Pomona College

A thesis submited in partial fulfillment for the degree of Bachelor of Arts

in

Physics and Astronomy

Modeling the Evolution of Surface Charge on a Wire-Capacitor Circuit

Author:

Will Buchholtz

Advisors:

Dr. Tom Moore

Dr. Alma Zook

Contents

1 Introduction

1.1 Computational Problem

In this senior thesis, I compute the surface charge distribution that appears on a wire connected to a discharging capacitor. To do this, I have written a Python routine which uses an FDTD (Finite Difference Time Domain) algorithm to solve Maxwell's equations. The routine takes as input the shape of any wire which can be drawn in the xy-plane, provided that it begins and ends at set capacitor plates. This outline is then stacked along the z-axis to create a three-dimensional circuit. Using this shape and an initial charge for the capacitor plates, the program calculates the surface charge on the top of the wire at any future moment.

As it turns out, the charge distribution on the surface of a wire is very tricky to model. Although the electromagnetic theory behind the phenomenon is well understood, no analytic expressions exist to describe its evolution and so one must use numerical methods to generate results. Ultimately, one faces the difficult and computationally expensive task of simultaneously solving all four of Maxwell's equations for each point in space. Since each of Maxwell's equations are vector equations this task, in the most complete form, requires solving *twelve* coupled differential equations. The inherent difficulty of this calculation is that the surface charge at a position on the wire depends on the electric and magnetic fields found there but finding these fields requires in turn solving Maxwell's equations for all other points in space. This problem therefore links local and collective quantities inextricably. Generally speaking, one must know, from the outset, the solution to the problem one is trying to solve.

This thesis attempts to build on past work by producing more accurate results. Although methods have previously been developed to approximate the surface charge distributions on wires, particularly [11] and [12], none have been able to fully solve Maxwell's equations for an arbitrary circuit, which I attempt to do here. As further discussed, the results from this thesis could be a useful learning aid when teaching circuits in a high school or undergraduate physics course.

Figure 1: *A length of copper wire - the main material appearing in this system. Image obtained from [5].*

1.2 Motivation

The phenomenon of charge appearing on the surface of current carrying wires is not new physics. It is perhaps because of this and the lack of a related application that the topic has not been extensively explored. Nonetheless, there are reasons as to why it is worth studying and why the calculations in this thesis, even though restricted to simple wire-capacitor circuits, are of value to the scientific community. The most general is that since wires are so ubiquitous and integral to modern society, we should be scientifically rigorous in understanding their basic behavior. Studying the surface charge on wires *is* a scientific luxury but it does fill a remarkable gap in our knowledge. Given that the evolution of surface charge is an exotic process, modeling this system is also worthwhile if only because its complexity is intriguing.

Finally, there is educational utility. The surface charge on wires is not often discussed in introductory physics courses so the plots produced could be of unique use when teaching the physics of circuits, perhaps here at Pomona College. Since surface charge is crucial in allowing a wire to conduct current, it is important to explain its existence in order to accurately describe the physics behind currents. As well, I use standard techniques to solve the problem and so this thesis could function as an effective and instructive example of computational physics in practice.

1.3 Existing Literature

Part of the reason for choosing this thesis is that there is only a small literature on the topic. However, it appears that there has been a slight increase of interest in modeling surface charge in recent time. Over the last 20 years a number of papers have been published that either present simulations of specific distributions or suggest general approximation methods. Several of these papers can be found cited in the list of references.

Two of the more substantial publications include [11] by Mueller and [12] by Preyer. The paper by Mueller presents a novel graphical method for approximating simple distributions and uses this technique to model a number of wire configurations. There is a useful list of six steps for solving for the distribution in two dimensions, which one can use as a first approximation. The major drawback of this approach, however, is that one must solve for the distribution manually. This can be a very tedious task, especially with complicated wire configurations.

The paper by Preyer, meanwhile, employs a numerical approach. The evolution of surface charge is taken to be semi-static and no radiation effects are considered. Again, several calculations are given, this time for simple resistor-capacitor circuits. Four circuits are modeled using different wire resistance types, namely a, "uniform resistive wire", "lumped resistor", "narrow wire", and a "sinuous wire". Ultimately, my work attempts to build substantially on this paper by directly solving Maxwell's equations and by allowing for the treatment of an arbitrarily shaped wire.

2 Physical Theory

2.1 Preliminaries

Although it is assumed that the reader has some foundational knowledge of Electricty and Magnetism the following sections provide a review and discussion of the basic principles. In particular, a discussion of why surface charges must necessarily exist is included.

2.1.1 Maxwell's Equations

The physical theory used by this thesis is that of classical electrodynamics. Classical electrodynamics, as it turns out, is completely described by a combination of the Lorenz force law

$$\vec{F} = q(\vec{E} + \vec{v} \times \vec{B}) \tag{1}$$

and Maxwell's celebrated equations

$$\vec{\nabla} \cdot \vec{E} = \frac{\rho}{\epsilon_0} \tag{2}$$

$$\vec{\nabla} \cdot \vec{B} = 0 \tag{3}$$

$$\vec{\nabla} \times \vec{E} = -\frac{\partial \vec{B}}{\partial t} \tag{4}$$

$$\vec{\nabla} \times \vec{B} = \mu_0 \vec{J} + \mu_0 \epsilon_0 \frac{\partial \vec{E}}{\partial t} \tag{5}$$

The Lorenz force law tells us how to calculate the force acting on a charge moving through electric and magnetic fields. Maxwell's equations, on the other hand, establish constraints for these fields and explain how they may arise. The first two of Maxwell's equations, referred to respectively as Gauss's law and Gauss's law for

magnetism [1], describe the divergence of the fields. The third equation, Faraday's law, and the last, Ampere's Law, then describe the curl. According to Helmholtz's theorem [2], a vector field is completely determined by its divergence and curl, meaning that Maxwell's equations provide all the necessary information to construct the electric and magnetic fields from their sources. It is also important to note that these relations are *local* field equations. This means that they describe the electric and magnetic fields in terms of charge and fields existing in the *same* region of space, thereby avoiding the issue of instantaneous action at a distance.

In this thesis, as per usual convention, $\vec{E}$ and $\vec{B}$ are always the electric and magnetic fields respectively and $\vec{J}$ is the current density. The parameters ϵ_0 and μ_0 are the permittivity and permeability of free space. These constants can applied to matter in the case of linear dielectrics via the transformations

$$\epsilon = \epsilon_0(1 + \chi_e)$$

$$\mu = \mu_0(1 + \chi_m)$$

where χ_e and χ_m are the electric and magnetic susceptibility respectively. χ_e and χ_m are important characterizing parameters for a material and, in general, determine how easily a material permits electric and magnetic dipoles. Quite similarly

$$\epsilon_r = \frac{\epsilon}{\epsilon_0}$$

$$\mu_r = \frac{\mu}{\mu_0}$$

are the dielectric constant and relative permeability and are often used to indicate the permittivity and permeability of a material.

The gradient operator, $\vec{\nabla}$, also plays a significant part in Maxwell's equations and is defined in Cartesian coordinates as

[1] Some authors, such as [8], consider equation (3) to be unnamed.
[2] For a thorough treatment see [6].

$$\vec{\nabla} = \left\langle \frac{\partial}{\partial x}, \ \frac{\partial}{\partial y}, \ \frac{\partial}{\partial z} \right\rangle$$

When the gradient is applied to a vector via a dot product the resulting scalar quantity, the divergence of the vector, is a measure of the net "outwardness" or "inwardness" of the field across a surface. If, instead, the gradient operator is applied to a vector via a cross product, the resulting vector quantity, the curl of the vector, is a measure of how curved the vector field is in a region of space. Maxwell's equations include one equation for the curl and divergence of both the electric and magnetic fields. Together, the equations for divergence and curl link the two fields and, along with the boundary conditions imposed on them, completely determine both.

