FDTD Documentation

Floris Laporte

June 22, 2016

Contents

1 Preface

The main goal of this document is to provide some documentation and background for my Finite Difference Time Domain (*FDTD*) python code meant to simulate the interesting dynamics of light in photorefractive crystals. That being said, I think this documentation can also serve anyone interested in learning simple concepts of the *FDTD* method.
This document will focus specifically on the propagation and interaction of light with the crystal, therefore we will often transition between the theory of photorefractive crystals and the implementation in the code respectively.
The *FDTD* method is used in plenty of commercial electromagnetic simulating software, However because of the vast timescale differences between the governing phenomena in a photorefractive crystal, I opted for an implementation of my own.
A great resource I used in writing this code was the book of Schneider [8], which is a really good introduction to the *FDTD* method.
A nice starting point on photorefractive crystals can be the Scientific American paper of Pepper [6], which tells the story of how photorefractive crystals cought the interest of research in the 80s.
The simulation of a photorefractive can roughly be divided into two parts. First there is a pure *FDTD* method that calculates the propagation of electromagnetic waves through the crystal, while secondly the diffusion of the free carriers throught the crystal is taken into account.
All the phenomena considered happen on different timescales. We have for example the time scale of the propagation of the light through the crystal ($\approx ps$), the information bitrate ($\approx ns$), the diffusion in the crystal ($\approx \mu s$) and the generation of free carriers due to incident light ($\approx ms$). We thus seperate these phenomena in seperate sub-simulations.

2 Introduction and conventions

2.1 Electromagnetism Recap

Light and in general all electromagnetic phenomena are governed by the *Maxwell Equations*:

$$\nabla \times \mathbf{H} = \epsilon_0 \epsilon \cdot \frac{d\mathbf{E}}{dt} \qquad \nabla \cdot \mathbf{H} = 0$$
$$\nabla \times \mathbf{E} = -\mu_0 \mu \cdot \frac{d\mathbf{H}}{dt} \qquad \nabla \cdot \mathbf{H} = \rho$$

with

Field	Description	Unit
$\mathbf{E}$	Electric field	V/m
$\mathbf{H}$	Magnetic field	A/m
ϵ_0	Vacuum permittivity	As/Vm
μ_0	vacuum permeability	Vs/Am
ϵ	Relative permittivity *tensor*	1
μ	Relative permeability *tensor*	1

At any moment in time, the energy density in the electromagnetic field can be written as

$$\mathcal{E} = \frac{1}{2}\epsilon\epsilon_0 E^2 + \frac{1}{2}\mu\mu_0 H^2$$

This quantity can be seen as the energy at a certain point and has units J/m^3. If one wants to quantise the energy flow in the system, the *Poynting Vector* is needed:

$$\mathcal{S} = \mathbf{E} \times \mathbf{H}$$

which describes the energy flow in a certain direction $[W/m^2]$.
The Poynting vector and the energy density have the following relationship:

$$\frac{d\mathcal{E}}{dt} = -\nabla \cdot \mathcal{S}$$

From the Poynting vector, the light intensity (or in fact *irradiance*) in a certain direction can be deduced as the time averaged component of the Poynting vector in that direction:

$$\begin{aligned} I_x &= \mathcal{S}_x = E_y H_z - E_z H_y \\ I_y &= \mathcal{S}_x = E_z H_x - E_x H_z \\ I_z &= \mathcal{S}_z = E_x H_y - E_y H_x \end{aligned}$$

or the total irradiance:

$$I_{tot} = \sqrt{I_x^2 + I_y^2 + I_z^2}$$

2.2 Grid spacing and time step

We call du and dt the grid spacing and the time step respectively. These two values are given in units of meters and seconds.
For accuracy reasons we will assume a number $N_\lambda \geq 10$ gridpoints per *smallest* wavelength in the simulation. Therefore, for every cell in the grid, we assume for the grid spacing du:

$$du = \frac{1}{N_\lambda}\frac{\lambda_0}{n_{\max}}$$

For stability reasons, we have to choose the time step of the simulations dt to be equal to

$$dt = s_c \frac{du}{c},$$

with $s_c = \mathtt{sc} \leq 1/\sqrt{2}$ given by the CFL-condition. This condition can intuitively be understood as follows: it is clear that the timestep cannot be bigger than this value, as it would allow for waves to travel faster than light over the diagonal of a cell. This way the update equations cannot follow the propagation speed of the wave and the field values will explode. For numerical accuraxy reasons, the optimal value for s_c turns out to be exaclty $1/\sqrt{2}$ [8].

2.3 Simulation parameters

To remove ϵ_0 and μ_0 from the Maxwell's parameters, we choose as simulation quantities for the fields, which will consistently be written in monospace font:

$$\mathtt{H} = \sqrt{\mu_0}\mathbf{H}$$
$$\mathtt{E} = \sqrt{\epsilon_0}\mathbf{E}$$

This brings the Maxwell equations for the simulation Fields into a nice symmetric form:

$$\nabla \times \mathtt{H} = \frac{1}{c}\epsilon \cdot \frac{d\mathtt{E}}{dt} \qquad \nabla \cdot \mathtt{H} = 0$$
$$\nabla \times \mathtt{E} = -\frac{1}{c}\mu \cdot \frac{d\mathtt{H}}{dt} \qquad \nabla \cdot \mathtt{H} = \rho$$

using the stability criterion, we can rewrite the Maxwell equations in their final form:

$$\mathtt{curl(H)} = \mathtt{eps} \cdot \frac{\mathtt{dE}}{\mathtt{sc}} \qquad \mathtt{grad(H)} = 0$$
$$\mathtt{curl(E)} = -\mathtt{mu} \cdot \frac{\mathtt{dH}}{\mathtt{sc}} \qquad \mathtt{grad(H)} = \mathtt{rho}$$

Where we defined

$$\mathtt{curl(\cdot)} \equiv \mathtt{du}\nabla\times$$
$$\mathtt{grad(\cdot)} \equiv \mathtt{du}\nabla\cdot$$
$$\mathtt{rho} \equiv \rho\mathtt{du}$$

Note that this particular choice of simulation parameters simplifies the form of the energy density:

$$\mathcal{E} = \frac{1}{2}\left(\epsilon\epsilon_0 E^2 + \mu\mu_0 H^2\right) = \frac{1}{2}\left(\epsilon\mathtt{E}^2 + \mu\mathtt{H}^2\right)$$

The expression for the Poynting vector becomes

$$\mathcal{S} = \mathbf{E} \times \mathbf{H} = c\mathtt{E} \times \mathtt{H}$$

2.4 Grid discretization

The basis for any electromagnetic simulation is Maxwell's equations. For FDTD, these differential equations of - in real life - continuous fields are solved on a discrete grid. The electromagnetic fields are thus discretized in both space and time.
The discretization scheme used is the Yee-algorithm [10], which discretizes the fields and field components at different points of the grid. In 2D this can be visualized as follows

$\ddots$	$\vdots$	$\vdots$	$\iddots$
$\cdots$	$E_z^{m,n}$	$E_y^{m,n+\frac{1}{2}}$ $H_x^{m,n+\frac{1}{2}}$	$\cdots$
$\cdots$	$E_x^{m+\frac{1}{2},n}$ $H_y^{m+\frac{1}{2},n}$	$H_z^{m+\frac{1}{2},n+\frac{1}{2}}$	$\cdots$
$\iddots$	$\vdots$	$\vdots$	$\ddots$

These cells are repeated next to each other. The orientation of the x-axis and the y-axis is a little bit unconventional, but is chosen to agree with the first index and second index of a numpy array in python. We chose to discritize the notation in such a way that

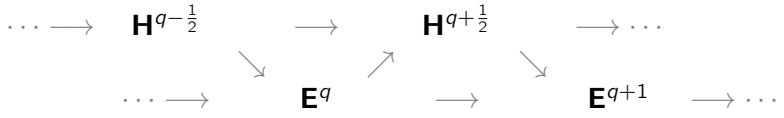
$$E_i^{m,n} = E_i(x, y) = E_i(mdu, ndu)$$
$$H_i^{m,n} = H_i(x, y) = H_i(mdu, ndu)$$

2.5 Leap frog update scheme

Apart from the staggering in space, the fields are also staggered in time. This happens in such a way that the electric field at time $t = qdt$ and the magnetic field at time $t + dt/2 = (q + \frac{1}{2})dt$ leap frog each other to compute the/ next component:

$$\mathbf{H}^{q+\frac{1}{2}} = f_H\left(\mathbf{H}^{q-\frac{1}{2}}, \mathbf{E}^q\right)$$
$$\mathbf{E}^{q+1} = f_E\left(\mathbf{E}^q, \mathbf{H}^{q+\frac{1}{2}}\right)$$

The new electric field component is a function of the old electric field component one timestep away and the magnetic field component half a timestep away. Or in a more schematic way:



Here, we again chose to discritize the notation in such a way that

$$E_i^q = E_i(t) = E_i(qdt)$$
$$H_i^q = H_i(t) = H_i(qdt)$$

2.6 Remark

In general, we do not want to keep track of the half integer values of the locations and time step indices in the grid during the simulations. We therefore define the monospace notation

$$\texttt{Ez[m,n]} = E_z^{m,n} \qquad \texttt{Ex[m,n]} = E_x^{m,n+\frac{1}{2}} \qquad \texttt{Ey[m,n]} = E_y^{m+\frac{1}{2},n}$$
$$\texttt{Hz[m,n]} = H_z^{m+\frac{1}{2},n+\frac{1}{2}} \qquad \texttt{Hx[m,n]} = H_x^{m+\frac{1}{2},n} \qquad \texttt{Hy[m,n]} = H_y^{m,n+\frac{1}{2}}$$

