Simulation Electrodynamics

Rasmus Bruhn Nielsen

1 Non-Responsive Electrodynamics

The goal of these notes will be to develop a way to simulate electrodynamics. We will approximate Maxwell's equations to be able to simulate them with focus on speed of the simulation as well as saving memory. In this chapter we want to focus on non-responsive electrodynamics, that is electrodynamics where no charges respond to the induced electric or magnetic fields. Possible expansions of the model could be to allow current flow and charge displacement in response to electromagnetic fields (metals), polarization of materials (dielectrics) and magnetization (dia-, para- and ferromagnets). Throughout the notes we will be using the text book "Introduction to Electrodynamics" by David J. Griffiths, we will be referring to an equation from this as (Griffiths, [Number]).

It is not meant as a perfect solver with very good efficiency. The focus will be on making a simple interface which is easy to use. This way the students can play around with electrodynamics to get a better feeling for the concepts. If you are interested in a high efficiency, high accuracy simulator you should look elsewhere. The library can be downloaded using "pip install KUEM"

1.1 The Theory

To start off with we have Maxwell's equations (Griffiths, 7.40)

$$\begin{aligned} \boldsymbol{\nabla} \cdot \mathbf{E} &= \frac{\rho}{\epsilon_0} \\ \boldsymbol{\nabla} \cdot \mathbf{B} &= 0 \\ \boldsymbol{\nabla} \times \mathbf{E} &= -\frac{\partial \mathbf{B}}{\partial t} \\ \boldsymbol{\nabla} \times \mathbf{B} &= \mu_0 \mathbf{J} + \mu_0 \epsilon_0 \frac{\partial \mathbf{E}}{\partial t} \end{aligned} \tag{1}$$

Where $\mathbf{E}, \mathbf{B}$ are the electric and magnetic fields, $\rho, \mathbf{J}$ are the charge and current density and ϵ_0, μ_0 are the permittivity and permeability of free space with the relation $\frac{1}{\epsilon_0 \mu_0} = c^2$ where c is the speed of light.

These are the differential equations we want to simulate, also when we actually do simulations we usually define $c \equiv 1$ and $\mu_0 \equiv 1$ which also means $\epsilon_0 = 1$. This will make sure that all numbers we get is around the order of 1 which is easier to understand and we run into less trouble with the computer.

The big problem with these equations are that there are 6 functions which are determined by first order partial differential equations mixing all the functions together. If we instead use potentials we can simplify this a lot. Then we can turn the problem into finding 4 functions determined by independent second order partial differential equations. This will be much easier to simulate as we don't need to take into account the effects of the functions on each other and we go from 6 to 4 functions which will save time and memory. The potentials

V and $\mathbf{A}$ are defined by:

$$\begin{aligned}\mathbf{E} &= -\boldsymbol{\nabla} V - \frac{\partial \mathbf{A}}{\partial t} \\ \mathbf{B} &= \boldsymbol{\nabla} \times \mathbf{A}\end{aligned} \tag{2}$$

Using potentials and the Lorenz gauge transformation, Maxwell's equations become (Griffiths, 10.16)

$$\begin{aligned}\frac{1}{c}\Box^2 V &= -\mu_0 c\rho \\ \Box^2 \mathbf{A} &= -\mu_0 \mathbf{J}\end{aligned} \tag{3}$$

Where $\Box^2 \equiv \nabla^2 - \frac{1}{c^2}\frac{\partial^2}{\partial t^2}$

We can write this equation event shorter if we define:

$$\begin{aligned}A^\mu &= \left(\frac{1}{c}V, A_x, A_y, A_z\right) \\ J^\mu &= (c\rho, J_x, J_y, J_z)\end{aligned} \tag{4}$$

Here A^μ and J^μ are just 4 dimensional vectors. You get a component of one of these vectors by inserting a number instead of μ. So $A^0 = \frac{1}{c}V$ and $A^2 = A_y$. This notation is reminiscent of special relativity where this is 4-vector notation, but here we will not consider special relativity and it is just a normal vector. Then our defining differential equation becomes:

$$\Box^2 A^\mu = -\mu_0 J^\mu \tag{5}$$

It is the exact same 4 equations just written in a different notation. Now we have achieved what we wanted, we have now obtained 4 independent second order partial differential equations. So now we just need to simulate this much simpler equation and then we can calculate the electromagnetic fields using Eq. 2.

1.2 The Model

We will model our system as a 3D grid with a finite resolution along each axis, and we will use Cartesian coordinates. The resolution along the axis will be given as $\Delta x, \Delta y, \Delta z$. For shorthand notation we will write $\Delta\mathbf{x} \equiv (\Delta x, \Delta y, \Delta z)$ and $\mathbf{x} \equiv (x, y, z)$. We also assume that the boundary conditions are predefined such that the potential is 0 everywhere on the boundary. This means that for a function $f(x, y, z, t)$ of 3 spacial variables and 1 time variable, we sample it as $f(n_x\Delta x, n_y\Delta y, n_z\Delta z, t)$ with the value $n_i \in \mathbb{N}$ and $0 \leq n_i < N_i$ with $N_i \in \mathbb{N}$. Here $\mathbf{N} \equiv (N_x, N_y, N_z)$ is the resolution of the model and $\mathbf{n} \equiv (n_x, n_y, n_z)$ determines the position. Again we introduce shorthand notation $f(\mathbf{x_n}, t) \equiv f(n_x\Delta x, n_y\Delta y, n_z\Delta z, t)$ where $\mathbf{x_n} \equiv n_x\Delta x\hat{\mathbf{x}} + n_y\Delta y\hat{\mathbf{y}} + n_z\Delta z\hat{\mathbf{z}}$. Sometimes we will also use $x = x^1, y = x^2, z = x^3$. We will also model this in discrete time intervals with the length of the intervals being Δt and we will assume we start at time $t_0 = 0$.

For our model to be good we need that the change of any function f in a time interval Δt is small, and we need that the functions we will simulate don't change much between neighbouring points. For the first condition to be correct we need that $\frac{\Delta x_i}{\Delta t} >> c$ since then changes will take many frames to propagate and thus it should happen slowly. Basically Δt needs to be small enough. And for the second condition to be true we must have that the distance between points Δx_i is small enough. What we are saying here is just that the resolution of our problem should be small enough for a given problem, which is something that needs to be true for all simulations. If these conditions are not satisfied we will quickly notice since our solutions will probably diverge and create unphysical situations.