2.1.2 Deriving Coulomb's Law

For a beginning physics student, Coulomb's law is the foundation of electrodynamics. However, it is important to note that Coulomb's law follows directly from Gauss's law. The derivation is a useful illustration of how Maxwell's equations are the true cornerstone of electromagnetism and proves to be helpful when examining the computational method explained by Preyer in [12].

Now, in integral form, Gauss's law reads

$$\oint \vec{E} \cdot d\vec{r} = \frac{Q_{enc}}{\epsilon_0}$$

In the case of a point charge, q, the charge enclosed in a centered Gaussian sphere is $Q_{enc} = q$. As well, $\vec{E}$ points radially and since there is no special direction arising from this charge distribution $|E|$ must be spherically symmetric. Therefore, $\vec{E} \cdot d\vec{r} = |E|dr$ meaning

$$\oint |E| dr = \frac{q}{\epsilon_0}$$

$$\Rightarrow |E| \oint dr = \frac{q}{\epsilon_0}$$

And, because of the chosen Gaussian surface, the surface integral will simply evaluate to the surface area of a sphere of radius r.

$$\Rightarrow |E|(4\pi r^2) = \frac{q}{\epsilon_0}$$

$$\Rightarrow \vec{E} = \frac{1}{4\pi\epsilon_0}\frac{q}{r^2}\hat{r} \tag{6}$$

which is Coulomb's law. Therefore, as is the case in [12] any computation that makes use of Coulomb's law is ultimately making use of Gauss's law.

2.2 Existence of Charge Distributions on Wires

Conceptually, it is simple to argue that surface charge on wires must necessarily exist. As noted in [11], the Drude model can be used to reach this conclusion. A useful discussion of the topic is given in [9], which is reproduced here.

The Drude model concerns electron transport and was proposed in 1900 by Paul Drude, remarkably only three years after J.J. Thompson discovered the electron in 1897. The main idea of the model is that only the outermost electrons of an atom are free to wander through a metal. These valence electrons therefore become conduction electrons and are what create the flow of charge known as current. As described in [9], the mean drift velocity of charge along a particular direction, in this example the x axis, can be described by

$$\frac{d}{dt}\langle v_x \rangle = \frac{q}{m}E_x - \frac{1}{\tau}\langle v_x \rangle$$

Note that τ is the mean free time, in other words, the time between collisions. When the mean free velocity is constant, its time derivative is zero leaving

$$\langle v_x \rangle = \frac{q\tau}{m}E_x$$

Now the current density J_x, the net flux of of n charge carriers per time, is

$$J_x = nq\langle v_x \rangle$$

Therefore, combining the two relations gives

$$J_x = n\frac{q^2\tau}{m}E_x$$

And since the conductivity of a material is given by the expression

$$\sigma = \frac{nq^2\tau}{m}$$

Subbing this in gives the following expression for current density

$$J_x = \sigma E_x$$

which can then be generalized as the vector equation

$$\vec{J} = \sigma\vec{E} \tag{7}$$

This equation connects current density $\vec{J}$ with electric field $\vec{E}$ via the conductivity of the metal σ. Although electrons typically undergo a certain amount of random motion within a metal, this equation asserts that there will be no *net* motion, i.e. current, unless there is an applied electric field across the material.

From this fact alone, one might intuit that surface charge must necessarily exist. A more rigorous argument, however, can be made by demonstrating that no net charge can exist on the inside of a wire. One proof using this approach can be found in [10]. If net charge did exist on the inside of a wire then this would create a non-uniform internal field. The net charge could perhaps be arranged so that the resulting electric field vanished at certain points, but no configuration could avoid a variation in field. However, a non-uniform field would cause an ill-defined potential.

Figure 2 depicts a vector field whose curl is nonzero and is therefore nonconservative. For such a field, a line integral evaluated between arbitrary points A and B

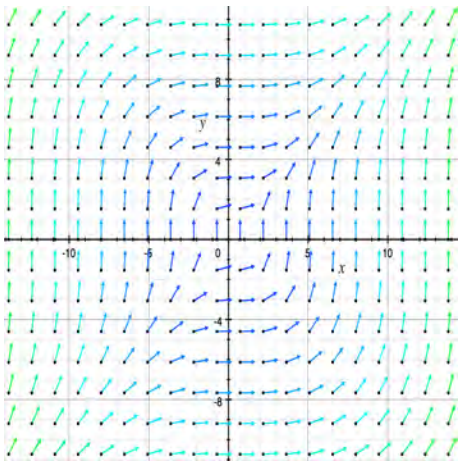

Figure 2: *Plot of the non-conservative vector field* $\vec{F} = \langle y^2, x^2 \rangle$.

might give a different value depending on the path taken. Now in an electric field the potential between two points A and B is the line integral of the field:

$$V(B) - V(A) = \oint_A^B \vec{E} \cdot d\vec{r}$$

This means if $\vec{E}$ is not conservative then the potential between two points will be path dependent and so not well defined. Therefore, since no net charge can exist on the inside, the only place that could allow for the necessary net charge is the surface of the wire.

It should be noted that the terminals of the battery driving the circuit, because they are charged, *do* create an electric field on the inside of the wire. However, since this field only points directly outward from the terminals, for any curved wire it does not have significant magnitude along the wire's axis for much of the circuit. Therefore, there is a clear need for the field created by the surface charge.

2.3 Solving for the Surface Charge Distribution

When solving for the surface charge distribution of a wire there are surprisingly few starting parameters. The key information that is required is the shape of the

wire and the initial charge on the capacitor plates. Additionally, it is necessary to know the conductivity σ of the wire as well as of the surrounding medium.

Ultimately, what we wish to solve for is charge density. Fortunately, Maxwell's equations do relate the electric and magnetic fields directly to charge, as can be seen by the charge density term ρ in Gauss's law

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

and the current density term $\vec{J}$ in Ampere's laws

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

However, Faraday's law also contains an electric field term

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

and since we have introduced magnetic fields into the problem it is necessary to also solve Gauss's law for magnetism

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

In order to solve for the charge distribution of the wire we must therefore simultaneously solve all *four* of Maxwell's equations. This means we must solve a system of four coupled partial differential equations, which, needless to say, is a very non-trivial task. Ultimately, this will give us the electric and magnetic fields from which we can recover the charge density at any point using Gauss's law. Ideally, we would like to find closed form expressions for the fields, $E(x, y, z)$ and $B(x, y, z)$, perhaps in "wire" coordinates r, ϕ and s where r and ϕ are a cross-sectional radius and angle and s is the distance along the wire. However, given the complexity of this system of differential equations, no analytic methods exist to extract such a function and, in fact, no closed form solution exists at all. Instead, we are forced to use computational methods to approximate the solution at each future instant.

3 Computational Approach

3.1 The Model

Physics at every level is fundamentally a model. Even if one were to program every physical law into a simulation the resulting calculation would still be an approximation. In practice, when running a calculation one must make decisions about which effects to account for and, in some cases, how best to evolve a system so that it behaves physically. In this thesis, I attempt to make my model as complete as possible by fully solving Maxwell's equations. Beyond this decision, my task as a physicist is to decide how to implement my model through code.

3.1.1 The Independent Physical Quantities

The physical system under consideration is a current carrying wire attached to a discharging capacitor. The physical quantities that determine its evolution are, predictably, the electric field, the magnetic field, the shape of the circuit, the charge following through the wire and the surface charge that results. Computing all of these quantities presents itself as a daunting task. Fortunately, though, the magnitude of this problem can be reduced by recognizing that some of these quantities are dependent on the others. Firstly, as can be seen from Gauss's law, knowing the electric field at a point is sufficient to determine the charge there.