This means the unit cell can be rewritten as

$\ddots$	$\vdots$	$\vdots$	$\iddots$
$\cdots$	`Ez[m,n]`	`Ey[m,n]` `Hx[m,n]`	$\cdots$
$\cdots$	`Ex[m,n]` `Hy[m,n]`	`Hx[m,n]`	$\cdots$
$\iddots$	$\vdots$	$\vdots$	$\ddots$

Usually the components of a field are bundled in one big array where the first two indices indicate the location in the grid and the third index indicates the component. A certain component at a certain position can then for example be accessed as

$$\texttt{Ex[m,n]} = \texttt{E[m,n,0]} \qquad \texttt{Ey[m,n]} = \texttt{E[m,n,1]} \qquad \texttt{Ez[m,n]} = \texttt{E[m,n,2]}$$

or one component on all grid points simultaneously:

$$\texttt{Ex} = \texttt{E[...,0]} \qquad \texttt{Ey} = \texttt{E[...,1]} \qquad \texttt{Ez} = \texttt{E[...,2]}$$

Similar as for the spatial components, the half integer times will also never be referred to explicitly. We will instead write

$$\texttt{Hz[q]} = H_z^{q+\frac{1}{2}} \qquad \texttt{Hx[q]} = H_x^{q+\frac{1}{2}} \qquad \texttt{Hy[q]} = H_y^{q+\frac{1}{2}}$$

Usually, however, the time dependence will not explicitly be referenced and we choose to distinguish the current time and the next time by an accent:

$$\texttt{E}' = \texttt{E[q+1]} \qquad \texttt{E} = \texttt{E[q]}$$
$$\texttt{H}' = \texttt{H[q+1]} \qquad \texttt{H} = \texttt{H[q]}$$

3 Grid

```
In [2]: from fdtd import grid
```

3.1 Introduction

Before we can do any simulation, we need to have the space for it. This is where the Grid class comes in:

```
class Grid(object):
```

The initialization of this class can take 5 arguments:

```
    def __init__(self, shape, wl=632.8e-9, Nwl=10, eps=1., pml_thickness = 10):
        ...
```

From these arguments, the core parameters necessary for an FDTD simulation are saved into the Grid class. From the parameters `wl` $= \lambda$ and `Nwl` $= N_\lambda$, the grid spacing du is calculated. `eps` specifies the grid's isotropic relative permittivity ϵ, while the parameter `pml_thickness` specifies the thickness of the absorbing boundaries (Perfectly Matched Layer) at the edges of the grid. Thicker boundaries typically result in better absorption, but considerably slower computation time. The PML will be discussed in more detail later.
Furthermore, there are some derived quantities, such as dt, which is derived from the CFL condition. We also choose to store the inverse of the permittivities as they are more important in the update equations.
Also objects can be stored in the grid. These can be crystals with anisotropic permittivity or sources or detectors. They are stored in their respective lists.
Most importantly, the electromagnetic fields `E` and `H` are stored in the grid.

3.2 Update equations

So the grid stores all the important information, but as long as there are no possibilities to do anything with it, this would be a quite useless module. For this we have the update equations. They update the fields `E` and `H` for every time step.

3.2.1 Electric field

We start from the simplified Maxwell's curl equations for the electric field in simulation units.

$$\mathtt{dE} = \mathtt{sc} \cdot \mathtt{inv\_eps} \cdot \mathtt{curl\_H}$$

From which we derive the updates

$$\mathtt{E}' = \mathtt{E} + \mathtt{sc} \cdot \mathtt{inv\_eps} \cdot \mathtt{curl\_H}$$

Where we assumed `eps`, and thus `inv_eps` to be a scalar in the grid. `curl_H` gets calculated by using a function from the tools module:

$$\mathtt{curl\_H} = \mathtt{tls.curl(H)}$$

3.2.2 Magnetic field

Just as for the electric field update equations, we start from Maxwell's curl equations. This time the curl equation for `E`.

$$\mathtt{dH} = -\mathtt{sc} \cdot \mathtt{inv\_mu} \cdot \mathtt{curl\_E}$$

From which we derive the updates

$$\mathtt{H}' = \mathtt{H} - \mathtt{sc} \cdot \mathtt{inv\_mu} \cdot \mathtt{curl\_E},$$

similarly as for `curl_H`, `curl_E` gets calculated by using a function from the tools module:

$$\mathtt{curl\_E} = \mathtt{tls.curl(E)}$$

3.2.3 What if there are objects in the grid?

It is important to note that if there are objects such as crystals and detectors in the grid, also their respective update equations are called during the grid updates. In the case of a detector, this means that the field values at a certain location are stored in the detector object. In the case of a Crystal, the fields in the crystal are update seperately from the fields in the rest of the grid. This is done by setting `inv_eps` and `inv_mu` equal to zero at the location of the crystal during its creation and using the crystal update equations in stead.

3.3 FDTD loop

in this loop, where the time step `q` runs from `0` to `Q`, everything comes together. Every iteration, first the electric field is updated. After updating the electric field by the normal update equations, extra energy is added by calculating the source value for the electric field at the right time. After updating the electric fields, the same is done for the magnetic fields.

4 Sources

```
In [3]: from fdtd import source
```

4.1 Source Parent

Before we can start propagating the light with FDTD, we have to generate it. This is done by the source class, which has the following typical structure

```
class Source(object):
    def E(self, q):
        ...
    def H(self, q):
        ...
```

All other source classes are subclasses of this parent class. The only thing these methods (of subclassed sources) do, is calculating the value of the E-field and H-field at a certain (integer) time q at a specific location in the grid.

4.2 Hard sources and soft sources

Sources can be implemented in two ways: as a *hard* source or as a *soft* source. For a hard source, the update equation sets the fields in the grid equal to the desired source value at a certain time:

```
    def E(self, q):
        self.grid.E[self.locX,self.locY,2] = source_at_time(q)
```

For a soft source, the field at a certain time is merely added to the existing fields in the grid:

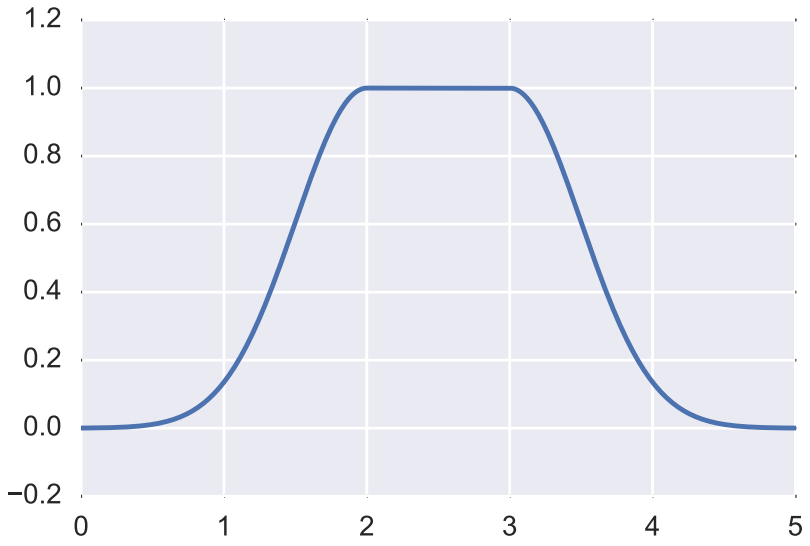
```
    def E(self, q):
        self.grid.E[self.locX,self.locY,2] += source_at_time(q)
```

Hard sources tend to be numerically more stable, however, as the position of the source fixes the field at a certain point in the grid they will also act as a reflector.
`self.locX` and `self.locY` denote the position of the source in the grid. The number 2 denotes the component of the field with number 2, since python starts counting from 0, the z-component gets updated. Since the z-component is perpendicular to the 2D grid in x and y, this is a TE mode.

4.3 Gaussian envelope

For numerical stability, the light has to be introduced smoothly into the grid. Therefore, we define an pulse function:

```
In [4]: q = linspace(0,5,1000)
        y = source.pulse(q, sigma=0.5, pulselength=5, r=4)
        plt.plot(q,y)
        plt.show()
```



Gaussian Envelope

`sigma` defines the steepness of the rising and falling edge, `pulselength` defines how long the pulse is (rising and falling edges included) and `r` defines how many `sigma`'s the rising edge takes (delay of the pulse)

