac{1}{377} \tag{16}$$

The above results mean that selecting a specific step size for time or space is not an issue, nor is it relevant. Instead, we decide on the number of nodes (`N_z`) we would like spaced along the z-axis and create an array using `np.arange()` to list all the indices, then do the same for t with `N_t`. In the following, `z` and `times` represent these numpy arrays. We have used here values of `N_z = 500` and `N_t = 1000`.

To implement the first update equation, we focus on the n^{th} index, e.g. `Ex[n]`. Using (13) and (15), we can program the next value for this node, bearing in mind the index mappings introduced earlier:

```
Ex[n] = Ex[n] - (Hy[n] - Hy[n-1])*Z0
Hy[n] = Hy[n] - (Ex[n+1] - Ex[n])/Z0
```

where `Z0` is the impedance of free space, equal to Z_0. Since `n` is arbitrary, this can be applied to all nodes across `z`. For each time-step, `t`, in `times`, we use a `for` loop over `z` before moving onto the next `t`. Almost immediately, we identify a problem: the update equations for `Ex` and `Hy` do not have access to the required nodes at the first and last nodes, respectively. Consider the final `Hy` node, `Hy[500]`. When we plug this into the above equation, the algorithm requests the value of `Ex[501`, which is outside of our grid and does not exist in our scheme. The node therefore does not update and always remains at a value of 0. This is commonly known as a perfect magnetic conductor, a region where no magnetic fields may pass through.

There are multiple ways to deal with this issue, all of which involve imposing some sort of boundary condition. We could use a Dirichlet boundary condition[5] where the boundary values are simply set to zero at each iteration. This would simulate a perfect electric and magnetic conductor at each end. Whenever each field reaches the corresponding conductor, it reflects off of the surface and inverts itself. The ideal scenario, however, would be to have the wave solutions move on and into the grid edge as if there were more nodes beyond the boundary. This is known as an *Absorbing Boundary Condition*(ABC) and we will explore this in the next section. There are many ways to implement ABCs, however we will consider here only the simplest 1D version, also known as a Mur boundary.

Absorbing Boundary Conditions (Mur boundary)

The ABC is implemented by considering what happens to waves at the boundaries. We know that when the E_x wave reaches the last node on the far right of the grid, it encounters a perfect magnetic conductor. This doesn't affect the field immediately but it does cause H_y to be reflected and inverted. In a similar fashion, the E_x field is not properly handled at the left side of the grid and itself is reflected and inverted. What we need to do, is ensure that the fields do not get returned when they meet the conducting layers. To do this, we quite literally make the nodes lying on the conductors absorb any field value they encounter. This is equivalent to setting the first E_x node equal to the second node and setting the final H_y node equal to the penultimate node of that field. In coding terms:

```
Ex[0] = Ex[1]
Hy[-1] = Hy[-2]
```

At this point, let us take a look at our updating process. Initially, we chose to use a `for` loop over `times` and then a nested loop that cycles through each `n` in `z`. However, this is a waste of computational time as we are using NumPy arrays, which offer far more efficiency when dealing with vector objects and operations. We note that at any given `n`, the spacial relationship between `Ex[n]` and its 'neighbouring' `Hy` nodes is consistent across space. Therefore we can remove the need for a `for` loop over `z` by instead splicing and working with the entire array, bar the nodes involved in the ABCs. Note we have switched to using a shortcut notation for adding/subtracting values

```
Ex[1:] +=  - (Hy[1:] - Hy[:-1])*Z0
Hy[:-1] += - (Ex[1:] - Ex[:-1])/Z0
```