We will also require for any function f that:

$$\left|\sum_{n=3}^{\infty}\frac{1}{n!}\frac{\partial^n f}{\partial x^n}(\Delta x)^n\right| << \left|\frac{\partial f}{\partial x}\Delta x + \frac{1}{2}\frac{\partial^2 f}{\partial x^2}(\Delta x)^2\right| \tag{6}$$

For some variable x and the resolution in that variable Δx, note that x could also be time here. We assume this so that we can Taylor expand functions to second order later on. This is basicly the same condition as requiring that Δx_i is small enough.

Partial Derivatives

Before we can go on we first needs to do some approximations to the partial derivatives. We will not consider partial derivatives of orders greater that 2 since we will not need these. Since our functions are no longer continuous we can no longer calculate the partial derivatives of our functions. Therefor we will a way to approximate the derivatives. To do this we will use Taylor expansion.

Assume we have a function f, it is a function of some variable x with a resolution Δx. The function could also depend on more variables but we are only interested in x since we only consider partial derivatives. Now Taylor expand the function around some value $x = x_0$

$$f(x+\delta x) \approx f(x_0) + \left.\frac{\partial f}{\partial x}\right|_{x=x_0}\delta x + \frac{1}{2}\left.\frac{\partial^2 f}{\partial x^2}\right|_{x=x_0}(\delta x)^2 \tag{7}$$

Using this we see that

$$\begin{aligned}\left.\frac{\partial f}{\partial x}\right|_{x=x_0} &\approx \frac{1}{2\Delta x}(f(x_0+\Delta x) - f(x_0-\Delta x))\\ \left.\frac{\partial^2 f}{\partial x^2}\right|_{x=x_0} &\approx \frac{1}{(\Delta x)^2}(f(x_0+\Delta x) + f(x_0-\Delta x) - 2f(x_0))\end{aligned} \tag{8}$$

1.3 Solving The Problem

We are now ready to solve our problem to find a way to simulate electromagnetic fields in the non-responsive electrodynamics case. First we make some assumptions.

Assumptions

We assume that the generalised current vector J^μ is a predefined function, it can change with time but it must be predefined. Otherwise it would be responsive. We also assume that we have some starting conditions. So we assume given

$$\begin{aligned} &J^\mu(\mathbf{x}, t) \\ &A^\mu(\mathbf{x}, t_0) \\ &\left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0} \end{aligned} \tag{9}$$

For all $\mathbf{x}$ and some initial time t_0.

We will also assume the boundary condition that $A^\mu = 0$ outside our model.

Idea

The idea behind our simulation will be to evaluate $J^\mu\big(\mathbf{x}, t + \frac{1}{2}\Delta t\big)$ and $\left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0+\frac{1}{2}\Delta t}$ for all $\mathbf{x}$. Then using these we can evolve the system from time t_0 to time $t_0 + \Delta t$. The reason why we don't just evolve the system directly without calculating the state of the system at time $t_0 + \frac{1}{2}\Delta t$ is that this would be very unstable. That method is known as Euler integration and it has a tendency to diverge. By doing this extra step our simulation is much safer from breaking down.

Approximating

In the following if no space coordinate or the time coordinate is given for a function then $\mathbf{x}$ or t is implicitly included. From Eq. 5 we get

$$\frac{\partial^2 A^\mu}{\partial t^2} = c^2\big(\nabla^2 A^\mu + \mu_0 J^\mu\big) \tag{10}$$

If we now Taylor expand A^μ at the time $t_0 + \frac{1}{2}\Delta t$ then we get

$$\begin{aligned} A^\mu\left(t_0 + \frac{1}{2}\Delta t + \delta t\right) &= A^\mu\left(t_0 + \frac{1}{2}\Delta t\right) + \left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0+\frac{1}{2}\Delta t}\delta t + \frac{1}{2}\left.\frac{\partial^2 A^\mu}{\partial t^2}\right|_{t=t_0+\frac{1}{2}\Delta t}(\delta t)^2 \\ &= A^\mu\left(t_0 + \frac{1}{2}\Delta t\right) + \left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0+\frac{1}{2}\Delta t}\delta t + \frac{1}{2}c^2\left(\left.\nabla^2 A^\mu\right|_{t=t_0+\frac{1}{2}\Delta t} + \mu_0 J^\mu\left(t_0 + \frac{1}{2}\Delta t\right)\right)(\delta t)^2 \end{aligned} \tag{11}$$

And by differentiating Eq. 11 with respect to δt we get

$$\left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0+\frac{1}{2}\Delta t+\delta t} = \left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0+\frac{1}{2}\Delta t} + c^2\left(\left.\nabla^2 A^\mu\right|_{t=t_0+\frac{1}{2}\Delta t} + \mu_0 J^\mu\left(t_0 + \frac{1}{2}\Delta t\right)\right)\delta t \tag{12}$$

Now we need $A^\mu\big(t_0 + \frac{1}{2}\Delta t\big)$ and $\left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0+\frac{1}{2}\Delta t}$ in terms of the starting conditions from Eq. 9. By inserting $\delta t = -\frac{1}{2}\Delta t$ in Eq. 12 we get

$$\left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0} = \left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0+\frac{1}{2}\Delta t} - \frac{1}{2}c^2\left(\left.\nabla^2 A^\mu\right|_{t=t_0+\frac{1}{2}\Delta t} + \mu_0 J^\mu\left(t_0+\frac{1}{2}\Delta t\right)\right)\Delta t \tag{13}$$

By rearranging we get

$$\left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0+\frac{1}{2}\Delta t} = \left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0} + \frac{1}{2}c^2\left(\left.\nabla^2 A^\mu\right|_{t=t_0+\frac{1}{2}\Delta t} + \mu_0 J^\mu\left(t_0+\frac{1}{2}\Delta t\right)\right)\Delta t \tag{14}$$

We can insert this into Eq. 11 to get

$$A^\mu\left(t_0+\frac{1}{2}\Delta t+\delta t\right) = A^\mu\left(t_0+\frac{1}{2}\Delta t\right) + \left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0}\delta t + \frac{1}{2}c^2\delta t(\delta t+\Delta t)\left(\left.\nabla^2 A^\mu\right|_{t=t_0+\frac{1}{2}\Delta t} + \mu_0 J^\mu\left(t_0+\frac{1}{2}\Delta t\right)\right) \tag{15}$$