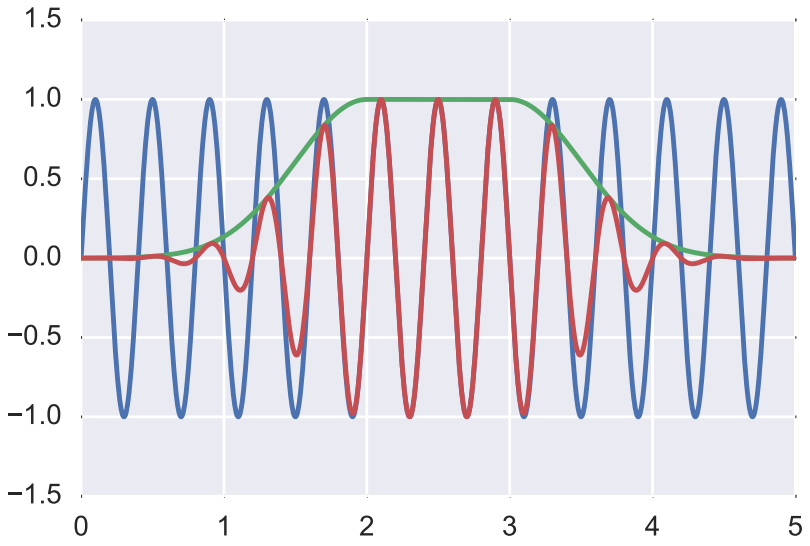
4.4 Oblique Gaussian

4.4.1 Update equations

This is the most used source in the simulations. At any timestep `q`, the core of the source generator can be written as

$$\texttt{source\_at\_time(q)} = \sin\left(\frac{2\pi \texttt{q}}{\texttt{period}}\right) \cdot \texttt{pulse(q, sigma)}$$

```
In [5]: q = linspace(0,5,1000)
        pulse = source.pulse(q, sigma=0.5, pulselength=5, r=4)
        sinusoid = sin(2*pi*q/0.4)
        plt.plot(q,sinusoid)
        plt.plot(q,pulse)
        plt.plot(q,sinusoid*pulse)
        plt.show()
```

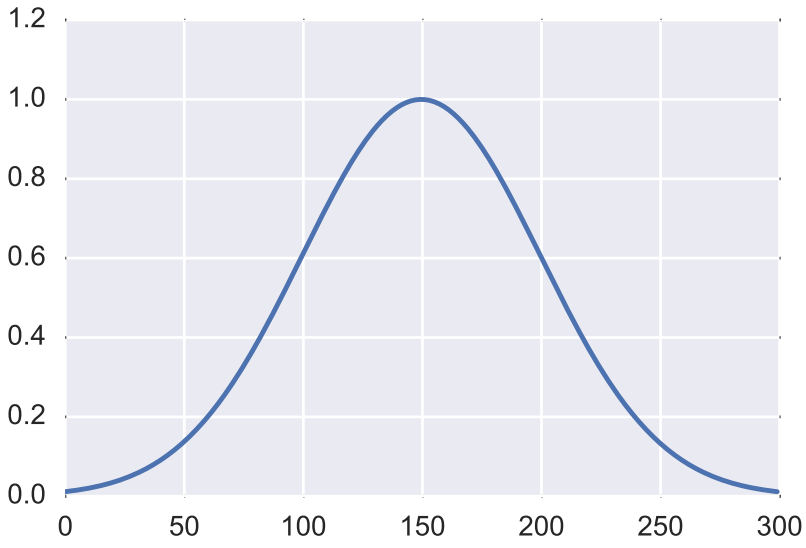


Pulse of an Oblique Gaussian source

However, someting special about this source is that it also has a gaussian profile in space:

4.4.2 Spatial profile

```
In [6]: size = 300.
        vect = arange(size)-size/2+0.5
        profile = exp(-18*vect**2/size**2)
        plt.plot(profile)
        plt.show()
```



Gaussian source profile

4.4.3 Average energy density

Average energy density of a sinusoidal source A sinusoidal source (or plane wave) is given by:

$$E = A \sin \frac{2\pi t}{T}$$

The average energy density of this source is thus

$$\begin{aligned} \langle \mathcal{E} \rangle &= \frac{1}{2} \epsilon \epsilon_0 \frac{1}{T} \int_0^T E^2 \\ &= \frac{1}{2} \epsilon \epsilon_0 \frac{A^2}{T} \int_0^T \sin^2 \frac{2\pi t}{T} \\ &= \frac{1}{4} \epsilon \epsilon_0 A^2 \end{aligned}$$

From this, we can also conclude that the average electric field amplitude of this source is equal to

$$\langle A \rangle = \frac{A}{\sqrt{2}}$$

Average energy density of a gaussian pulse A gaussian pulse is given by:

$$E = A \exp\left(-\frac{x^2}{2\sigma^2}\right)$$

Assuming the pulse goes from $-r\sigma$ to $r\sigma$ in time, with $r \geq 3$:

$$
\begin{aligned}
\langle \mathcal{E} \rangle &= \frac{1}{2}\epsilon\epsilon_0 \frac{1}{2r\sigma} \int_{-r\sigma}^{r\sigma} E^2 \\
&= \frac{1}{2}\epsilon\epsilon_0 \frac{A^2}{2r\sigma} \int_{-r\sigma}^{r\sigma} \exp\left(-\frac{x^2}{\sigma^2}\right) \\
&\approx \frac{1}{2}\epsilon\epsilon_0 \frac{A^2}{2r\sigma} \int_{-\infty}^{\infty} \exp\left(-\frac{x^2}{\sigma^2}\right) \\
&= \frac{1}{2}\epsilon\epsilon_0 \frac{A^2}{2r\sigma} \sigma\sqrt{\pi} \\
&= \frac{\sqrt{\pi}}{4r}\epsilon\epsilon_0 A^2
\end{aligned}
$$

Average energy density of a sinusoidal gaussian pulse Assuming that $T \ll r\sigma$, we can just substitue the average amplitude of the sinusoidal source into the expression for the energy density of the gaussian pulse:

$$
\langle \mathcal{E} \rangle = \frac{\sqrt{\pi}}{8r}\epsilon\epsilon_0 A^2
$$

average energy density of a sinusoidal source with gaussian edges Combining the results one last time, we can write for $t > 2r\sigma$:

$$
\begin{aligned}
\langle \mathcal{E} \rangle &= \frac{2r\sigma}{t} \cdot \left(\frac{\sqrt{\pi}}{8r}\epsilon\epsilon_0 A^2\right) + \left(1 - \frac{2r\sigma}{t}\right) \cdot \left(\frac{1}{4}\epsilon\epsilon_0 A^2\right) \\
&= \frac{1}{4}\epsilon\epsilon_0 A^2 \left(1 - \frac{2r\sigma}{t} + \frac{2r\sigma}{t}\frac{\sqrt{\pi}}{2r}\right) \\
&= \frac{1}{4}\epsilon\epsilon_0 A^2 \left(1 + \frac{2r\sigma}{t}\left(\frac{\sqrt{\pi}}{2r} - 1\right)\right)
\end{aligned}
$$

4.4.4 Total energy introduced in the system:

The total energy the source introduced into the system after a time t can be given by

$$
\begin{aligned}
\mathbb{E} &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \langle \mathcal{E} \rangle \, dxdydz \\
&= h \cdot ct \int_{-\infty}^{\infty} \langle \mathcal{E} \rangle \, dx \\
&= \frac{h \cdot ct}{4}\epsilon\epsilon_0 A_0^2 \left(1 + \frac{2r\sigma}{t}\left(\frac{\sqrt{\pi}}{2r} - 1\right)\right) \int_{-\infty}^{\infty} \exp\left(-\frac{x^2}{\left(\frac{L}{3}\right)^2}\right) dx \\
&= \frac{h \cdot ct \cdot L}{12}\epsilon\epsilon_0 A_0^2 \sqrt{\pi} \left(1 + \frac{2r\sigma}{t}\left(\frac{\sqrt{\pi}}{2r} - 1\right)\right)
\end{aligned}
$$

with $L = \texttt{size}$ is the length of the spatial source profile and $A_0 = A(x = 0)$.
This makes the average energy density in the grid with dimensions $Mdu \times Ndu \times h$ equal to

$$
\begin{aligned}
\langle \mathcal{E} \rangle &= \frac{\mathbb{E}}{M \cdot N \cdot du^2 \cdot h} \\
&= \frac{c \cdot L}{12MNdu^2}\epsilon\epsilon_0 A_0^2 \sqrt{\pi} \left(t + \left(\sqrt{\pi} - 2r\right)\sigma\right)
\end{aligned}
$$

Which becomes in simulation units:

$$\langle \mathcal{E} \rangle = \frac{c \cdot L}{12MNdu^2} \epsilon \mathrm{A}^2 \sqrt{\pi} \left(t + \left(\sqrt{\pi} - 2r \right) \sigma \right)$$

We can check wether this equation holds by creating a simple FDTD setup:

```
In [7]: # Free space 150x150 grid with no absorbing boundaries
        grd = grid.Grid(shape=(120,700), pml_thickness=0)
        M,N,du,dt,eps,mu = grd.M,grd.N,grd.du,grd.dt,grd.eps,grd.mu
        Q = 500 # Number of time steps
        src = source.Oblique(grd, center=(60,350), size=100, tangent=0,
                             period=10, pulselength=Q, sigma=50, r=4)
        L, t, sigma, r = src.size*du, src.pulselength*dt, src.sigma*dt, src.r
        A = src.amp = 1
        grd.run_fdtd(Q) # Run for Q time steps
        plt.imshow(grd.E[...,2]) # Show the z-component of the E-field
        plt.show()
```

Oblique Gaussian Pulse originating from the center.