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

$$\Rightarrow \rho(x, y, z) = \epsilon_0 \left(\frac{\partial}{\partial x} E_x + \frac{\partial}{\partial y} E_y + \frac{\partial}{\partial z} E_z \right)$$

This therefore enables one to calculate the surface charge at a point on the wire simply by calculating the divergence of the electric field there and multiplying by ϵ_0, the permittivity of free space. On a similar note, equation (7) from above says

$$\vec{J} = \sigma \vec{E}$$

Since $\vec{J}$ is the current density then

$$|\vec{J}| = \frac{I}{A}$$

where A is the cross-sectional area of the wire. That is

$$\frac{I}{A} = \sigma|\vec{E}|$$

$$\Rightarrow \qquad I = A\sigma|\vec{E}|$$

This result shows how the current, much like the charge, can be calculated at a point in space assuming the field there is known, with the additional restriction that one must also know the conductivity σ. The notation in this case is very unfortunate. σ is often the symbol used to indicated surface charge, and since σ is multiplied by A in this expression it is tempting to assume that $\sigma A = Q$, the charge appearing on some cross section of the wire. Because of this confusion I am careful to note the distinction between surface charge and conductivity when the ambiguity arises.

Of all the physical quantities listed above the shape of the circuit is the only one that remains static. At first, there may be some confusion as to how one can incorporate the *shape* of the circuit into some equation or a program for that matter. This is cleverly done by indicating the conductivity at each point in space, a quantity built into the physical theory by equation (7). Low or no conductivity corresponds to empty space while high conductivity implies the presence of the wire or a circuit element. In my program, part of the wire's shape is assumed and the rest is initialized by the user. Since the conductivity appears as a parameter and is permanently determined this means the only independent quantities, the ones which must be actively computed, are the electric and magnetic fields. The evolution of these fields, as discussed above, can be completely determined by Maxwell's equations. Although determining the evolution of this system amounts to determining the evolution of the electric and magnetic fields, the resulting computation remains quite challenging.

3.1.2 Pixelating Space

In order to model this system I make the fundamental assumption that the continuum of space can be accurately approximated by a fine enough pixelation of points. This allows me to define three dimensional arrays for the above physical quantities in which each trio of indices indicates a point in space and the corresponding array entry is the value found there. Arguably the single most important parameter in my program is the size of these arrays. Their size defines the degree spatial pixelation and determines how long the simulation must run and how accurate the results are. Another quantity of importance is the initial charge that appears on the capacitors. This value affects how much current will ultimately flow. As I experienced in my simulations, although the amount of current flowing does not affect the shape of the distribution, it does affect the amount of charge that appears in it.

3.1.3 Discretizing Gauss's Law

Another essential concept employed in this thesis is the idea that differentials can be approximated by sufficiently small differences. In some sense, the entire computational side of this project is an elaborate application of this one principle. As shown above, Gauss's law can be used to find the charge density at a point in space, assuming the electric field is known there. Replacing partials with differentials in

$$\rho(x,y,z) = \epsilon_0 \left(\frac{\partial}{\partial x} E_x + \frac{\partial}{\partial y} E_y + \frac{\partial}{\partial z} E_z \right)$$

gives

$$\rho(x,y,z) = \epsilon_0 \left(\frac{\Delta}{\Delta x} E_x + \frac{\Delta}{\Delta y} E_y + \frac{\Delta}{\Delta z} E_z \right)$$

which becomes

$$\rho(x,y,z) =$$

$$\epsilon_0 \left(\frac{E_x(x+\delta,y,z) - E_x(x-\delta,y,z)}{2\delta} + \frac{E_y(x,y+\delta,z) - E_x(x,y-\delta,z)}{2\delta} + \frac{E_z(x,y,z+\delta) - E_x + (x,y,z-\delta)}{2\delta} \right)$$

In my code the electric field is represented as a three dimensional array. Therefore, $E_x(x+\delta, y, z)$ and $E_x(x-\delta, y, z)$, for instance, correspond to the array entries on either side of the entry for $E_x(x, y, z)$ along the index representing the x direction. Finally, δ is then a parameter set to be the assumed distance between each array element. As can be seen, once the electric field is known, calculating the charge density at each point in space is surprisingly trivial.

3.1.4 Discretizing Faraday's and Ampere's Laws

Faradays' law and Ampere's law are the two Maxwell equations which relate the electric and magnetic fields. Therefore the discretized forms of these equations define the principle update equations that drive the evolution of the wire system. Consider, first, Faradays's law. One begins by breaking the vector equation into three scalar equations. That is

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

reads fully as

$$\Big\langle \frac{\partial}{\partial y}E_z - \frac{\partial}{\partial z}Ey,\ \ \frac{\partial}{\partial z}E_x - \frac{\partial}{\partial x}E_z,\ \ \frac{\partial}{\partial x}E_y - \frac{\partial}{\partial y}Ex \Big\rangle = -\Big\langle \frac{\partial}{\partial t}B_x\ \ , \frac{\partial}{\partial t}B_y\ \ , \frac{\partial}{\partial t}B_z \Big\rangle$$

which means that

$$\frac{\partial}{\partial y}E_z - \frac{\partial}{\partial z}Ey = -\frac{\partial}{\partial t}B_x$$

$$\frac{\partial}{\partial z}E_x - \frac{\partial}{\partial x}Ez = -\frac{\partial}{\partial t}B_y$$

$$\frac{\partial}{\partial y}E_x - \frac{\partial}{\partial y}Ey = -\frac{\partial}{\partial t}B_z$$

Next, one replaces the partial derivatives with differences. This is a bold ap-

proximation. However, when the size of the arrays used for the calculation becomes the approximation becomes more and more accurate.

$$\frac{\Delta E_z}{\Delta y} - \frac{\Delta E_y}{\Delta z} = -\frac{\Delta B_x}{\Delta t}$$

$$\frac{\Delta E_x}{\Delta z} - \frac{\Delta E_z}{\Delta x} = -\frac{\Delta B_y}{\Delta t}$$

$$\frac{\Delta E_x}{\Delta y} - \frac{\Delta E_y}{\Delta y} = -\frac{\Delta B_z}{\Delta t}$$

Multiplying through by $-\Delta t$ gives

$$-\frac{\Delta t}{\Delta y}\Delta E_z + \frac{\Delta t}{\Delta z}\Delta E_y = \Delta B_x$$

$$-\frac{\Delta t}{\Delta z}\Delta E_x + \frac{\Delta t}{\Delta x}\Delta E_z = \Delta B_y$$

$$-\frac{\Delta t}{\Delta y}\Delta E_x + \frac{\Delta t}{\Delta y}\Delta E_y = \Delta B_z$$

In each of these equations there is a ratio of a time difference and a spatial difference. As explained in [13], this ratio is known as the Courant number and is a parameter one will have to define when writing code to model Faraday's and Ampere's laws. Arbitrarily setting the Courant number to 1 allows for the simplest algebra and gives

$$-\Delta E_z + \Delta E_y = \Delta B_x$$

$$-\Delta E_x + \Delta E_z = \Delta B_y$$

$$-\Delta E_x + \Delta E_y = \Delta B_z$$

Using the subscripts i and f to indicate initial and final fields we can break apart the deltas. Isolating the final magnetic fields then yields the update equations:

$$B_{x,f} = B_{x,i} - (E_{z,f} - E_{z,i}) + (E_{y,f} - E_{y,i}) \tag{8}$$

$$B_{y,f} = B_{y,i} - (E_{x,f} - E_{x,i}) + (E_{z,f} - E_{z,i}) \tag{9}$$

$$B_{z,f} = B_{z,i} - (E_{y,f} - E_{y,i}) + (E_{x,f} - E_{x,i}) \tag{10}$$

A similar approach can be applied to Ampere's law.