Setting $\delta t = -\frac{1}{2}\Delta t$ in Eq. 15 we get

$$A^\mu(t_0) = A^\mu\left(t_0+\frac{1}{2}\Delta t\right) - \frac{1}{2}\left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0}\Delta t - \frac{1}{8}c^2(\Delta t)^2\left(\left.\nabla^2 A^\mu\right|_{t=t_0+\frac{1}{2}\Delta t} + \mu_0 J^\mu\left(t_0+\frac{1}{2}\Delta t\right)\right) \tag{16}$$

So we get

$$A^\mu\left(t_0+\frac{1}{2}\Delta t\right) - \left.\frac{1}{8}c^2(\Delta t)^2\nabla^2 A^\mu\right|_{t=t_0+\frac{1}{2}\Delta t} = A^\mu(t_0) + \frac{1}{2}\left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0}\Delta t + \frac{1}{8}c^2(\Delta t)^2\mu_0 J^\mu\left(t_0+\frac{1}{2}\Delta t\right) \equiv R^\mu(\mathbf{x},t_0) \tag{17}$$

This is an implicit equation for $A^\mu\left(t_0+\frac{1}{2}\Delta t\right)$ which we will need to solve. Once we have solved it we can rearrange Eq. 17 to get

$$\frac{1}{8}c^2(\Delta t)^2\left(\left.\nabla^2 A^\mu\right|_{t=t_0+\frac{1}{2}\Delta t} + \mu_0 J^\mu\left(t_0+\frac{1}{2}\Delta t\right)\right) = A^\mu\left(t_0+\frac{1}{2}\Delta t\right) - A^\mu(t_0) - \frac{1}{2}\left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0}\Delta t \tag{18}$$

Then by inserting $\delta t = \frac{1}{2}\Delta t$ and Eq. 18 into Eq. 15 and Eq. 12 we get

$$\begin{aligned} A^\mu(t_0+\Delta t) &= 4A^\mu\left(t_0+\frac{1}{2}\Delta t\right) - 3A^\mu(t_0) - \left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0}\Delta t \\ \left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0+\Delta t} &= \frac{8}{\Delta t}\left(A^\mu\left(t_0+\frac{1}{2}\Delta t\right) - A^\mu(t_0)\right) - 3\left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0} \end{aligned} \tag{19}$$

Calculating The Result

Now we have a theoretical way to determine $A^\mu(t_0+\Delta t)$ and $\left.\frac{\partial A^\mu}{\partial t}\right|_{t=t_0+\Delta t}$ using $A^\mu(t_0)$ for Eq. 19 and Eq. 17. Although we have these equations, we still do not have any way to calculate it since we don't know how to calculate $\nabla^2 A^\mu$. So now we need to use the fact that we are using Cartesian coordinates. Until now it has been completely general but now we will write the laplacian in Cartesian coordinates

$$\nabla^2 A^\mu = \frac{\partial^2 A^\mu}{\partial x^2} + \frac{\partial^2 A^\mu}{\partial y^2} + \frac{\partial^2 A^\mu}{\partial z^2} \tag{20}$$

Using Eq. 8 we can approximate this as

$$\nabla^2 A^\mu(\mathbf{x}) = \sum_{i\in\{x,y,z\}} \left(\frac{1}{(\Delta x_i)^2} (A^\mu(\mathbf{x} + \Delta x_i \hat{\mathbf{x}}_\mathbf{i}) + A^\mu(\mathbf{x} - \Delta x_i \hat{\mathbf{x}}_\mathbf{i}) - 2A^\mu(\mathbf{x})) \right) \tag{21}$$

Where everything is evaluated at the same time. Inserting this into Eq. 17 we get

$$A^\mu(\mathbf{x}) - \frac{1}{8} c^2 (\Delta t)^2 \sum_{i\in\{x,y,z\}} \left(\frac{1}{(\Delta x_i)^2} \left(A^\mu(\mathbf{x} + \Delta x_i \hat{\mathbf{x}}_\mathbf{i}) + A^\mu(\mathbf{x} - \Delta x_i \hat{\mathbf{x}}_\mathbf{i}) - 2A^\mu\left(\mathbf{x}, t_0 + \frac{1}{2}\Delta t\right) \right) \right) = R^\mu(\mathbf{x}, t_0) \tag{22}$$

Where everything on the left side is evaluated at the time $t_0 + \frac{1}{2}\Delta t$.

This is a beastly set of equations, if we define $V \equiv N_x N_y N_z$ then we have V variables with V coupled equations. For a simulation we will expect V to be between 10^6 and 10^9 so we have a lot of equations. Luckily they are linear, so we can solve the problem using linear algebra. We could also try to solve it using other methods but linear algebra is easy to use and it is very widely used, so it is very optimised.

To use linear algebra we need to be able to put all values of A^μ into a vector. This can be done in many different ways but we choose to define $\mathbf{A}^\mu$ such that

$$A^\mu(n_x \Delta x, n_y \Delta y, n_z \Delta z) = (\mathbf{A}^\mu)_{n_x + n_y N_x + n_z N_x N_y} \tag{23}$$

Now we can define the matrix representing the laplacian

$$(\mathbf{M}^\mu)_{nm} = \begin{cases} -2\sum_{i\in\{x,y,z\}} \frac{1}{(\Delta x_i)^2} & \text{for } m = n \\ \frac{1}{(\Delta x)^2} & \text{for } (m = n+1 \text{ and } n_x \neq N_x - 1) \text{ or } (m = n-1 \text{ and } n_x \neq 0) \\ \frac{1}{(\Delta y)^2} & \text{for } (m = n + N_x \text{ and } n_y \neq N_y - 1) \text{ or } (m = n - N_x \text{ and } n_y \neq 0) \\ \frac{1}{(\Delta z)^2} & \text{for } (m = n + N_x N_y \text{ and } n_z \neq N_z - 1) \text{ or } (m = n - N_x N_y \text{ and } n_z \neq 0) \\ 0 & \text{else} \end{cases} \tag{24}$$

For $n = n_x + n_y N_x + n_z N_x N_y$.