Let's analyse this block of code. The first line imports the Grid class. We will go into the details of this class in the next section. For now, the only things we need to know is that we created a grid with dimensions $120du \times 500du$ in which we can propagate the light by using the `grd.run_fdtd(number_of_time_steps)` function. We also chose to not include absorbing boundaries (Perfectly Matched Layer or PML) by setting the `pml_thickness` parameter to zero. After creating the grid, we put a source in the center of the grid (at location $(60, 250)$) with the specified parameters. Next, we let the light propagate over a total time of $400dt$. Finally, we visualize the fields. Note that the x-axis is vertical pointing downwards and the y-axis is horizontal pointing to the right in our convention.
We can compare the theoretically predicted introduced energy of the source with the total numerical energy int the grid:

```
In [8]: estimated = c*L*eps*A**2*sqrt(pi)*(t+(sqrt(pi)-2*r)*sigma)/(12*M*N*du**2)
        numerical = 0.5*eps*(grd.E**2).sum()/(M*N)+0.5*mu*(grd.H**2).sum()/(M*N)
        print "estimated:",estimated
        print "numerical:",numerical
```

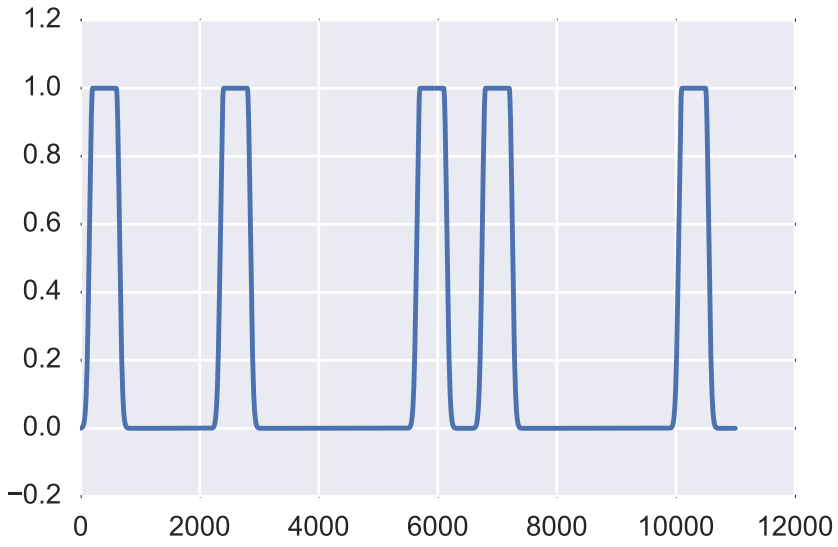
```
estimated: 0.0232170151229
numerical: 0.0277871627527
```

Of course there is some descrepancy between the estimated and real numerical values, as we made some approximations in the process; especially in assuming that $T \ll \sigma$.

4.5 Creating a bit stream

It is also possible to create a random bit stream with the source module. This is done by specifying the number of bits `Nbits` and the bitlength `bitlength` in the initialization of the source. Optionally, a random seed can be specified to have reproducable bitstreams:

```
In [9]: grd = grid.Grid(shape=(120,700), pml_thickness=0)
        src = source.Oblique(grd, center=(60,350), size=100, tangent=0,
                             period=10, pulselength=800, sigma=50, r=4,
                             bitlength=1100, Nbits=10, seed=6)
        plt.plot(src.stream)
        plt.show()
```



Ten random bits [1010011001] generated by the source

5 Perfectly Matched Layer

```
In [10]: from fdtd import pml
```

5.1 General idea

Basically a PML is a layer that absorbs incident light without having an impedance mismatch between the region of interest (and thus having no reflections between both layers).
The PML can be seen as a uniaxial material for which

$$\epsilon_r = \mu_r \equiv s = \begin{pmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & s_x & 0 \\ 0 & 0 & s_x \end{pmatrix}$$

for a pml that absorbs in the x-direction.
in the frequency domain, one writes

$$s_x = 1 + \frac{\sigma'_x}{j\omega\epsilon_0},$$

Where we made the substitution of

$$\texttt{sigma} = \sigma' = \epsilon^{-1}\sigma.$$

`sigma` is completely free to choose although for best performance one genereally takes it gradially increasing towards the edge of the grid.
The maxwell's equations for the z-component of the E-field become in this way (frequency domain):

$$\epsilon j\omega E_z = \frac{1}{s_x}\frac{dH_y}{dx} + \frac{1}{s_y}\frac{dH_x}{dy},$$

with

$$\frac{1}{s_x} = \frac{1}{1 + \frac{\sigma'_x}{j\omega\epsilon_0}} = 1 - \frac{1}{1 + \frac{j\omega\epsilon_0}{\sigma}}$$

or we can write for the full E-field:

$$\epsilon j\omega \mathbf{E} = \tilde{\nabla} \times \mathbf{H}$$

with

$$\tilde{\nabla} = \begin{pmatrix} \frac{1}{s_x}\frac{d}{dx} \\ \frac{1}{s_y}\frac{d}{dy} \\ \frac{1}{s_z}\frac{d}{dz} \end{pmatrix}$$

This becomes in the time domain:

$$\epsilon\frac{d\mathbf{E}}{dt} = \nabla \times \mathbf{H} + \Phi_E$$

with

$$\Phi_E = \begin{pmatrix} \psi_{E_x,y} \\ -\psi_{E_y,x} \\ \psi_{E_z,x} - \psi_{E_z,y} \end{pmatrix},$$

where the ψ terms are results of the integration coming from transforming the

$$\frac{1}{s} = 1 - \frac{1}{1 + \frac{j\omega\epsilon_0}{\sigma}}$$

into the time domain (generally $\frac{1}{j\omega}$ will always yield integrations)
We can iteratively integrate the ψ's by using the following update scheme:

$$\psi_{E_z,x} = b_x \psi_{E_z,x} + c_x \frac{dH_y}{dx}$$

And all other permutations of x,y and z and with $b = \exp(-\sigma \cdot dt)$ and $c = (b - 1)$

5.2 Remarks

- An extra parameter a can be added for numerical stability. (see chapter 11 of Schneider [8] for the new b and c).
- An other parameter κ can still be implemented. (see chapter 11 of Schneider [8] for the new b and c).
- Note that the pml module has only been tested for constant epsilon so far. It is thus far not allowed to put a (Crystal) object inside the pml.

6 Photorefractive crystal

```
In [12]: from fdtd import crystal
```

6.1 Introduction

```
class Crystal(object):
```

As for the grid class a few parameters are needed during initialization of the crystal:

```
    def __init__(self, grid, shape, ul, fast_update=False):
        ...
```

During the initialization, the crystal stores a reference to the grid it's in, while also storing a reference to itself by appending itself to the list of objects in the grid class. Since there is now a reference to the grid in the crystal, the field values and core parameters such as grid spacing and index can easily be accessed.
The `shape` and `ul` arguments define the shape and location of the upper left corner of the crystal in the grid respectively. Fast access to the x and y location in the grid is stored in the slice variables `locX` and `locY`. These slice parameters are also used to set the permittivity and permeability in the grid at the location of the crystal to zero. This way, the general grid update gets suppressed.
the parameter `fast_update` allows for a faster update than the standard update equations of the crystal. It reverts the update to a similar update equation as can be found in the grid class. This is of course only possible if the matrix caracter of the permittivity tensors gets neglected.
Apart from creating some parameters required for calculating the space charge field **S**, the only thing the initialization still does is setting the material parameters.

6.2 Material parameters

```
def _material_parameters(self):
    ...
```

The material parameters are all the parameters that influence the diffusion of the carriers in the crystal and the propagation of the light through the crystal.
Under the influence of light, electrons get excited from the traps in the photorefractive crystal. These electrons are now free to diffuse through the crystal until they arrive at positions of lower light intensity, where they will get trapped again. This diffusion of charges creates a remnant electric field $E_{\text{rem}} \equiv \mathtt{R}$ that in turn will have an effect on the permittivity tensor ϵ of the crystal.
In the simulations, we use a simplified model with some assumptions to keep the model as simple as possible, while at the same time trying to understand the essence of the problem:

- Firstly, no difference is made between traps and donors. We assume any unfilled trap is positively charged and any filled trap is a (neutral) donor, while in reality there may be some irreversible effects.
- Secondly, it is assumed that all traps have the same energy
- At last, we assume there is no valence band as an electron reservoir.

The dynamics of the electrons in the crystal can be reduced to the following equations from Kukhtarev [5]:

$$\begin{aligned}
\frac{dn}{dt} &= \frac{dN_D^+}{dt} + \nabla \cdot \mathbf{J} \\
\frac{dN_D^+}{dt} &= (sI + \beta)(N_D - N_D^+) - \gamma n N_D^+ \\
\mathbf{J} &= \frac{\mu k T}{e} \nabla n - \mu n \mathbf{S}
\end{aligned}$$

optionally, absorption can also be introduced:

$$I = I_0 \exp(-\alpha r)$$

Typical values for a lithium niobate crystal are for example [1, 3, 4]:

	Parameter	Value	Unit
s	Photoionization cross section	0.0025	m^2/J
β	Thermal excitation rate	1.0	s^{-1}
γ	Recombination rate	10^{-15}	m^3/s
μ	Mobility	0.0015	m^2/Vs
N_D	Donor density	$6.6 \cdot 10^{24}$	m^{-3}
N_D^+	Initial excited donor density	$3.3 \cdot 10^{24}$	m^{-3}
n	Initial free electron density	$1 \cdot 10^{17}$	m^{-3}
ϵ	Static relative permittivity	32	1
ϵ	Relative permittivity @ 1500 nm	4.9 - 4.6	1
$\mathbf{S}$	Space Charge Electric Field	$< 10^5$	V/m
α	Absorption	20	m^{-1}

	Parameter	Value	Unit
r_{ij}	Electro optic coefficients	$r_{22} = 7$ $r_{13} = 10$ $r_{33} = 32$ $r_{42} = 32$	pm/V