Source functions

When it comes to actually using the program, we need to insert some kind of non-zero source into the grid for the algorithm to work with and there are numerous ways to go about this. Perhaps the easiest way is to hardwire an initial non-zero state across `z` and allow the system to evolve over time. Alternatively, a node could be selected, `Ex[100]` for example (note we only insert sources into `Ex` and `Ey`, as is standard across the literature). Here, the source would be a function of time. Popular functions are Gaussian sources such as

$$E_x[100] = exp\left[-\left(\tfrac{t-10}{10}\right)^2 \right]$$

The issue with hard sources is that any reflected fields cannot pass through the source node. The node is repeatedly set to the source function only. What we really need is a grid that continues running the Yee method as normal, but has the source function being fed into the source node as we iterate the algorithm, i.e. a superposition of the standard method and the source function. This is known as a soft source, one that does not interfere with the running of the method. The only addition needed here (considering `Ex` here) would be the following line: `Ex[100] += np.exp(-((t-t0)/tau)**2)`. Now we have a Gaussian pulse at the 40th node that moves out along the z-axis and seemingly disappears off to infinity in both directions. The parameter `tau` represents the width of the curve, while `t0` represents the temporal offset so that the source node does not suddenly jump to a high value. It is recommend to have `t0` be at least equal to 6*`tau`. This gives plenty of time for the pulse to work its way up and the field grows smoothly. The section of code to be iterated over is as follows:

```
Ex[0] = Ex[1]
Hy[-1] = Hy[-2]
Ex[100] += np.exp(-((t-t0)/tau)**2)

Ex[1:] +=  - (Hy[1:] - Hy[:-1])*Z0
Hy[:-1] += - (Ex[1:] - Ex[:-1])/Z0
```

Total-field/Scattered-field boundaries (TFSF)

The last stage of improvement of boundaries we will consider concerns the source node itself. As mentioned above, the generated pulse travels in both directions. Imagine if we set the far right side of the grid to be a perfect electric and magnetic conductor, such that the fields are forced to be reflected. If we had a one-way source at `Ex[100]` that directed a pulse in the positive z-direction, it would travel all the way to the end, be reflected and then travel back and pass through the source node. This region to the right of the node is called the total-field as it contains scattered/reflected waves as well as the original source waves moving towards the conducting wall. The region to the left of the source node is known as the scattered-field, simply because it only contains waves reflected back into that area and no source waves.

TFSF boundaries are useful in many simulations where the source needs to be controlled precisely. The idea stems from the fact that this algorithm is a leapfrog algorithm, alternating between updating $\vec{E}$ and $\vec{H}$ over time. This means that when a pulse is generated at the source node, the $\vec{H}$-field is the first to feel its effects at the nodes spatially adjacent to the source node. If we could remove the effect of the pulse on the node to the left of the source node, no further effects would be transmitted in that direction. The solution is to perform the algorithm as normal for this `Hy[n-1]` node, but decrease the value of the contribution that `Ex[n]` makes by the value of the source function at that instant in time. This negates the effect of the source and allows a one way source to be implemented.

```
Ex[0] = Ex[1]
Hy[-1] = Hy[-2]

Ex[100] += np.exp(-((t-t0)/tau)**2)
Hy[99] += np.exp(-((t-t0)/tau)**2)/Z0

Ex[1:] +=  - (Hy[1:] - Hy[:-1])*Z0
Hy[:-1] += - (Ex[1:] - Ex[:-1])/Z0
```

References

[1] Kane Yee (1966). *Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media*. IEEE Transactions on Antennas and Propagation. 14 (3): 302–307. Bibcode:1966ITAP...14..302Y.

[2] Allen Taflove and Susan C. Hagness (2005). *Computational Electrodynamics: The Finite-Difference Time-Domain Method, 3rd ed.* Artech House Publishers. ISBN 978-1-58053-832-9.

[3] John B. Schneider, *Understanding the Finite-Difference Time-Domain Method*, www.eecs.wsu.edu/~schneidj/ufdtd, 2010.

[4] Doug Neubauer, *The FDTD Yee Algorithm*, https://dougneubauer.com/yeealgorithm/#headnumber11, 2018

[5] Gernot Ohner, *Two Dimensional Finite Difference Time Domain Computation of Electromagnetic Fields in Python*, http://physik.uni-graz.at/~pep/Theses/Ohner_FDTD_Bachelorarbeit~final.pdf