Now we can rewrite the equations to

$$\left(\mathbf{I} - \frac{1}{8} c^2 (\Delta t)^2 \mathbf{M}^\mu \right) \mathbf{A}^\mu = \mathbf{R}^\mu \tag{25}$$

Really what we have here is 4 matrix equations, one for each value of μ. Now we can also notice that the demand that $\frac{\Delta x_i}{\Delta t} >> c$ is equivalent to saying that $\left\| \left(\frac{1}{8} c^2 (\Delta t)^2 \mathbf{M}^\mu \right)_{nm} \right\| << 1$ or $\left(\mathbf{I} - \frac{1}{8} c^2 (\Delta t)^2 \mathbf{M}^\mu \right) \approx \mathbf{I}$.

Now we just needs to solve this matrix equation. There are however 2 problems. First lets look at the size of the matrix, it is of size V^2. If we are to store this matrix on a computer then for a small system with $V = 10^6$ and 32-bit floating point values we will still need 4000 gigabytes of memory. This is completely impossible on a normal laptop, but luckily there is a way around this. We notice that the matrix in question mostly consists of 0's and therefor we can use something known as a sparse matrix. This is a matrix where you only save the non-zero elements. Then the memory usage is of order V and not V^2 which means that we can save it in the memory. The other problem is that if we try to use a linear solver we will have to wait for a long time to simulate anything. The reason for this is that the speed of such a solver is of the order of the size of the matrix, so V^2. Since we need to solve this for every single time interval, the simulation will never end. So we must find an approximation. If we rearrange we get

$$\mathbf{A}^\mu = \frac{1}{8} c^2 (\Delta t)^2 \mathbf{M}^\mu \mathbf{A}^\mu + \mathbf{R}^\mu \tag{26}$$

We will now introduce a series $(\mathbf{A}^\mu_\mathbf{0}, \mathbf{A}^\mu_\mathbf{1}, ...)$ defined by

$$\begin{aligned} \mathbf{A}^\mu_{\mathbf{n+1}} &= \frac{1}{8} c^2 (\Delta t)^2 \mathbf{M}^\mu \mathbf{A}^\mu_\mathbf{n} + \mathbf{R}^\mu \\ \mathbf{A}^\mu_\mathbf{0} \left(t_0 + \frac{1}{2} \Delta t \right) &= \mathbf{A}^\mu(t_0) + \frac{1}{2} \Delta t \left. \frac{\partial \mathbf{A}^\mu}{\partial t} \right|_{t=t_0} \end{aligned} \tag{27}$$

We notice that if the series $\{\mathbf{A}^\mu_\mathbf{n}\}$ converges then the limit is

$$\lim_{n \to \infty} \mathbf{A}^\mu_\mathbf{n} \left(t_0 + \frac{1}{2} \Delta t \right) = \mathbf{A}^\mu \left(t_0 + \frac{1}{2} \Delta t \right) \tag{28}$$

And from testing in Python we see that it does converge as long as $\frac{\Delta x_i}{\Delta t}$ is large enough. Using the method the speed of the calculation will be of order V. It will not be exact but if changes happen slowly enough then it will give the answer to a high precision within just a few iterations.

1.4 Non-Responsive Electrostatics

Now we have a complete model and we can implement it in Python. It is fairly easy to do since it is just some linear algebra. But when we try it out we get nonsense results. As an example a point charge will create an oscillating field. This makes no sense. The reason for this is actually not a numerical error but arises because our starting conditions are non-physical. For the point charge, the starting condition was just $\mathbf{A}^\mu = \mathbf{0}$. But we can't just make a charge appear in empty space. To fix this we must first calculate the starting condition. What we will do is assume that the system is frozen in time at $t = 0$ and calculate the starting condition here, then we will use this result as the starting condition for the simulation. This means that we have to perform

an electrostatics problem now.

Changing The Model

For electrostatics we have $\frac{\partial^2 A^\mu}{\partial t^2} = 0$, then Eq. 10 becomes

$$\nabla^2 A^\mu = -\mu_0 J^\mu \tag{29}$$

Representing this as a matrix equation we get

$$\mathbf{M}^\mu \mathbf{A}^\mu = -\mu_0 \mathbf{J}^\mu \tag{30}$$

Again using a linear solver will be very slow but may be possible here since we only need to solve it once, and not for every time interval. If it is too slow we can make the same kind of approximating as before

$$\begin{aligned} \mathbf{A}^\mu_{\mathbf{n+1}} &= (\mathbf{I} + k\mathbf{M}^\mu)\mathbf{A}^\mu_{\mathbf{n}} + k\mu_0 \mathbf{J}^\mu \\ \mathbf{A}^\mu_{\mathbf{0}} &= \mathbf{0} \end{aligned} \tag{31}$$

For any number k. We will hope that this series does not diverge, if it converges it should converge on the correct result. We can also define a different $\mathbf{A}^\mu_{\mathbf{0}}$ if we have a better guess, this may make the convergence faster.

Determining Convergence Parameters

To optimise our algorithm we should find the optimal values for convergence of the series from Eq. 31. This means we have to figure out how many steps is needed to converge and what the optimal value for k is. This is not a trivial task. If k is too small then not much happens between iterations and it will take a very long time to converge. On the other hand if k is too large then it will overshoot the correct result and maybe even diverge. To find the best value for k we can try to write an expression for it which is not recursive. After fiddling around with the equation we can arrive at the expression

$$\mathbf{A}^\mu_{\mathbf{n}} = (\mathbf{I} + k\mathbf{M}^\mu)^n \mathbf{A}^\mu_{\mathbf{0}} + k\mu_0 \sum_{i=0}^{n-1} (\mathbf{I} + k\mathbf{M}^\mu)^i \mathbf{J}^\mu \tag{32}$$

From this expression we see what we want. We want the first term to go to zeros as fast as possible because of cause we don't want our final answer to depend on our initial guess. We also want the sum to converge as fast as possible. Both of these conditions actually require the same thing, that the matrix $\mathbf{I} + k\mathbf{M}^\mu$ is as close to zeros as possible. But what does it mean for a matrix to be close to zero. To answer this we can use the norm of the matrix, which is a measure of how "big" the matrix is. The norm of a matrix $\|\mathbf{C}\|$ is defined as the maximum length of the vector $\mathbf{Cv}$ given that the norm of the vector $\mathbf{v}$ is $\|\mathbf{v}\| = 1$. This is written as

$$\|\mathbf{C}\| \equiv \max\left(\|\mathbf{Cv}\|; \|\mathbf{v}\| = 1\right) \tag{33}$$