6.3 Absorption in the crystal

A good question we may ask ourselves is how absorption can be introduced into the grid. In general, we can assume the intensity of the light to decay exponentionally over a distance r in the material:

$$I = I_0 \exp(-\alpha r)$$

On the other hand, in frequency domain simulations, absorption is often introduced by considering a complex refractive index of the material:

$$\tilde{n} = n - i n_i,$$

which in the end also results in an exponentially decaying electric field:

$$\mathbf{E} = \mathbf{E}_0 \exp(i\omega t - nkr) \exp(-n_i k r)$$

which results for the intensity into

$$I = I_0 \exp(-2 n_i k)$$

Equating the two formulas for the intensity, we get for the absorption coefficient:

$$\alpha = \frac{4\pi n_i}{\lambda}$$

There is however one problem. We cannot simply plug in complex refractive indices in a time domain simulation. Therefore, we try something different, something unphysical: we introduce a magnetic conductivity in the maxwell curl equation for E:

$$\nabla \times \mathbf{E} = -\mu_0 \mu \cdot \frac{d\mathbf{H}}{dt} - \sigma_M \mathbf{H}$$

For the right hand side we can write in the frequency domain:

$$\begin{aligned} \mu_0 \mu \cdot \frac{d\mathbf{H}}{dt} + \sigma_M \mathbf{H} &= i\omega\mu_0 \left(\mu - i \frac{\sigma_M}{\mu_0 \omega} \right) \mathbf{H} \\ &= i\omega \tilde{\mu} \mathbf{H} \end{aligned}$$

Where we defined the complex permeability $\tilde{\mu} = \mu - i\frac{\sigma_M}{\mu_0 \omega}$. This leads to a complex refractive index:

$$\tilde{n} = n - i n_i = \sqrt{\epsilon \tilde{\mu}} = \sqrt{\epsilon\mu - i \frac{\epsilon \sigma_M}{\mu_0 \omega}}$$

Assuming $\mu = \mathtt{mu} = 1$, we get $n = \sqrt{\epsilon}$ and

$$n^2 - n_i^2 - 2inn_i = n^2 - in^2\frac{\sigma_M}{\mu_0\omega}$$

or

$$n_i = n\frac{\sigma_M}{2\mu_0\omega} = \sqrt{\epsilon}\frac{\sigma_M}{2\mu_0\omega}$$

Which gives us for the absorption α:

$$\alpha = \frac{2\pi\sqrt{\epsilon}\sigma_M}{\mu_0\omega\lambda} = \frac{\sqrt{\epsilon}\sigma_M}{\mu_0 c} = \frac{\sqrt{\epsilon}\sigma_M}{\eta_0}$$

with $\eta_0 = \sqrt{\mu_0/\epsilon_0}$ the impedance of free space.

6.4 Update equations for the magnetic field

```
def update_H(self, curl_E):
    f = 0.5 * self.sigmaM
    Hpre = self.grid.H[self.locX,self.locY]
    self.grid.H[self.locX,self.locY] = (1-f)/(1+f)*Hpre - self.grid.sc/(1+f)*curl_E
```

In the update equations for the crystal, we have to take into account the absorption. This is done by taking into account the a carefully chosen (unphysical) magnetical conductivity:

$$\begin{aligned}
\mu_0\frac{d\mathbf{H}}{dt} + \sigma_M H &= -\nabla\times\mathbf{E} \\
\sqrt{\mu_0}\frac{d\mathrm{H}}{dt}du + du\frac{\sigma_M}{\sqrt{\mu_0}}\mathrm{H} &= -du\nabla\times\frac{1}{\sqrt{\epsilon_0}}\mathrm{E} \\
\frac{d\mathrm{H}}{dt}du + du\frac{\sigma_M}{\mu_0}\mathrm{H} &= -c\cdot\mathtt{curl(E)} \\
\mathrm{dH} + \frac{\sigma_M dt}{\mu_0}\cdot\mathrm{H} &= -\mathtt{sc}\cdot\mathtt{curl(E)}
\end{aligned}$$

where we assumed $\mu = 1$. Making the substitution for the magnetic conductivity to simulation units yields

$$\mathrm{dH} + \mathtt{sigma}\cdot\mathrm{H} = -\mathtt{sc}\cdot\mathtt{curl\_E}$$

By using finite difference for `dH` and averaging `H` over the time steps `q` and `q-1`, we get

$$\begin{aligned}
&\mathrm{H}' - \mathrm{H} + \mathtt{sigma}\cdot\frac{1}{2}\left(\mathrm{H}' + \mathrm{H}\right) = -\mathtt{sc}\cdot\mathtt{curl_E} \\
\Leftrightarrow\ &\left(1 + \frac{\mathtt{sigma}}{2}\right)\mathrm{H}' = \left(1 - \frac{\mathtt{sigma}}{2}\right)\mathrm{H} - \mathtt{sc}\cdot\mathtt{curl_E} \\
\Leftrightarrow\ &\mathrm{H}' = \frac{\left(1 - \frac{\mathtt{sigma}}{2}\right)}{\left(1 + \frac{\mathtt{sigma}}{2}\right)}\mathrm{H} - \frac{\mathtt{sc}}{\left(1 + \frac{\mathtt{sigma}}{2}\right)}\cdot\mathtt{curl_E}
\end{aligned}$$

Where we assumed $\mu = 1$ throughout the crystal.

6.4.1 Keeping track of the absorption

```
da = tls.H_Hz( (self.grid.H[self.locX,self.locY] + Hpre)
              *((1-(1-f)/(1+f))*Hpre - (1-1/(1+f))*self.grid.sc*curl_E) )
self.Ab += 0.5*tls.sumc(da)
```

In the update equations for H, we can also keep track of how much absorbed energy $\mathcal{A}$ that gets dissipated. This is done by tracking the change in energy density in the field per timestep t:

$$\begin{aligned} d\mathcal{A}(q) &= \frac{d\mathcal{E}}{dt}(q)dt = \mathtt{H}(q) \cdot \frac{d\mathtt{H}}{dt}(q)dt \\ &= \mathtt{H}(q) \cdot \left(\left.\frac{d\mathtt{H}}{dt}\right|_{\sigma_M=0} - \left.\frac{d\mathtt{H}}{dt}\right|_{\sigma_M\neq 0} \right) dt \\ &= \frac{\mathtt{H[q]} + \mathtt{H[q\text{-}1]}}{2\mathtt{dt}} \cdot \left(\mathtt{H[q]}_{\sigma_M=0} - \mathtt{H[q\text{-}1]} - \mathtt{H[q]}_{\sigma_M\neq 0} + \mathtt{H[q\text{-}1]} \right) dt, \end{aligned}$$

which gives

$$d\mathcal{A} = \frac{\mathtt{H'} + \mathtt{H}}{2} \cdot \left(\mathtt{H}'_{\sigma_M=0} - \mathtt{H}'_{\sigma_M\neq 0} \right),$$

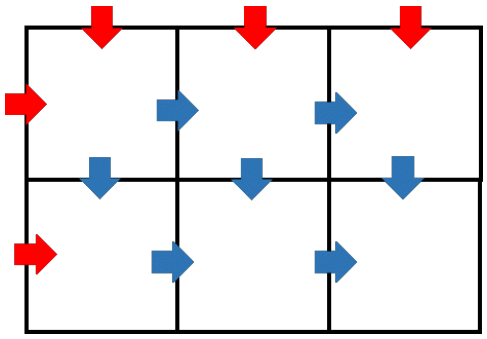
Note that the '$\cdot$' signifies an inner product in this context, and thus da is just a scalar.
The total absorption after a time t is thus:

$$\begin{aligned} \mathcal{A}(t) &= \int_0^t d\mathcal{A} \\ \Leftrightarrow \mathcal{A}[q] &= \frac{1}{2} \sum_0^q \left(\mathtt{H}' + \mathtt{H} \right) \cdot \left(\mathtt{H}'_{\sigma_M=0} - \mathtt{H}'_{\sigma_M\neq 0} \right) \end{aligned}$$

6.4.2 Implementation of the absorption

```
@property
def sigmaM(self):
    ret = self.alpha*eta0*np.sqrt(self.inv_eps)*self.grid.dt
    ret[0,:,0] = 0.
    ret[:,0,1] = 0.
    return ret
```

There is one small detail that has to be considered. Look at the figure of the electric field H in a 2×3 grid:



Magnetic Field Components in a 2×3 grid

The grid is assymetrical. This means that we should set the absorption (and thus σ_M) at the components in red to zero, to have a more symmetrical absorption.