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

Written out fully, this equation has the Cartesian form

$$\Big\langle \frac{\partial}{\partial y} B_z - \frac{\partial}{\partial z} B_y, \;\; \frac{\partial}{\partial z} B_x - \frac{\partial}{\partial x} B_z, \;\; \frac{\partial}{\partial x} B_y - \frac{\partial}{\partial y} B_x \Big\rangle$$
$$= \mu_0 \Big\langle J_x \;\; , J_y \;\; , J_z \Big\rangle + \mu_0 \epsilon_0 \Big\langle \frac{\partial}{\partial t} E_x, \;\; \frac{\partial}{\partial t} E_y, \;\; \frac{\partial}{\partial t} E_z \Big\rangle$$

This then defines the scalar equations

$$\frac{\partial}{\partial y} B_z - \frac{\partial}{\partial z} B_y = \mu_0 J_x + \mu_0 \epsilon_0 \frac{\partial}{\partial t} E_x$$

$$\frac{\partial}{\partial z} B_x - \frac{\partial}{\partial x} B_z = \mu_0 J_y + \mu_0 \epsilon_0 \frac{\partial}{\partial t} E_y$$

$$\frac{\partial}{\partial x} B_y - \frac{\partial}{\partial y} B_x = \mu_0 J_z + \mu_0 \epsilon_0 \frac{\partial}{\partial t} E_z$$

Replacing the partials with finite differentials and multiplying through by Δt gives

$$\Delta t\ (\mu_0 J_x) + \mu_0\epsilon_0\ \Delta E_x = \frac{\Delta t}{\Delta y}\Delta B_z - \frac{\Delta t}{\Delta z}\Delta B_y$$

$$\Delta t\ (\mu_0 J_y) + \mu_0\epsilon_0\ \Delta E_y = \frac{\Delta t}{\Delta z}\Delta B_x - \frac{\Delta t}{\Delta x}\Delta B_z$$

$$\Delta t\ (\mu_0 J_z) + \mu_0\epsilon_0\ \Delta E_z = \frac{\Delta t}{\Delta x}\Delta B_y - \frac{\Delta t}{\Delta y}\Delta B_x$$

Moving over the current density term, dividing by $\mu_0\epsilon_0$ and setting the Courant number to 1 again the yields

$$\Delta E_x = \frac{1}{\mu_0\epsilon_0}(\Delta B_z - \Delta B_y) - \frac{1}{\epsilon_0}\Delta t\ (J_x)$$

$$\Delta E_y = \frac{1}{\mu_0\epsilon_0}(\Delta B_x - \Delta B_z) - \frac{1}{\epsilon_0}\Delta t\ (J_x)$$

$$\Delta E_z = \frac{1}{\mu_0\epsilon_0}(\Delta B_y - \Delta B_x) - \frac{1}{\epsilon_0}\Delta t\ (J_x)$$

In addition, we can set the time step Δt to be 1. When writing the update equations in code time will simply be an index on a *for* loop that will be incremented by one. Finally, we can break apart the deltas and solve for the final electric field. Note that the subscripts i and f indicated initial and final and the bracketed $i, j, k's$ indicate field positions in the x, y and z directions respectively. This produces the update equations

$$E_{x,f} = E_{x,i} - \frac{1}{\mu_0\epsilon_0}((B_z[k+1] - B_z[k]) - (B_y[j+1] - B_y[j])) - \frac{1}{\epsilon_0}\ (J_x)$$

$$E_{y,f} = E_{y,i} - \frac{1}{\mu_0\epsilon_0}((B_x[i+1] - B_x[i]) - (B_z[k+1] - B_z[k])) - \frac{1}{\epsilon_0}\ (J_y)$$

$$E_{z,f} = E_{z,i} - \frac{1}{\mu_0 \epsilon_0}((B_y[j+1] - B_y[j]) - (B_x[i+1] - B_x[i])) - \frac{1}{\epsilon_0}\,(J_z)$$

In this case there remain the current density terms J_x, J_y and J_z. Fortunately, these can be easily eliminated using the important equation

$$\vec{J} = \sigma \vec{E}$$

This field corresponds to the initial field which means we can then write

$$E_{x,f} = E_{x,i} - \frac{1}{\mu_0 \epsilon_0}((B_z[k+1] - B_z[k]) - (B_y[j+1] - B_y[j])) - \frac{\sigma}{\epsilon_0}\,E_{x,i}$$

$$E_{y,f} = E_{y,i} - \frac{1}{\mu_0 \epsilon_0}((B_x[i+1] - B_x[i]) - (B_z[k+1] - B_z[k])) - \frac{\sigma}{\epsilon_0}\,E_{y,i}$$

$$E_{z,f} = E_{z,i} - \frac{1}{\mu_0 \epsilon_0}((B_y[j+1] - B_y[j]) - (B_x[i+1] - B_x[i])) - \frac{\sigma}{\epsilon_0}\,E_{z,i}$$

which simplifies to

$$E_{x,f} = E_{x,i}(1 - \frac{\sigma}{\epsilon_0}) + \frac{1}{\mu_0 \epsilon_0}((B_z[k+1] - B_z[k]) - (B_y[j+1] - B_y[j])) \qquad (11)$$

$$E_{y,f} = E_{y,i}(1 - \frac{\sigma}{\epsilon_0}) + \frac{1}{\mu_0 \epsilon_0}((B_x[i+1] - B_x[i]) - (B_z[k+1] - B_z[k])) \qquad (12)$$

$$E_{z,f} = E_{z,i}(1 - \frac{\sigma}{\epsilon_0}) + \frac{1}{\mu_0 \epsilon_0}((B_y[j+1] - B_y[j]) - (B_x[i+1] - B_x[i])) \qquad (13)$$

Note also that the conductivity term σ is a function of x, y, z. That is, its value varies as we iterate through space.

3.2 Alternative Approaches

3.2.1 Semi-Quantitative Model

As noted in [11] by Mueller, computing the surface charge on a wire is a difficult task except for the simplest geometries. However, as he proceeds to demonstrate there are relatively easy ways to *approximate* this distribution. His method is broken into six steps.

First, one must determine the current and potential at each conducting element in the circuit (wires, resistors, capacitors ect.) Next, one must be able to mark equal potential differences along the circuit, perhaps in $2V$ increments or other appropriate divisions. Thirdly, one marks an equal number of starting points between the poles of the battery or capacitor.

Using these evenly spaced potential values one can then construct equipotential curves from the starting points between the battery to the points along the circuit, keeping a few rules in mind. That is, none of the equipotential curves may cross each other, equipotential curves only cross conductors at the points determined in step two, and the curves will appear in the most spaced out configuration possible (as if the curves repel each other.)

Once the equipotential lines are drawn most of the tedious work is done. At this point, one can determine the magnitude of the surface charge at each equipotential point using the equation

$$\sigma = \epsilon_0 \frac{I\rho}{A} \tan\alpha$$

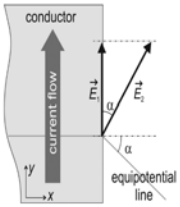

Figure 3: *Figure from [11] illustrating the kink angle α.*

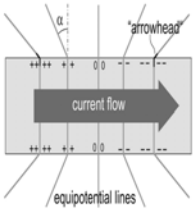

Figure 4: *This figure from [11] helps one assign the sign of surface charges. This is determined by the orientation of the kink angle relative to the direction of the current.*

In this expression, I is the current, ρ is the charge density at that point, A is the cross-sectional area and α is the angle between the incident equipotential line and the normal to the wire as shown in Figure 3. The sign of this charge can then be calculated using Figure 4. Lastly, one can determine the surface charge between two circuit elements with different resistivities using the equation

$$\sigma = \epsilon_0 \frac{I}{A}(\rho_2 - \rho_1)$$

where ρ_1 and ρ_2 are the charge densities in the two different circuit elements.

The accuracy of this method, ultimately, is determined by how many equipotential lines one is able (or willing) to draw. In practice one might not wish to use the technique extensively. Nonetheless, it is a useful approximation and can even be used to check the validity of computer models.