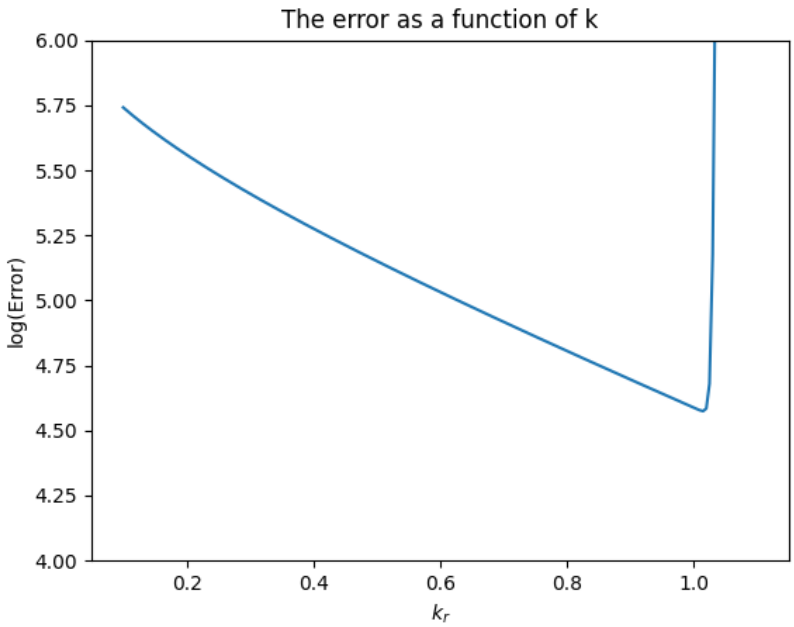

Figure 1: *A plot of the logarithm of the average error as a function of* k_r *where* $k = k_r k_0$ *and* $k_0 = \frac{1}{2\sum_{i\in\{x,y,z\}} \frac{1}{(\Delta x_i)^2}}$.

So now all we need to do is to minimise $\|\mathbf{I} + k\mathbf{M}^{\mu}\|$ by changing k right?. Well the problem is that actually doing the minimisation is a horror of an exercise and we will not attempt to do this here. Instead we will just determine the optimal value of k by doing the simulation on a system small enough to find the exact result. Then we can calculate the error as a function of k and then minimise.
In figure 1 we see that the optimal value for k is approximately $k_0 = \frac{1}{2\sum_{i\in\{x,y,z\}} \frac{1}{(\Delta x_i)^2}}$ which is exactly minus one divided by the diagonal element of the laplacian. This is probably not an accident though, because with this value of k the diagonal of $\mathbf{I} + k\mathbf{M}^{\mu}$ is 0 and the off diagonals a of the order 0.1, so it does make sense that this value of k is good. We also see that if k is just a bit larger then the series quickly diverge. Actually the value of k which minimises the error is a few percent higher ($k_r \approx 1.01$) but this value changes depending on the geometry of the problem so to be safe we will just use the value $k_0 = 1$.

The other value we needed to fix was the number of steps required, lets call is n_{max}. Every step we update the value in one cell depending on the values in the neighbouring cells. This means that it would take in total $N_x + N_y + N_z$ steps for the corners to exchange information. A naive approach would then be to assume that $n_{max} \propto N_x + N_y + N_z$. But this is not true. After doing some testing we can conclude that with an unreasonable accuracy we have

$$n_{max} = a\big(N_x^2 + N_y^2 + N_z^2\big) \tag{34}$$

Where a is just a constant which depends on how low we want the error to be. I cannot explain why is has this form but we will still use it since we can show empirically that it does.

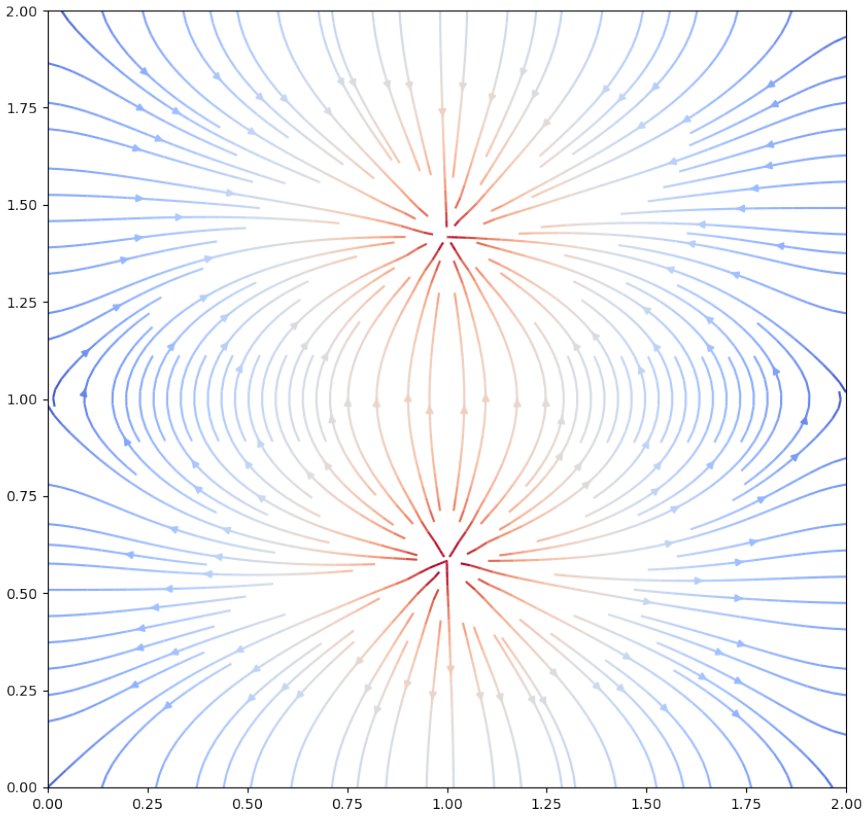

Figure 2: *Electric field of an electric dipole with closed boundary conditions*

1.5 Boundary Conditions

Now we are in principle done, but the model is still not perfect. If we try to simulate waves we see this very clearly, but we can also see this in electrostatics, see figure 2. Here we clearly see that the model does not produce the correct result, since the electric field close to the edges point in the wrong directions and waves gets reflected at the boundaries. Then is the model wrong? Should we just delete the simulator and give up? No, because when we think about it, it is in fact obvious why the model does not produce the predicted result. It is because the simulator is not simulating the system in vacuum. The problem is not inside the simulation but the boundaries. We assumed that the potential should vanish at the boundaries $A^\mu = 0$, but this is not the boundary conditions of vacuum. It is the boundary conditions for a grounded conducter. This means that we are actually simulating a point charge in a metal box which is very different from a point charge in vacuum, so of cause the theory and the simulation does not match up. To fix this we have to take a look at boundary conditions.