6.5 Electrooptic effect

A redistribution of charges in the crystal gives rise to a change in refractive index (or permittivity) by means of the electro optick Pockels effect: the charge distribution gives rise to a space charge field in the crystal, which influences the inverse permittivity in a linear matter:

$$\Delta\epsilon^{-1} = \begin{pmatrix} r_{11}E_x + r_{12}E_y + r_{13}E_z & r_{61}E_x + r_{62}Ey + r_{63}Ez & r_{51}E_x + r_{52}E_y + r_{53}E_z \\ r_{61}E_x + r_{62}E_y + r_{63}E_z & r_{21}E_x + r_{22}Ey + r_{23}Ez & r_{41}E_x + r_{42}E_y + r_{43}E_z \\ r_{51}E_x + r_{52}E_y + r_{53}E_z & r_{41}E_x + r_{42}Ey + r_{43}Ez & r_{31}E_x + r_{32}E_y + r_{33}E_z \end{pmatrix}$$

where the r_{ij} pockels parameters are depending on the material's crystal symmetry. For example for a $4mm$ material like Barium tetraborate this looks like

$$r = \begin{pmatrix} 0 & 0 & r_{13} \\ 0 & 0 & r_{13} \\ 0 & 0 & r_{33} \\ 0 & r_{42} & 0 \\ r_{42} & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

While for a $3m$ material like Lithium Niobate, the tensor looks like

$$r = \begin{pmatrix} 0 & r_{12} & r_{13} \\ 0 & -r_{12} & r_{13} \\ 0 & 0 & r_{33} \\ 0 & r_{42} & 0 \\ r_{42} & 0 & 0 \\ r_{12} & 0 & 0 \end{pmatrix}$$

The update of the inverse permittivity has to be done carefully though. This is because the components of the space charge field also form a staggered grid. Therefore, we store always three versions of epsilon; one interpolated the the E_x position, one interpolated to the E_y position and one interpolated to the E_z position.
The r_{ij} parameters are usually very small: they are expressed in picometers per volt. Typical Lithium Niobate has maximal values for r_{ij} around $30pm/V$, while the best photorefractive crystals have values of around $100pm/V$. Usually values for the space charge field are limited by the breakdown voltage, which is (up to an order of magnitude) about $100kV/m$; therefore, the maximal refractive index change is limited by the value of $r_{ij}E_j$, which usually will not be higher than 10^{-5}
However, in most of the simulations, we will opt for higher values for the pockels parameters, because the crystal considered in simulation is much smaller than a realistic crystal. Therefore the effects on the refractive index should be exagerated to reach visible effects.

6.6 Update equations for the Electric field

6.6.1 Version of Werner and Cary

The electric field inside the crystal is updated in exactly the same way as it is done in the grid, however, this time, `inv_eps` will be a *different matrix* at *every grid point*.

We get:

$$\mathtt{E}' = \mathtt{E} + \mathtt{sc} \cdot \mathtt{inv\_eps} \cdot \mathtt{curl\_H},$$

where `inv_eps` is now a spatially varying tensor. The problem is now that `inv_eps` depends on the spatially varying space charge field in the crystal, which also form a staggered grid. Therefore the three tensors `inv_epsX`, `inv_epsY` and `inv_epsZ` were defined as the inverse permittivity tensors at the respective E_x, E_y and E_z locations of the grid.
The fields are now updated using the stable algorithm of Werner and Cary [9]:

```
def update_E(self, curl_H):
    curl_H_z = tls.E_Ez(curl_H)
    self.grid.E[self.locX, self.locY, 2] +=
            self.grid.sc*( tls.matrix_multiply(self.inv_epsZ, curl_H_z) )[..., 2]

    curl_H_z_y = curl_H_z.copy()
    curl_H_z_y[...,1] = 0
    curl_H_y = np.zeros((self.M, self.N, 3))
    curl_H_y[...,1] += curl_H[..., 1]
    self.grid.E[self.locX, self.locY, 1] +=
            self.grid.sc*( tls.Ez_Ey(tls.matrix_multiply(self.inv_epsZ,curl_H_z_y))
                           + tls.matrix_multiply(self.inv_epsY, curl_H_y) )[..., 1]

    curl_H_z_x = curl_H_z.copy()
    curl_H_z_x[...,0] = 0
    curl_H_x = np.zeros((self.M, self.N, 3))
    curl_H_x[...,0] += curl_H[..., 0]
    self.grid.E[self.locX, self.locY, 0] +=
            self.grid.sc*( tls.Ez_Ex(tls.matrix_multiply(self.inv_epsZ,curl_H_z_x))
                           + tls.matrix_multiply(self.inv_epsX, curl_H_x) )[..., 0]

    dG = self.s*(c*self.grid.E[self.locX, self.locY, :]**2/np.sqrt(self.inv_eps))*self.grid.dt
    self.G += tls.sumc(dG)
```

The scheme can be summarized as follows.
To have a numerically stable algorithm, it should only be allowed to multiply field components at a certain position with tensors at the same position. This forces us to store three versions of the permittivity tensor in the crystal each at a E_x, E_y or E_z position respectively. Let the superscripts {x},{y},{z} denote the position of the tensor/vector. The stable algorithm becomes:

$$E_x \mathrel{+}= \left[\texttt{inv\_eps}^{\{z\}} \cdot \begin{pmatrix} 0 \\ \texttt{curl(H)}_y^{\{z\}} \\ \texttt{curl(H)}_z \end{pmatrix}\right]^{\{x\}} + \texttt{inv\_eps}^{\{x\}} \cdot \begin{pmatrix} \texttt{curl(H)}_x \\ 0 \\ 0 \end{pmatrix}$$

$$E_y \mathrel{+}= \left[\texttt{inv\_eps}^{\{z\}} \cdot \begin{pmatrix} \texttt{curl(H)}_x^{\{z\}} \\ 0 \\ \texttt{curl(H)}_z \end{pmatrix}\right]^{\{y\}} + \texttt{inv\_eps}^{\{y\}} \cdot \begin{pmatrix} 0 \\ \texttt{curl(H)}_y \\ 0 \end{pmatrix}$$

$$E_z \mathrel{+}= \texttt{inv\_eps}^{\{z\}} \cdot \begin{pmatrix} \texttt{curl(H)}_x^{\{z\}} \\ \texttt{curl(H)}_y^{\{z\}} \\ \texttt{curl(H)}_z \end{pmatrix}$$

For the z-component of the electric field, we just interpolate all components of `curl_H` to the E_z position by using the function from the tools module `tls.E_Ez`, which moves an E-type field to the corner location of the E-field. Then the interpolated `curl_H` gets matrix multiplied by the inverse of epsilon at the E_z position by the `tls.matrix_multiply` function, which matrix multiplies the spatially varying matrix `inv_epsZ` with the spatially varying vector `curl_H` at *every* grid point. Finally, the z-component of the result is taken to update the z-component of the electric field.
For the y-component, we use the interpolated version of `curl_H` at the E_z position for which the y-component is set to zero. This vector we multiply with `inv_epsZ` and interpolate the result to the E_y location of the grid. To this partial result, the matrix multiplication of `inv_epsY` and `curl_H` at the E_y location for which the x and z component are set to zero is added te end up with the full update for the E_y component.
The x-component is updated in the same way as the y-component.
In addition to the updates, we also store how many electrons are freed due to the intensity of the light in the crystal. We store this in the parameter self.G, which will later be used in the diffusion of the electrons through the crystal.

6.6.2 Alternative version

Note: This version is not implemented.
The permittivity tensor can be implemented by assuming multiplication can only happen between components of the matrix and vector which are located at the same locations of the grid. Assuming the permittivity tensor ϵ:

$$\epsilon = \begin{pmatrix} \epsilon_{xx} & \epsilon_{xy} & \epsilon_{xz} \\ \epsilon_{yx} & \epsilon_{yy} & \epsilon_{yz} \\ \epsilon_{zx} & \epsilon_{zy} & \epsilon_{zz} \end{pmatrix}$$

And assuming that each row of the permittivity tensor lives at respectively the E_x, E_y or E_z locations of the grid:

$$
\begin{array}{|c|c|c|c|}
\hline
\ddots & \vdots & \vdots & \iddots \\
\hline
\cdots & \begin{matrix} \epsilon_{zx}^{m,n} \\ \epsilon_{zy}^{m,n} \\ \epsilon_{zz}^{m,n} \end{matrix} & \begin{matrix} \epsilon_{yx}^{m,n+\frac{1}{2}} \\ \epsilon_{yy}^{m,n+\frac{1}{2}} \\ \epsilon_{yz}^{m,n+\frac{1}{2}} \end{matrix} & \cdots \\
\hline
\cdots & \begin{matrix} \epsilon_{xx}^{m+\frac{1}{2},n} \\ \epsilon_{xy}^{m+\frac{1}{2},n} \\ \epsilon_{xz}^{m+\frac{1}{2},n} \end{matrix} & & \\
\hline
\iddots & \vdots & \vdots & \ddots \\
\hline
\end{array}
$$

For multiplication with an E-field, the E-field has to be defined at the correct grid points. We define

$$
\begin{aligned}
E^{\{x\};m,n} &= \begin{pmatrix} E_x^{m,n} \\ \frac{1}{2}\left(E_y^{m,n} + E_y^{m+1,n-1}\right) \\ \frac{1}{2}\left(E_z^{m,n} + E_z^{m+1,n}\right) \end{pmatrix} \\
E^{\{y\};m,n} &= \begin{pmatrix} \frac{1}{2}\left(E_x^{m,n} + E_x^{m-1,n+1}\right) \\ E_y^{m,n} \\ \frac{1}{2}\left(E_z^{m,n} + E_z^{m,n+1}\right) \end{pmatrix} \\
E^{\{z\};m,n} &= \begin{pmatrix} \frac{1}{2}\left(E_x^{m,n} + E_x^{m-1,n}\right) \\ \frac{1}{2}\left(E_y^{m,n} + E_y^{m,n-1}\right) \\ E_z^{m,n} \end{pmatrix}
\end{aligned}
$$

These new field can be accessed by the `Eto_Ex`, `Eto_Ey`, `Eto_Ez`, functions of the `tools` module. This way, we can define the multiplication between ϵ and E as

$$
\epsilon \cdot E \equiv \begin{pmatrix} \epsilon_{xx} & \epsilon_{xy} & \epsilon_{xz} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot E^{\{x\}} + \begin{pmatrix} 0 & 0 & 0 \\ \epsilon_{yx} & \epsilon_{yy} & \epsilon_{yz} \\ 0 & 0 & 0 \end{pmatrix} \cdot E^{\{y\}} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \epsilon_{zx} & \epsilon_{zy} & \epsilon_{zz} \end{pmatrix} \cdot E^{\{z\}} \equiv \begin{pmatrix} \epsilon_x \cdot E^{\{x\}} \\ \epsilon_y \cdot E^{\{y\}} \\ \epsilon_z \cdot E^{\{z\}} \end{pmatrix}
$$

with $\epsilon_a \equiv (\epsilon_{ax}, \epsilon_{ay}, \epsilon_{az})\ \forall a \in \{x, y, z\}$.