3.2.2 Relaxation Model

In [12], Preyer also uses a discretization method to calculate the surface charge. However, his model is more simple and, as observed above, ultimately attempts only to satisfy Gauss's law. In this model the wire is divided into cells and any charge appearing in each cell is assumed to reside in the center. Iterating through each cell, Coulomb's law is then used to determine the field due to all other cells. That is

$$\vec{E} = \sum_i \frac{q_i}{4\pi\epsilon_0 r_i^2} \hat{r_i}$$

where r_i is the distance between the cell where the field is being calculated and the the ith piece of charge. Once the field at all cells is calculated, charges from each cell are moved according to the relation

$$\Delta q = \sigma E_n s^2 \Delta t$$

where s^2 is the area of the cell face and E_n is the normal component of the field on a particular face and Δt is the elapsed time between iterations, set to $5 \times 10^{-20} s$ in Preyer's calculation. Therefore, the entire computation forms a loop where one iteration calculates the field at each cell and then an appropriate amount of charge. In [12] roughly 200 such iterations are required to reach a steady state, that is when $\Delta q \approx 0$.

This method will produce a stationary charge distribution that minimizes the potential energy of the electric field. However, it will not be stable in the long run. As with my calculations, Preyer considers a wire connected to a discharging capacitor. As $t \to \infty$ the current will drop to zero. Nonetheless, the characteristic time of discharge is much longer than the time required for the charge distribution to relax into a stable state.

4 Running the Simulation

4.1 Choice of Programming Language

Originally the code for this project was written in Xojo. Xojo (formerly known as RealBasic) is a language generally geared toward designing applets. It is also reasonably fast and was well known to me at the start of this thesis. However, part way through the project I made the decision to translate the code into Python. The main reason for this was that Python was easier to use when writing a scientific program such as this. Additionally, Python is a commonly used language in computational physics and I felt that it would be well worth improving my knowledge of it. I also gave brief consideration to translating the code into to C++. Ultimately, I chose not to use C++ because I felt its superior speed was outweighed by the ease of using Python.

4.2 Python Implementation

In order to simulate the evolution of surface charge on a wire I created a Python routine with five files. Listed in the order that they are run, these are:

`System_Parameters.py`
`Get_Wire_Shape.py`
`Circuit_Initialization.py`
`Update_Equations.py`
`Color_Plot.py`

The first file, `System_Parameters.py`, is simply a repository for important parameters that appear throughout the program. It is not used to compute any values but it does play a significant role in the routine. By placing all the global parameters in a separate file it reduces clutter and minimizes the number of times parameters have to be imported between files.

The most important parameter I set within `System_Parameters.py` is SIZE,

which determines the size of the arrays I use. This parameter is so influential because almost every array is set to be of dimension SIZE $\times$ SIZE $\times$ SIZE, which means that SIZE greatly influences not only the precision of the simulation but also the runtime. Further, this crucial constant also determines the values of many other parameters. In particular, the parameter *spaceStep*, which determines the distance between adjacent nodes, is set this to be 0.1/SIZE. Since the units of distance are assumed to be meters, this means that my wire system represents a cube of space $1cm \times 1cm \times 1cm$. In this file, I also give the value of *timeStep*, which determines how much time is assumed to pass between successive updates of the fields and is set to $0.1s$.

Lastly, `System_Parameters.py` is, naturally, the file where I list the physical constants used in my code. Most importantly, this is where I make explicit the assumption that the circuit is made of copper. This means that, in my conductivity array, nodes at which the circuit appears have a conductivity of [3]

$$\sigma_{circuit} = 5.95 \times 10^6 \ \ 1/(\Omega m)$$

and nodes which correspond to free space have a conductivity of

$$\sigma_{free\ space} = 0$$

In addition to needing the conductivity of copper my program also requires knowing its dielectric constant and permeability. Approximate literature values are used, meaning these are set to be [4]

$$\epsilon_r = 5.6$$

$$\mu_r = -1.0$$

These values are then used to set the permittivity and permeability at nodes where the wire appears. At every other node the permittivity and permeability remain ϵ_0

[3] Value obtained from [1]

[4] ϵ_r was obtained from [14] and μ_r was obtained from [2].

```
In [1]: %run "/Users/willcjbuchholtz/Box Sync/Thesis/Thesis Code/Actual Code/Get_Wire_Shape.py"
This program allows the user to initialize a portion of the wire.
Enter a size for the initialization grid.

10
Here one can give numbers corresponding to squares so as to initialize a wire array.
Give a square number. If finished, type 0.
```

Figure 5: *The prompt that appears when* ***`Get_Wire_Shape.py`*** *is run. In this instance the grid size has been set by the user to 10.*

and μ_0 respectively.

The next two files `Get_Wire_Shape.py` and `Circuit_Initialization.py` function as initialization files. `Get_Wire_Shape.py` allows the user to specify a wire shape manually. This is done by plotting a numbered grid and prompting the user to enter square numbers that will then correspond to part of the wire. Figure 5 shows the prompt that appears when the file is run and Figure 6 shows the corresponding grid that the user would be presented with. As can be seen, the capacitor and the two ends of the wire are pre-initialized. Although a valid circuit must terminate at these ends, the user has the freedom to construct any continuous shape in-between. As can be seen by the prompt in Figure 5, the user also has the freedom to indicate the size of grid with which he or she wishes to conduct the initialization. Larger grids allow for a more intricate wire shape but are more tedious to work with. Figure 7 shows such a grid where the side length has been set to 20.

To enable a user to initialize a three dimensional wire via a two dimensional interface, it is assumed that the circuit has z axis symmetry. That is, the grid from `Get_Wire_Shape.py` is assumed to lie in the xy-plane and the z direction is initialized simply by stacking similar planes together, a process visualized in Figure 9. Technically, though, there is not complete z-axis symmetry. The capacitor plates are assumed to extend higher and lower than the wire, meaning that there are some xy-planes in which only the capacitor appears.

In `Circuit_Initialization.py`, the wire shape specified by the user is placed into the conductivity array. In addition, there are two important events which occur

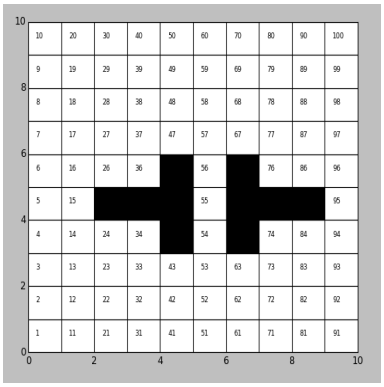

Figure 6: *The initial grid that appears when* ***`Get_Wire_Shape.py`*** *is run. Note that two squares are automatically shaded squares. These represent endpoints for the user's wire which must be reached in order for the program to run.*

and deserve note. Firstly, the user is prompted to indicate how much charge initially appears on the capacitor plates. This, along with the shape of the wire, is one of the key initial parameters that this program takes. Using this value the program is able to initialize the electric field within the capacitor using the equation

$$\vec{E} = \frac{\sigma}{\epsilon_0}\hat{n}$$

In this case σ is *not* conductivity but rather surface charge density. Note also that the unit vector $\hat{n}$ indicates that the field points perpendicular to the capacitor, from the positive plate to the negative plate. Therefore, when a charge of Q is entered, `Circuit_Initialization.py` places $+Q$ on the left plate, $-Q$ on the right plate and sets the field to be

$$\vec{E} = \frac{Q}{A\epsilon_0}\hat{n} = \frac{Q}{(\text{Capacitor Length} \cdot \text{Capacitor Height})\ \epsilon_0}\ \hat{y}$$

Outside the capacitor the electric field is set to zero. Naturally, this fails to account for fringing effects at the edges of the capacitor plates. Neglecting this effect is one notable way in which my program makes an approximation.

Once `Get_Wire_Shape.py` and `Circuit_Initialization.py` have run the wire

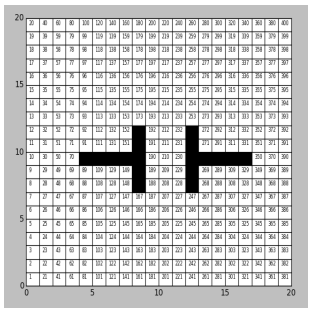

Figure 7: *An initial grid that with side length set to 20.*

and initial electric field are completely set. Additionally, because there is no moving charge at the instant the capacitor begins discharging this means that magnetic field is initially zero everywhere, which `Circuit_Initialization.py` ensures.