Open Boundaries

First we would like to make what we will call open boundary conditions. This would be to have the system embedded in a vacuum. But this is actually rather hard and I did not manage to make a working boundary condition for this yet, so it will have to be implemented later.

Flat Boundaries

Another very similar boundary condition is what we will call flat boundaries. The problem we want to solve here is: let's say we know the fields should be zero at the boundary but we don't know what the potential value is. This means that the potential is flat. An example of this could be the infinite plate capacitor where we define one side to have potential 0 then the other side will have some other unknown potential but still be flat. Let's assume we are at a boundary such that $A^\mu(\mathbf{x} + \Delta x_i \hat{\mathbf{x}}_\mathbf{i})$ is outside of our box. Now our goal is to estimate what this value should be. Since there are no charges or currents outside of our box we see from Eq. 29 that the laplacian is 0, in the electrostatics case. Because of this we expect that

$$\left.\frac{\partial A^\mu}{\partial x_i}\right|_{\mathbf{x}+\Delta x_i \hat{\mathbf{x}}_\mathbf{i}} \approx \left.\frac{\partial A^\mu}{\partial x_i}\right|_{\mathbf{x}} \tag{35}$$

This is actually rather easy to implement, we just have to redefine our laplacian matrix. To do this we will define the new laplacian matrix as

$$\mathbf{M}^\mu = \mathbf{M}_\mathbf{0}^\mu + \sum_{i\in\{x,y,z\}} \mathbf{B}_{\mathbf{f},\mathbf{i}}^\mu \tag{36}$$

Where $\mathbf{M}_\mathbf{0}^\mu$ is just the normal laplacian defined in Eq. 24 and $\mathbf{B}_{\mathbf{f},\mathbf{i}}^\mu$ is the boundary part for the $\hat{\mathbf{x}}_\mathbf{i}$ direction. This can be written as

$$\mathbf{B}_{\mathbf{f},\mathbf{i}}^\mu = \begin{cases} \frac{1}{(\Delta x_i)^2} & \text{for } (n_i = 0 \text{ or } n_i = N_i - 1) \text{ and } m = n \\ 0 & \text{else} \end{cases} \tag{37}$$

Here we should be careful because here we did not define any ground voltage so the simulation could add a constant voltage which would also solve the problem. This means that the solution may diverge, this can be solved by grounding the system somewhere at the boundaries. For the example with the capacitor we would combine this with a grounded (metal wall) boundary condition at one side to ground the system.

Periodic Boundaries

Another important boundary condition is the periodic boundary conditions. It does not have any real physical significance but is a very useful tool for modelling. This is for two different reasons. First let's assume that we try to build this simulator but with spherical coordinates, then we have some angle coordinates and of course when something like ϕ reaches 2π then it should be the same as 0, this is precisely periodic boundary conditions. Another reason for the importance is to efficiently model symmetric systems. Let us assume we want to model

the plate capacitor. One could build a large 100 x 100 x 100 grid and put int a plate capacitor, but this is not impressively large and it will take a long time. One could instead define periodic boundary conditions in the x and y directions and then build a 1 x 1 x 10000 grid. Since we know the system is invariant under translation in the x and y directions then the one grid point in the x and y directions can model the problem exactly. Then we just add a lot of points into the last direction. This is so much easier to solve that we can solve it exactly in around 1 ms. Now that we have seen why this boundary condition is important let us define the matrices:

$$\mathbf{B}^{\mu}_{\mathbf{p},\mathbf{i}} = \begin{cases} \frac{1}{(\Delta x_i)^2} & \text{for } (n_i = 0 \text{ and } m_i = n_i + N_i - 1) \text{ or } (n_i = N_i - 1 \text{ and } m_i = n_i - N_i + 1) \\ 0 & \text{else} \end{cases} \tag{38}$$

Again just like for the other boundary conditions we need to be careful, we cannot only have periodic boundary conditions since then we have not defined a zero point potential.

Closed Boundaries

Finally we have the closed boundary conditions. This is a boundary condition where we define what the potential is at the boundaries for every time t. We already used a special case of this when we defined the potential to be zero at the boundaries, but now we want to be more general. This way we could as an example define boundary conditions that oscillate and this way model plane waves. To add this we cannot just add a boundary matrix, this is because the value at the boundary does not depend on the values of the grid points linearly. Therefor we have to go back to Eq. 22. If we have any values of A^{μ} in our sum which is outside the grid (at the boundary) then we take it outside the sum and collect it in the vector called $\mathbf{C}^{\mu}(\mathbf{x}, t_0)$ since this can depend on time. Then we get

$$\begin{aligned} &A^{\mu}(\mathbf{x}) - \frac{1}{8}c^2(\Delta t)^2 \sum_{i\in\{x,y,z\}} \left(\frac{1}{(\Delta x_i)^2} \left(A^{\mu}(\mathbf{x} + \Delta x_i \hat{\mathbf{x}}_\mathbf{i}) + A^{\mu}(\mathbf{x} - \Delta x_i \hat{\mathbf{x}}_\mathbf{i}) - 2A^{\mu}\left(\mathbf{x}, t_0 + \frac{1}{2}\Delta t\right)\right)\right) \\ &- \frac{1}{8}c^2(\Delta t)^2 \mathbf{C}^{\mu}(\mathbf{x}, t_0) = R^{\mu}(\mathbf{x}, t_0) \end{aligned} \tag{39}$$

Rewriting this we get

$$\left(\mathbf{I} - \frac{1}{8}c^2(\Delta t)^2 \mathbf{M}^{\mu}\right)\mathbf{A}^{\mu} = \mathbf{R}^{\mu} + \frac{1}{8}c^2(\Delta t)^2 \mathbf{C}^{\mu}(\mathbf{x}, t_0) \tag{40}$$

Now to do a simulation we will just need to define $\mathbf{C}^{\mu}$ at all times and we can do any simulation.