6.7 Electron generation and recombination

```
def diffuse(self):
    # Generation
    dn = (self.G + self.beta*self.Dt)*self.ND0
    self.n += dn
    self.ND0 -= dn

    # Electron diffusion
    self.update_n

    # Recombination
    dn = self.n*self.Dt/self.tau
    self.n -= dn
    self.ND0 += dn
```

Generation and recombination is implemented quite straightforwardly. Per time step `Dt` an amount `dn` of electrons is freed. These then diffuse through the crystal after which they recombine with the excited traps. Normally of course, this would happen simulatiously, therefore it is important not to choose `Dt` too big as this would introduce numerical errors.

6.8 Electron diffusion

```
def update_n(self):
    f = 0.01 # speedup factor
    for i in xrange(int(f*self.Dt/self.dt)+1):
        self.n = 0.5*(tls.savx(self.n) + tls.savy(self.n))
               + self.sc*(tls.sdx(self.F[...,0]) + tls.sdy(self.F[...,1]))
```

Consider the diffusion equation:

$$\frac{dn}{dt} = \left.\frac{dn}{dt}\right|_{\text{diff}} + \left.\frac{dn}{dt}\right|_{\text{drift}} = D\nabla^2 n - \nabla \cdot \mathbf{F}$$

with D the diffusion constant:

$$D = \frac{kT}{e}\mu$$

and F the electron flow:

$$F = n\mu\mathbf{S}$$

Let focus first on the diffusion part of the equation (in 2D):

$$\left.\frac{dn}{dt}\right|_{\text{diff}} = D\left(\frac{d^2}{dx^2} + \frac{d^2}{dy^2}\right) n$$

Using symmetric differences, we get:

$$\left.\frac{dn^{m,n}}{dt}\right|_{\text{diff}} = \frac{D}{du^2}\left(n^{m+1,n} + n^{m-1,n} + n^{m,n+1} + n^{m,n-1} - 4n^{m,n}\right)$$

Let's focus now on the drift part of the equation:

$$\left.\frac{dn^{m,n}}{dt}\right|_{\text{drift}} = -\nabla \cdot \mathbf{F},$$

We will now perform a Lax-Friedrich discretization [7] scheme, which uses symmetric differences in space:

$$\left.\frac{dn^{m,n}}{dt}\right|_{\text{drift}} = -\frac{1}{du}\left(\frac{F_x^{m+1,n}(q) - F_x^{m-1,n}(q)}{2} + \frac{F_y^{m,n+1}(q) - F_y^{m,n-1}(q)}{2}\right)$$

This gives for the rate of change in n:

$$\frac{dn^{m,n}}{dt} = \frac{D}{du^2}\left(n^{m+1,n} + n^{m-1,n} + n^{m,n+1} + n^{m,n-1} - 4n^{m,n}\right) - \frac{1}{du}\left(\frac{F_x^{m+1,n}(q) - F_x^{m-1,n}(q)}{2} + \frac{F_y^{m,n+1}(q) - F_y^{m,n-1}}{2}\right.$$

Which yields the update equations

$$n'^{m,n} = n^{m,n} + \frac{Ddt}{du^2}\left(n^{m+1,n} + n^{m-1,n} + n^{m,n+1} + n^{m,n-1} - 4n^{m,n}\right) - \frac{dt}{du}\left(\frac{F_x^{m+1,n}(q) - F_x^{m-1,n}(q)}{2} + \frac{F_y^{m,n+1}(q) -}{2}\right.$$

by carefully choosing the time step for this update equation as

$$dt = \frac{du^2}{4D} = \frac{edu^2}{4kT\mu},$$

The update equation gets considerably simplified:

$$n'^{m,n} = \frac{1}{4}\left(n^{m+1,n} + n^{m-1,n} + n^{m,n+1} + n^{m,n-1}\right) - \frac{edu}{4kT\mu}\left(\frac{F_x^{m+1,n}(q) - F_x^{m-1,n}(q)}{2} + \frac{F_y^{m,n+1}(q) - F_y^{m,n-1}(q)}{2}\right)$$

By defining s_c for the diffusion as

$$s_c = \frac{edu}{4kT\mu}$$

We finally get for the diffusion updates:

$$n'^{m,n} = \frac{1}{4}\left(n^{m+1,n} + n^{m-1,n} + n^{m,n+1} + n^{m,n-1}\right) - s_c\left(\frac{F_x^{m+1,n}(q) - F_x^{m-1,n}(q)}{2} + \frac{F_y^{m,n+1}(q) - F_y^{m,n-1}(q)}{2}\right)$$

Note:

- As for the FDTD updates, s_c must be smaller than $\sqrt{2}^{-1}$. However, for the mobilities found in most materials, this is trivially acheived.
- Also note that the diffusion time step $dt \ll Dt$, the generation and recombination time step. Therefore this update should be called about Dt/dt times in the `diffuse` method. In general, we choose to update until a quasi equilibrium is reached, which is usually in a lot less steps than Dt/dt, since the diffusion happens faster than the generation of free electrons.

6.9 Space Charge Field

```
def update_S(self, solver=tls.bicgstab):
    M = self.M
    N = self.N
    rl = self.du*self.charge.copy()/(self.eps_static*eps0)
    b = np.zeros((2*M*N-M-N))
    b[:M*N-1] = rl.flatten()[1:]

    self.x = solver(self.A, b, x0=self.x, tol=0.01, M=self.A.T)[0]

    x1, x2 = np.split(self.x, [M*N-N])

    S = np.zeros((M,N,3))
    S[1:, :, 0] = x1.reshape((M-1, N))
    S[:, 1:, 1] = x2.reshape((M, N-1))

    self.S = S
```

6.9.1 Introduction

The previous section implicitly assumed the knowledge of the space charge field **S**. Of course this field depends on the charge distribution in the crystal Q:

$$\rho = e(N_D^+ - n - N_c) \equiv \texttt{rho}$$

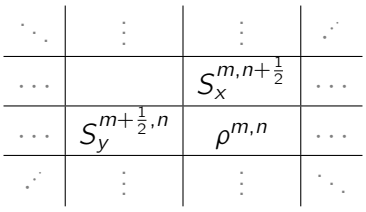
where N_c is the compensating charge, defined as the charge necessary to make the whole crystal charge neutral:

$$N_c = N_D^+(t_0) - n(t_0)$$

From the static maxwell equations in a dielectric, we get

$$\nabla \cdot \mathbf{S} = \frac{\rho}{\epsilon\epsilon_0}$$
$$\nabla \times \mathbf{S} = \mathbf{0}$$

Where we assumed that **S** varies slow enough to allow for the second equation, and where ϵ now denotes the relative permittivity for static fields. We will now solve the equations again on a staggered grid. However, we choose to stagger the Remnant field as an H-field:



We solve the first equation in the center of the grid:

$$S_x^{m+1,n+\frac{1}{2}} - S_x^{m,n+\frac{1}{2}} + S_y^{m+\frac{1}{2},n+1} - S_y^{m+\frac{1}{2},n} = \frac{\rho^{m,n}}{\epsilon\epsilon_0}$$
$$\texttt{S[m+1,n,0]} - \texttt{S[m,n,0]} + \texttt{S[m,n+1,1]} - \texttt{S[m,n,1]} = \frac{\texttt{rho[m,n]}}{\texttt{eps\_static} \cdot \texttt{eps0}},$$

While the second equation can be solved on the top left corner:

$$S_y^{m+\frac{1}{2},n} - S_y^{m-\frac{1}{2},n} - S_x^{m,n+\frac{1}{2}} + S_x^{m,n-\frac{1}{2}} = 0$$
$$\texttt{S[m,n,1]} - \texttt{S[m-1,n,1]} - \texttt{S[m,n,0]} + \texttt{S[m,n-1,0]} = 0$$

This gives two equations per cell, while there are also two unknowns per cell. This should thus give a solvable system.
Solving this system with 2MN unknowns and 2MN equations reduces to solving a linear system of the form

$$A\mathbf{x} = \mathbf{b}$$

With **x** the vector of unknowns:

$$\mathbf{x} = \begin{pmatrix} \texttt{S[0,0,0]} \\ \texttt{S[0,1,0]} \\ \vdots \\ \texttt{S[-1,-1,0]} \\ \vdots \\ \texttt{S[-1,-1,1]} \end{pmatrix}$$

b the targets:

$$\mathbf{b} = \begin{pmatrix} \texttt{rho[0,0]} \\ \texttt{rho[0,1]} \\ \vdots \\ \texttt{rho[-1,-1]} \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

and A the block matrix containing four blocks

$$A = \begin{pmatrix} \texttt{divX} & \texttt{divY} \\ \texttt{curlX} & \texttt{curlY} \end{pmatrix}$$