`Update_Equations.py` is the file where all the important physics takes place. Using the discretized forms of Maxwell's equations derived in section 3.1.3, `Update_Equations.py` essentially consists of a giant *for* loop whose index, "timeStep," increments forward in time. During each iteration of the loop, the amount of charge on the capacitor is reduced by stepping down the field between the plates via a negative exponential. The six update equations - one for each component of each the electric and magnetic fields - then move through each point in space and update the fields based on the previous values. It is also in this *for* loop where I insert the code implementing an ABC. This code modifies only the nodes on the boundaries of the electric field arrays and ensures that the arrays behave as if they were infinite.

As can be seen by examining equations $(9), (10), (11)$ and $(12), (13), (14)$, all the update equations involve components of both the electric and magnetic fields. Because of this coupling, my routine updates the three components of the magnetic field first, followed by the three components of the electric field. This "leapfrog" procedure alternately advances each field in time and is a practice described in [13]. Ultimately, the time step loop in `Update_Equations.py` continues only for some preset "Maxtime." This is set long enough that the decaying exponential

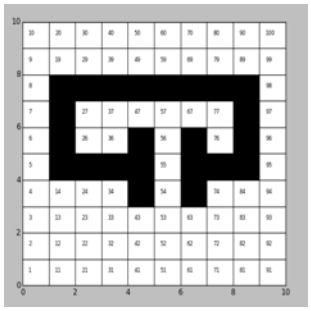

Figure 8: *A completed circuit appearing on a grid of side length 10. Note that the only restrictions are that the wire be continuous and that it terminate at the the capacitor plates.*

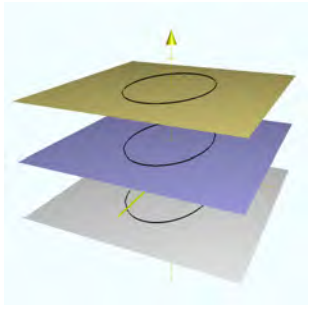

Figure 9: *An example of z-axis symmetry achieved by stacking identical planes. Because the capacitor plates extend above and below the wire, this symmetry actually occurs for only part of the z-axis.*

which models the discharge of the capacitor has become entirely negligible and so that there is sufficient time for the system to settle into equilibrium. Once a final electric field is obtained, `Update_Equations.py` then easily calculates the charge density at each point in space using Gauss's law, as described in section 3.1.3. These values are stored in an array which is passed to `Color_Plot.py`.

`Color_Plot.py` provides plots of both the final charge distribution as well as the final electric field. This is implemented via a scatter plot of a cross-section representing the top of the circuit, i.e. the plane above which there is only empty space and the capacitor plates. Ultimately, `Color_Plot.py` is the simplest of the four main files and can be best understood by being read. The code, along with the other four files, is provided in the Appendix.

5 Results

5.1 Preliminary Results

Because of the difficulty of the overall calculation, I decided to first troubleshoot pieces of my routine in isolation. The strategy was to confirm that key parts of my code could produce calculations that were indeed physical. The two main modules that I discuss here amount to a test implementation of Gauss's law and a test implementation of Faraday's and Ampere's laws.

5.1.1 Field of a Point Charge

The first module of code that I wrote is a simple application of Gauss's law. The file, provided in the appendix under the name `Point_Charge.py`, calculates the electric field of a point charge and then attempts to recover that simple distribution by taking the divergence of the field just calculated.

`Point_Charge.py` creates two plots. First, it produces a simple color plot in which the amount of red and green appearing at a given pixel is determined by the amount of electric field calculated there. (The decision to only include red and green while excluding blue was an arbitrary decision made for aesthetic reasons.) Secondly, a plot is produced showing the presence of charge at each pixel. Since only one point charge is calculated this plot results in only a couple of pixels being shaded. The results of the two plots can be seen in Figure 10 and Figure 11.

Overall, the results of this file confirm that my code was accurately able to recover a charge distribution using Gauss's law. However, an important lesson is learned from analyzing Figure 11. As can be seen, there is indeed a strong concentration of charge at the center of the plot, which is as expected given the field from Figure 10. Additionally, though, there is a small amount of charge that appears surrounding the center and should not be there. This is due to the fact that when implementing Gauss's law I was forced to use finite differentials instead of true partial derivatives. In the limit that my array is made larger and larger this numerical artifact disappears. Although, the presence of extra charge was in no

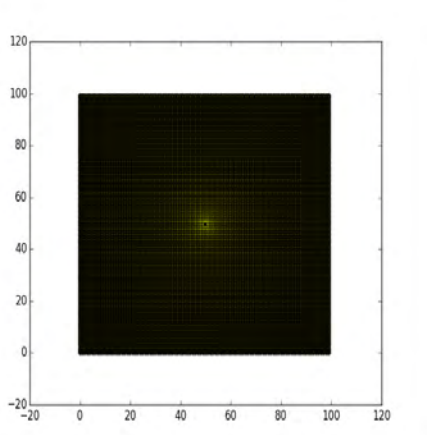

Figure 10: *This is a plot of the field of a point charge. As can be seen the magnitude drops off steeply, given the* $1/r^2$ *dependence, and so the relevant field is confined to a small space around the charge. The numbering on the axes represents only the position of the pixels within a* 100×100 *array.*

way fatal to this test calculation, it acts as a reminder of the importance of using large arrays. If one uses too small an array in a sensitive calculation, numerical approximations such as this could produce severely unphysical results.

5.1.2 Wave in Free Space

The central engine of my Python routine is the update equations produced by discretizing Faraday's and Ampere's laws. Confirming that these can be accurately implemented through code is therefore absolutely essential. To test my discretization I wrote a file called `Gaussian_Pulse.py`, which is an animated simulation of an electromagnetic wave propagating through space according to Maxwell's equations. The wave is generated by a Gaussian pulse which creates a nonzero electric field at a "source node." Because the discretized forms of Faraday's and Ampere's laws link neighboring nodes this allows the source node to transmit its pulse outward.

Figure 12 and Figure 13 present screen shots of `Gaussian_Pulse.py` midway through its simulation and demonstrate how it successfully creates a propagat-

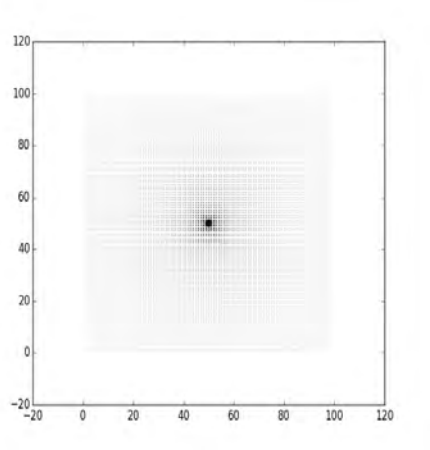

Figure 11: *This is the charge distribution calculated using Gauss's law and the field from Figure 10. Again, the numbering on the axes indicates the position of the pixels within a* 100×100 *array.*

ing electromagnetic pulse. Figure 12 presents the simulation just after starting and shows how the pulse, which originates from a source node at position 100, is able to generate two waves that move outward in opposite directions. Ultimately, apart from testing the ability of my discretization to propagate the pulse, `Gaussian_Pulse.py` also tests another important feature of my overall routine: an ABC (Absorbing Boundary Condtion). An ABC is an crucial modification to the basic computational grid because it allows one to make a finite array behave as an infinite one.

`Gaussian_Pulse.py` requires an ABC because I attempt to simulate a wave that travels in completely open and empty space, meaning that the waves should disappear when they reach the boundaries. This can be implemented through code in many ways and, in fact, there is a body of literature exploring different methods [5]. In `Gaussian_Pulse.py`, as well as my final program, I use a simple ABC implementation which is described in [13]. As can be seen in Figure 13, this implementation works to satisfaction as the left wave has completely moved off the left edge. The

[5] See [4] for an advanced discussion involving a PML (Perfectly Matched Layer), which is an idealization of an ABC.