2 Guide

Now we will just have a short chapter with a guide on how to get started simulating. This is not meant as an in depth tutorial on everything that you can do, but a guide on the most important features.

2.1 Setup

The first thing you should do when you want to do these simulations is to install the module with "pip install KUEM" and you should be good. Throughout this guide I will have imported it as:

```
import KUEM as EM
```

You can also check out the project at https://github.com/RasmusBruhn/KUEM

2.2 Boundary conditions

After importing the module you should define you system this is done by defining the boundary conditions and the charge- and current densities. First we will set up the boundary conditions.
We have 3 different directions that we should define the boundary conditions for. We do this by defining a 3 long list:

```
Boundaries = [Boundary_x, Boundary_y, Boundary_z]
```

Now we have to define the individual boundaries. If we want there to be periodic boundary conditions we just set

```
Boundary = "periodic"
```

otherwise the boundary should be a list with 2 elements, the conditions at the negative and positive boundary. These boundaries can be either "flat" or "closed". An example could be the infinite plate capacitor we we want the x and y direction to be periodic, the negative z-direction should be closed and the positive z-direction is flat:

```
Boundaries = ["periodic", "periodic", ["closed", "flat"]]
```

Now if you have any closed boundary conditions which should not be grounded to 0, like if you want to generate plane waves, then you should define a function to tell the simulation what the value of the potential should be at the different boundaries. The function should have the form:

```
def C(dx, N, x0, c, mu0):
    # Create an empty C
    C_Array = np.zeros(tuple(N) + (4, 3, 2), dtype = float)

    # Define the boundary here

    # Turn into vector
    C_Vector = EM.to_vector(C_Array, N)

    # Create the function to return the conditions
    # t:          The time
    def GetC(t):
```

```
        # Put some code here to modify C_Vector

        return C_Vector

    return GetC
```

The setup is quite simple but with a lot of notation, you should define a function, I call it C, which must take the parameters dx, N, x0, c and mu0. c and mu0 are just the values of c and mu0 used in the simulation, these are typically set to 1. x0 is a 3 element long numpy array with the lowest x, y and z values of the grid in the simulation. N is a 3 long numpy array with int dtype containing the number of grid points in the x, y and z direction. And dx is also a 3 long numpy array with the distance between grid points in the x, y and z directions. Usually you only need to use N and dx inside the function, but all the parameters still needs to be listed.

Inside the function you need to define another function, I call it GetC, which only takes one parameter, the time t. The function C must return the function GetC. Whenever the function GetC is called it must return the boundary conditions and the time t.

The boundary conditions is a numpy array of shape (V, 4, 3, 2) where V is the number of grid points (np.prod(N)). The first axis defines the grid point, the second axis is the μ label from Eq. 4. The third axis is the direction (0: x, 1: y or 2: z) and the last axis is 0 if it is the negative boundary and 1 for the positive. However this vector is not easy to work with so instead we can define the array version of it which should have shape (Nx, Ny, Nz, 4, 3, 2) where the last 3 axis are the same as for the vector but the first axis is expanded in the x, y and z components. To convert it back into a vector use the function EM.to_vector(C_Array, N). Now if you want to define the boundary at say the positive y direction, you should fill out the values of C_Array[:, -1, :, :, 1, 1]. Or for the negative x direction it is C_Array[0, :, :, 0, 0]. Then you fill out these values with the value of A^μ at the boundary. As an example we can make the boundary condition for a plane wave coming in from the negative z-direction. This means the negative z direction needs to be filled out with an oscillating value in the $\mu = y$ direction (The light is polarised in the y-direction):

```
Frequency = 1
Amplitude = 1

def C(dx, N, x0, c, mu0):
    # Create an empty C
    C_Array = np.zeros(tuple(N) + (4, 3, 2), dtype = float)

    # Add the plane wave
    C_Array[:, :, 0, 1, 2, 0] = Amplitude

    # Turn into vector
    C_Vector = EM.to_vector(C_Array, N)

    # Create the function to return the conditions
    # t:          The time
```

```
    def GetC(t):
        return C_Vector * np.cos(2 * np.pi * Frequency * t)

    return GetC
```

2.3 Charge- and current density

Next up is the most important part, here you have to define the system by define the charge- and current density of all space at all times. This is done in much the same way as with the boundary conditions. Here the function should be of the form:

```
def J(dx, N, x0, c, mu0):
    # Create J_Array
    J_Array = np.zeros(tuple(N) + (4,))

    # Define the currents here

    # Turn into a vector
    J_Vector = EM.to_vector(J_Array, N)

    # Return the vector
    def GetJ(t):

        # Calculate the J_Vector at time t

        return J_Vector

    return GetJ
```

Just like for the C function, now the J function takes the same input arguments. Inside the J function we also define a new function GetJ(t) which we return in the end. This function GetJ behaves the same as GetC, it returns the J_Vector at time t. The J_Vector has the shape (V, 4) where V is defined in the same way as before and again the first axis defines the position in the grid and the second axis defines the component of the J^μ vector from Eq. 4. Also just like for C we can first define the J_Array and then convert it into a vector afterwards.

An example of this could be for an oscillating line current. Here we have a wire going through the center of the xy-plane in the z-direction. Here we have to remember to divide the current by the area of one grid point $dx \cdot dy$ because we define the current density and not the current.

```
Current = 1

def J(dx, N, x0, c, mu0):
    # Create J_Array
    J_Array = np.zeros(tuple(N) + (4,))

    # Add in the current, normalising so the current is the same no matter the grid size
```

```
    J_Array[int(N[0] / 2), int(N[1] / 2), :, 3] = Current / (dx[0] * dx[1])

    # Turn into a vector
    J_Vector = EM.to_vector(J_Array, N)

    # Return the vector
    def get_J(t):
        return J_Vector * np.sin(2 * np.pi * Frequency * t)

    return get_J
```

2.4 Creating simulation

Now that you have defined the boundary conditions and the charge- and current density, we can set up the simulation class. This is done with:

```
EM.sim(N, delta_x = np.array([1, 1, 1]), x0 = np.array([0, 0, 0]), t0 = 0, dt = 1, c = 1, mu0 = 1, approx_n = 0.1, dyn_n
    = 10, J = default_J, C = default_C, boundaries = [["closed", "closed"], ["closed", "closed"], ["closed", "closed"]]) :
```