6.9.2 The matrix A

The general form of `divX` The general form of `divX` can be generalized as

$$\texttt{divX[k,:]} = \begin{pmatrix} 0 & \cdots & 0 & -1 & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \end{pmatrix},$$

where the -1 is at the k'th position in this row matrix and the 1 at the $k+N$'th position for an $M \times N$ grid. This is defined such that

$$\begin{pmatrix} \texttt{divX[k,:]} & 0 & \cdots & 0 \end{pmatrix} \cdot \mathbf{x} = S_x^{m+1,n+\frac{1}{2}} - S_x^{m,n+\frac{1}{2}}$$

with

$$m = k \bmod N$$
$$n = \lfloor k/N \rfloor$$

For a 3×4 grid, this becomes for example

$$\texttt{divX[k,:]} = \left(\begin{array}{cccccccccccc} -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{array}\right)$$

General form of `divY` Similarly, the general form of `divY`:

$$\texttt{divY[k,:]} = \begin{pmatrix} 0 & \cdots & 0 & -1 & 1 & 0 & \cdots & 0 \end{pmatrix},$$

where the -1 is always at the k'th position in this row matrix and the 1 is only at the $(k+1)$'th position if $(k+1) \bmod N \neq 0$. for an $M \times N$ grid. This is defined such that

$$\begin{pmatrix} 0 & \cdots & 0 & \texttt{divY[k,:]} \end{pmatrix} \cdot \mathbf{x} = S_y^{m+\frac{1}{2},n+1} - S_y^{m+\frac{1}{2},n}$$

with

$$m = k \bmod N$$
$$n = \lfloor k/N \rfloor$$

For a 3×4 grid, this becomes for example

$$\texttt{divY[k,:]} = \left(\begin{array}{cccccccccccc} -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{array}\right)$$

General form of `curlX` The general form of `curlX`:

$$\texttt{curlX[k,:]} = \begin{pmatrix} 0 & \cdots & 0 & 1 & -1 & 0 & \cdots & 0 \end{pmatrix},$$

Where the -1 is always at the k'th position, while the 1 is at the $k-1$ position, only if $k \bmod N \neq 0$. We get

$$\begin{pmatrix} \texttt{curlX[k,:]} & 0 & \cdots & 0 \end{pmatrix} \cdot \mathbf{x} = S_x^{m,n-\frac{1}{2}} - S_x^{m,n+\frac{1}{2}}$$

This becomes for a 3×4 grid:

$$\texttt{curlX[k,:]} = \left(\begin{array}{cccccccccccc} -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{array}\right)$$

General form of `curlY` The general form of `curlY`:

$$\texttt{curlY[k,:]} = \begin{pmatrix} 0 & \cdots & 0 & -1 & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \end{pmatrix},,$$

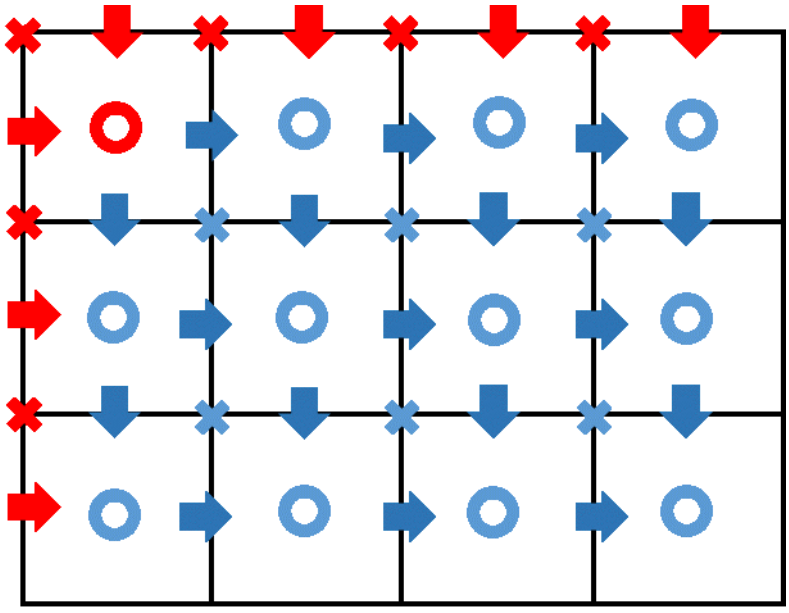
Where the 1 is always at the k'th position, while the -1 is at the $k - N$'th position. We get

$$\begin{pmatrix} 0 & \cdots & 0 & \texttt{curlY[k,:]} \end{pmatrix} \cdot \mathbf{x} = S_x^{m,n-\frac{1}{2}} - S_x^{m,n+\frac{1}{2}}$$

This becomes for a 3×4 grid:

$$\texttt{curlY[k,:]} = \left(\begin{array}{cccccccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 \end{array}\right)$$

6.9.3 What about the edges?



A depiction of the Space Charge Fields in a 3×4 grid. The direction of the arrows signify the component of the field. The crosses signify the locations where the curl equations are solved. The circles signify the locations where the charge equations are solved. The locations indicated in red are discarded.

Variables on the edges Unfortunately, this is not the end of the story. A depiction on the 3×4 grid explains.
Normally, the red arrows are included in the equations. This obviously makes the dynamics not symmetric. The easiest solution to this problem is just setting these components to zero. Since these variables are set to zero anyway, any product with them will result in a zero and thus the variables can be removed from $\mathbf{x}$ and the columns corresponding to a multiplication with any of these variables can be removed from the matrix A.
This is equivalent to removing the first N columns of `divX` and `curlX`, and removing M columns of `divY` and `curlY` starting at the first column (index 0), spaced N apart.
However, now we have a system with $2MN$ equations and $2MN - M - N$ variables. This is overdetermined and some equations should be removed as well.

Equations on the corners Luckily, a similar story holds for the equations at the corners, where the curl equations at locations where there are three or more neighbouring zero-fields should be discarded, because this would force the forth component to be zero as well. This off course would ultimately result in a zero field in the whole crystal.
Removing the necessary equations is equivalent with removing the lines of `curlX` and `curlY` with only one 1 or -1. This in total removes M+N-1 equations.

Total charge Unfortunately, this is still an overdetermined system with $2MN - M - N$ variables and $2MN - M - N + 1$ equations. However, there is still one trick that can be performed: the total charge. Since the total charge should always sum to zero, the sum of all components of $\mathbf{b}$ should be zero as well. This allows us to discard one arbitrary equation. We choose the top left cell as this has already two edge components set to zero. This finally gives us a system with $2MN - M - N$ equations and $2MN - M - N$ unknowns.

6.9.4 Solving the system

In solving the system, first the matrix A has to be created. This is done during the initialization of the crystal by the `sparse_matrix` function from the `tools` module.
Next, every time step `Dt` of the diffusion loop, the space charge field needs to be updated. There are several ways to go about this. The worst possible way is arguably storing the inverse of the matrix A in stead of the matrix A. and always multiplying this matrix with the constructed $\mathbf{b}$ matrix to arrive at the space charge field. This is a bad idea for three reasons. First, it is not sure A has an inverse in the first place. Secondly the inverse of a sparse matrix is not generally sparse itself, which will result in a large memory usage; let alone the computational effort of inverting this very large matrix in the first place. Thirdly, using this approach we always calculate the space charge from scratch. Since we are talking about updating every time step, we are only changing the charge distribution a little bit. Therefore, the space charge field should also only change a little bit.
Another approach is using one of scipy's linear algebra solvers for sparse matrices such as

`scipy.sparse.linalg.spsolve`

To solve the system $A\mathbf{x} = \mathbf{b}$ without inverting the matrix A. This solves the memory problem of storing and computing with a dense matrix, and is generally faster and more accurate than multiplying with an inverse anyway. However, it does not solve the problem that the space charge field is computated from scratch every time.

The approach chosen for in this code is using the stabilized biconjugate gradient method [2] `python scipy.sparse.linalg.bicgstab` to solve for the field.
This `bicgstab` method solves for the field by taking an estimate for the field (in this case the field values before the time step) and a preconditioner (generally a matrix that multiplies A in order to make it symmetric$. This way of solving for the space charge field works at least 10 times faster than the normal scipy solver.

References

[1] O Beyer, D Maxein, K Buse, B Sturman, HT Hsieh, and D Psaltis. Femtosecond time-resolved absorption processes in lithium niobate crystals. *Optics letters*, 30(11):1366–1368, 2005.

[2] Roger Fletcher. Conjugate gradient methods for indefinite systems. In *Numerical analysis*, pages 73–89. Springer, 1976.

[3] Vladimir M Fridkin. *Photoferroelectrics*, volume 9. Springer Science & Business Media, 2012.

[4] NA Gusak and NS Petrov. On the dependence of the free carrier concentration on light intensity in photorefractive crystals. *Technical Physics*, 46(5):635–637, 2001.

[5] Nikolaiv Kukhtarev et al. Holographic storage in electrooptic crystals. *Ferroelectrics*, 22(1):949–960, 1978.

[6] David Pepper et al. The photorefractive effect. *Scientific American*, 263(4):34–40, 1990.

[7] Luciano Rezzolla. *Numerical Methods for the Solution of Hyperbolic Partial Differential Equations*. 2005.

[8] John B Schneider. *Understanding the finite-difference time-domain method*. 2010.

[9] Gregory R. Werner and John R. Cary. A stable fdtd algorithm for non-diagonal, anisotropic dielectrics. *Journal of Computational Physics*, 226(1):1085 – 1101, 2007.

[10] Kane S Yee et al. Numerical solution of initial boundary value problems involving maxwells equations in isotropic media. *IEEE Trans. Antennas Propag*, 14(3):302–307, 1966.