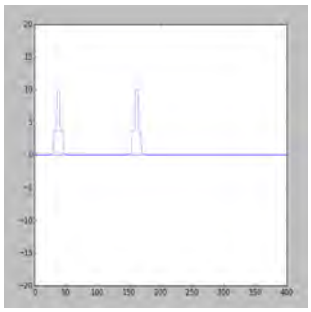

Figure 12: *This shows two waves traveling in opposite directions on a grid of length 400. The waves originate from a Gaussian pulse at node 100 (refer to horizontal axis) and propagate using discretized forms of Faraday's and Ampere's laws as update equations. The vertical axis is unitless.*

ABC is implemented on both sides of the grid so at a later time step the right wave moves off the right edge in a similar fashion. It should be noted that the ABC implemented in `Gaussian_Pulse.py` is a one-dimensional ABC whereas the ABC I use in my final code is three dimensional. The three-dimensional implementation is essentially an analogous extension of this method into higher dimensions. However, as explained in [13] the three-dimensional simulation is, as one might expect, slightly more sophisticated. Therefore, it is important to note that a successful proof of method in `Gaussian_Pulse.py` does not guarantee that the ABC implemented in `Update_Equations.py` is fully accurate.

It is also important to add a precaution about the successful implementation of Faraday's and Ampere's laws in `Gaussian_Pulse.py`. In a similar way, the three-dimensional implementation of Maxwell's equations is essentially the same as the one dimensional version but with a small amount of added sophistication. Although `Gaussian_Pulse.py` strongly suggests that the three-dimensional equations should function accurately one must note that it does not provide proof of this.

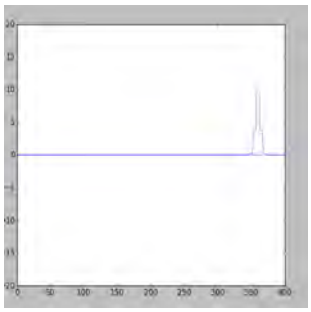

Figure 13: *This plot shows the same simulation in* Figure 12 *but a few time steps later. The left wave in* Figure 12 *has moved off the left edge due to the absorbing boundary condition. The other wave will continue moving to the right until it moves off the right edge. The horizontal axis again represents node numbers in the array while the vertical axis is unitless.*

5.2 Results of Full Simulation

After writing and debugging my Python routine, the simulations I ran proved inconclusive. The reason the results I produced were unacceptable is that they are clearly unphysical. In particular, the charge placed on the capacitor plates did not disperse itself throughout the wire and instead leaked out into open space. Since I am confident that the physics and computational algorithm used by this thesis are correct - Maxwell's equations are universally accepted and the FDTD method is widely used - I dedicate the following sections to presenting the distributions I calculated and exploring what went wrong with my implementation.

5.2.1 Calculated Plots

I found it most useful to present my results as two dimensional plots. As explained above, my Python routine uses a color-coded system to indicate where charge appears. Pixels that are black indicate the edge of the wire, pixels that are grey indicate points where there is no charge and colored pixels indicate points where there is charge. The coloring is done via a logarithmic scale where the degree of red indicates the amount of positive charge and the degree of blue indicates the amount of negative charge. Because it was necessary to scale every value between 0 and

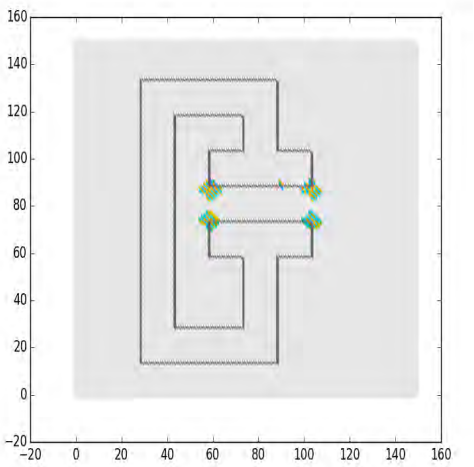

Figure 14: *A plot of the circuit after 0.5 seconds simulated using a* 150×150 *grid with* 1 *Coulomb placed on either plate. Runtime was about* 580 *seconds.*

1, all the positive logarithm values were divided by the maximum positive value and all the negative logarithm values were divided by the minimum negative value. For aesthetic reasons I set all pixels that contain nonzero charge to have a certain degree of green (0.75). This mean the color at any given pixel was determined by the rgb tuple

$$\text{rgb}(\ \log(\text{charge})\ /\ \text{Max Positive Value},\ \ 0.75,\ \ 0)$$

if the charge there was positive, or

$$\text{rgb}(\ \ 0,\ \ 0.75,\ \ \log(\text{charge})/\ \text{Min Negative Value})$$

if the charge there was negative. Whenever the charge was between 1 and -1 this value was rounded down to zero. This avoided such pixels getting mapped to large values via the logarithm and was a reasonable approximation (most of the charge values were much greater or less than 1.) It is good to remember that pixels which

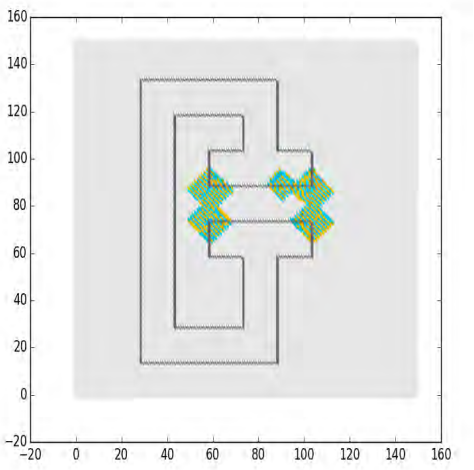

Figure 15: *The same circuit as shown in* Figure 14 *but after 1.0 seconds. Runtime in this case lasted about 956 seconds.*

appear mostly green contain almost no charge. Also, it is important to note that, in my code, the black wire pixels are added *last*. Therefore, in some cases, charge appears on these edges but is subsequently colored over.

Figure 14 shows a typical result produced by my program. This particular plot was made using a 150×150 grid requiring about 580 seconds of runtime. The circuit is assumed to have 1 Coulomb of charge placed on each plate and is shown 0.5 seconds after it begins discharging. The cross-section shown is the top of the wire, meaning that the plane immediately above this on the z-axis will only contain the capacitor plates, as they extend above and below the wire. Figure 15 shows this same circuit after a 1.0 seconds have elapsed. At this point, this amount of charge which has leaked off of the capacitor has grown substantially. In my simulation, the charge propagates outward in a square pattern which is, clearly, very unphysical. The charge also alternates sign rapidly within these squares, much like a checkerboard, which one would not expect. Lastly, Figure 16 shows the circuit after 1.5 seconds have passed. Here, the capacitor has, completely discharged and

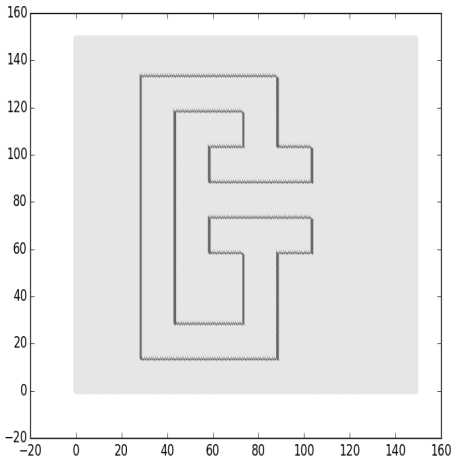

Figure 16: *The circuit from* Figure 14 *after* 1.5 *seconds, at which point the capacitor has completely discharged. Runtime lasted about 1620 seconds.*

so the results, which show no distribution, do make sense. There need be no surface charge because no current should flow once the capacitor is neutral.

As can be seen by the runtimes listed in Figures 14, 15, and 16 these simulations took approximately 10, 15 and 30 minutes to complete respectively. These runtimes scale linearly. This is because as the only change that was made between the simulations was the value of "Maxtime", a parameter which sets the number of iterations the time loop steps through. For instance, the second calculation simulated 1.5 times as much internal "circuit time" and so took about 1.5 times as long to complete.