N is the only variable which must be given. This is again a 3 long numpy array of int dtype giving the number of grid points in each direction. The larger the values of N the more accurate the simulation, but it will also get much slower. Here it is worth noting that if you have periodic boundary conditions along some axis, then you need only 1 grid point in that direction and then the value of x0 and delta_x in that direction does not matter anymore. delta_x is also a 3 long numpy array, this contains the length of each of the box your simulation is inside. x0 is a numpy array with the lowest x, y and z values of the box.

t0 is the starting value of time, this will usually just be set to 0. dt is the increment in time per timestep. c is the value of the speed of light and mu0 the value of μ_0, this will typically both be set to 1 to keep numbers in the simulation around unity.

approx_n and dyn_n are variables used for determining the accuracy of the simulation if it is not solved exactly, more about this later. Mostly this should just stay at the default values.

J should be set equal to you charge/current density function (the J function) and C equal to the C function created earlier. If J is not given then we will have $J^\mu = 0$ everywhere and if C is not given then all closed boundaries are grounded to 0, that is $A^\mu = 0$ at all the closed boundaries. boundaries should be set equal to the Boundaries list created earlier.

2.5 Samplers

Now you are almost done, in principle you could just tell the sim class to simulate the system. There is one problem however, we still need to plot it. To do this the samplers can help. You set up a number of samplers before you simulate and then during the simulation the samplers will automatically gather data. Then after simulating you can just tell the samplers to plot the data or create a video. There are a lot of different samplers which can sample different things, and you can define your own but they are all split into 4 different types of samplers (number, line, scalar, vector), you can read more about the individual samplers in the documentation.

The main difference between the different samplers are just how they can plot the data afterwards. The number samplers gather one value at each time step which they can plot as a function of time at the end. The rest of the samplers can either plot the data for some specific time or create a video. The line sampler gathers a one-dimensional array of data each time step. This can be plotted as a normal plot (plt.plot). The vector and scalar samplers both collect 2-dimensional data and the vector sampler gathers 2 sets of 2-dimensional data (vx and vy). The scalar sampler can then plot a heatmap (plt.imshow) of the data and/or a contour plot. The vector sampler can plot a vector field and/or field lines.

That is enough information about the samplers them selves. To set up the samplers you first need to define at which points in the grid they should sample. For the number sampler this is not necessary but for the line sampler a 2D array of points should be given where the first axis is only 3 long and determines the x, y and z coordinate of the point. If you just want a straight line of points you can use the function:

```
EM.sample_points_line(x1, x2, Resolution)
```

which creates a line of points (the number of points is Resolution) evenly distributed between the points x1 and x2. For the 2D samplers a 3D numpy array of points needs to be given, again the first axis is 3 long and determines the x, y and z component. To sample from a plane use the function:

```
EM.sample_points_plane(x_hat, y_hat, x_c, Size, Resolution)
```

x_hat and y_hat are 3 long numpy arrays defining unit vectors for the x- and the y-direction in the plane. x_c is also a 3 long numpy array defining the center of the plane. Size is a 2 long list or numpy array with the width and height of the plane (length in x- and y-direction). Resolution is also a 2 long list with the number of points along the x and the y direction.
An example of this could be to sample the xy-plane at z=0 and the x-axis from 0 out to the edge of the box:

```
# Define hat vectors
x_hat = np.array([1, 0, 0])
y_hat = np.array([0, 1, 0])
hat = np.array([0, 0, 1])
B_hat = np.array([0, −1, 0])

# Define the resolutions
Res_scalar = 1000
Res_vector = 30
Res_line = 1000

# Define extents
PointsSize = np.array([delta_x[0], delta_x[1]])
x_vals = np.linspace(0, delta_x[0] / 2, Res_line)

# Get grid points
Points_scalar = EM.sample_points_plane(x_hat, y_hat, np.array([0, 0, 0]), PointsSize, np.array([Res_scalar, Res_scalar]))
```

```
Points_vector = EM.sample_points_plane(x_hat, y_hat, np.array([0, 0, 0]), PointsSize, np.array([Res_vector, Res_vector]))
Points_line = EM.sample_points_line(np.array([0, 0, 0]), np.array([delta_x[0] / 2, 0, 0]), Res_line)
```

Finally we define the samplers by giving them the simulation they should sample from, the points at which they should sample and for the vector samplers we also need to give it 2 unit vectors defining what components of the vectors should be sampled. For the other samplers we may also need to give it one unit vector if it needs to sample from a vector field. To continue the last example we could do:

```
Sampler_B_2D = EM.sampler_B_vector(Sim, Points_vector, x_hat, y_hat)
Sampler_A_2D = EM.sampler_A_scalar(Sim, Points_scalar, hat = hat)
Sampler_B_1D = EM.sampler_B_line(Sim, Points_line, x = x_vals, hat = B_hat)
Sampler_A_1D = EM.sampler_A_line(Sim, Points_line, x = x_vals, hat = hat)
```

2.6 Simulating

The next step is really easy, now we have to simulate the system. First we find the starting conditions, this is done with:

```
Sim.solve(exact = False, progress = False)
```

Where Sim is the object returned by EM.sim(). If exact is False then it will approximate a solution to the simulation and the precision is determined by approx_n. If it is True then it will solve the simulation exactly. This does not mean that it will solve the system exactly but that the linear set of equations the simulation solves will be solved exactly. progress can either be False or some number. If it is False nothing happens but if it is some number then in intervals (in seconds) of that number, it will print out an estimate of the time remaining, this only works if exact is False.

This is it if we only have an electrostatics problem. If however we have an electrodynamics problem we also need to solve the dynamics. This is done with:

```
Sim.dynamics(Count, SubCount, exact = False, progress = False)
```

Here exact and progress is the exact same except that progress also works when exact is True. Count is the number of time steps to sample and SubCount is the number of time steps per sample, so as an example if Count was 100 and SubCount was 5 then it would take 500 steps and sample every 5th step in the samplers.

2.7 Plotting

The last thing we need to do is to plot and/or make videos. All samplers have the methods plot() and make_video(). The details of the inputs of the methods are not too interesting, you just have to play around with them to find some settings that you think look good. One thing to remember is that the file name of a video should have a video extension like .avi.