5.2.2 Sources of Error

My program clearly suffered from at least one, and probably several, fatal bugs. The most notable error can be identified when one attempts to calculate the initial charge distribution of the circuit, before the capacitor begins to discharge. Figure 17 shows the starting electric field I initialized for my circuit, which I model to be constant

throughout. An infinite capacitor should, indeed, have a constant internal field but a finite capacitor, such as this, would experience fringing effects on the edges. That is the field would fray outward at the edges of the plates and not point completely perpendicular to them. However, although this is a serious approximation, I do not believe that neglecting fringing effects, alone accounts for the degree to which my results were unphysical.

Figure 18 shows the initial charge distribution I calculated from the field given by Figure 17. Clearly this distribution is wrong. The initial surface charge distribution should have plotted a line of mostly-red pixels on one plate, corresponding to a sheet of positive charge, and a line of mostly-blue ones, corresponding to a negative sheet, on the other. This incorrect initial distribution immediately disqualifies all subsequent results. A system that does not begin with the correct initial conditions cannot be expected to evolve as desired. Given the results of Section 5.1.1, I conclude that the error in my program is not my implementation of Gauss's law. Instead, the bug resulting in Figure 18 must occur elsewhere. For me, resolving the dilemma presented by Figure 18 is key because, regardless of the accuracy of the rest of the program, no remotely physical result will be produced until it is resolved.

Because of the fatal bugs in my program, it is difficult to assess the impact of the *physical* approximations and sources of error on my simulation. Naturally though, there are many assumptions I make. For one, I assume that the wire discharges exponentially. This is a plausible guess (exponential decay is very common in nature) but otherwise completely unjustified. I also model the circuit as being made out of completely uniform copper. Any realistic circuit, of course, would have imperfections. In a better model I would want to account for this by varying the conductivity, permittivity and permeability within the wire. Additionally, because I was never able to make the program work fully, I was unable to test the efficacy of the ABC I implemented. The ABC I wrote was taken directly from [13]. However, Schneider admits that this particular ABC is relatively primitive. Ideally, I would implement a PML (Perfectly Matched Layer) which would mean that any radiation that approached the boundaries of the grid would fully transmit across it.

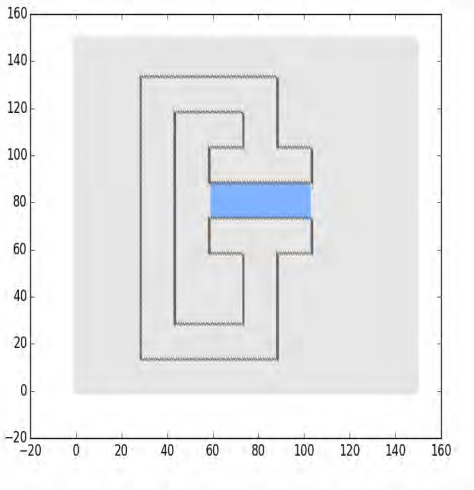

Figure 17: *This plot shows a typical initial field (in blue) between the capacitor plates. Note that the field is constant throughout the middle (as it should be) but does not account for fringing on the edges. The coloring scheme is arbitrary.*

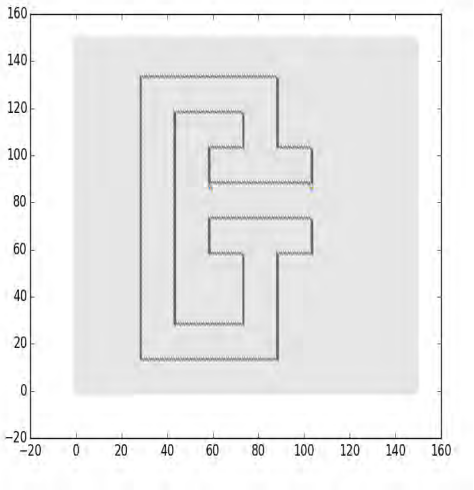

Figure 18: *This plot shows the initial charge distribution I calculated using an initial field of the type shown in* Figure 17*. Runtime for this plot was about 301 seconds. The small points of charge are obviously incorrect since there should be a line of positive charge on one plate and a line of negative charge on the other.*

6 Conclusion

6.1 Assessing the Success of the Project

Ultimately, this project produced no significant results. The nominal goal was to create physically accurate plots of the surface charge for several different wire shapes and I was unable to do so. However, from an educational perspective this project was a success. This was my first ever attempt at a simulation of this magnitude and complexity and I was able to produce a program that nearly achieved its objective. Also, all the skills learned in doing this project are essential tools for a scientist doing computational physics. Along the way I learned, among other things, how to

- Program in Python
- Approximate Maxwell's equations using differentials
- Use an FDTD algorithm
- Code an ABC

In particular, this thesis was essentially an extended exploration of how to approximate continuous media and equations using large matrices and finite differentials. In my final calculations, I approximated my circuit system as a number of 150×150 size matrices which defined a collection of values at each point in space. These values included the electric and magnetic fields, the conductivity, the permeability and permittivity, and the charge. I then used an FTDT algorithm to approximate Maxwell's equations and create update equations for my system. Using these update equations I placed the system arrays in a time loop and advanced them using an index which represented a time step.

The distributions I calculated were clearly wrong because charge escaped out into open space. In addition I experienced the phenomenon where I only calculated charge in rectangular pockets along the capacitor plates. However, as required, I did calculate that the surface charge distribution disappeared after the capacitor had completely discharged, as can be seen in Figure 16. It is my belief that the reason for

these unsatisfactory results is due to the *code* and not the theory. Although I had some latitude in setting the particulars of my circuit system (which algorithm to use, what material to model, how large to make the circuit, ect.) the essential physics was well known and the computational theory (learned from [13]) was mainstream. While working on this project, I discovered that small coding errors (misplaced subscripts, incorrect index bounds, ect.) often occurred and, when fixed, notably altered my results. I do not expect that the cause of my unphysical calculations is the result of any single bug. Rather, it is probably caused by a collection of trivial coding errors and oversights.

There is, in particular, one concrete way I can point to where bugs may have arisen while writing this program. Originally, my Python routine was a single file. After this file became unreasonably large I split the program into three files and eventually five. It is possible that some of the fatal errors occurred while restructuring my program. In particular, at one point I found it helpful to transfer most of the important variables to a single file, `System_Parameters.py`. In retrospect, this process was extremely risky and I might have been best served simply by rewriting the entire routine. It is possible that during this modification I failed to notice how the different files interacted. This was a naive mistake that I hope to avoid in the future.

6.2 Utility of the Program

Because I was unable to fix the bugs in my program, I am unfortunately unable to provide useful plots at this time. One motivating factor for this project was the idea that I would be able to generate simulations that would have educational value. If I were to return to this project this would still be a possibility, given that my program is close to achieving its goal. As demonstrated by the results presented in sections 5.1.1 and 5.1.2, pieces of my code work in isolation and it would seem that bringing the project to completion would only require minor editing rather than large structural changes. Therefore, this is a project that I hope to return to because interesting results could be achieved with relatively little effort.

It goes without saying this project currently does not contribute to the scientific literature investigating surface charge on wires. However, as discussed above, if I am able to bring the project to completion it should make a notable contribution. Given that I attempt to fully solve Maxwell's equations my results should be significantly more accurate than either [11] or [12]. If I ultimately produce results, this is a project I would seek to have published.

7 Appendix

7.1 Example Conductivity Array

The following is a printed example of how the circuit can be represented as an array. Values of 1 correspond to high conductivity and values of 0 are low conductivity. The shown arrays are two cross-sections of a three dimensional array and illustrate how some cross sections contain the full circuit and others only the capacitor plates.

```
[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  1.  1.  1.  1.  1.  1.  1.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  1.  0.]
 [ 0.  1.  0.  0.  1.  0.  1.  0.  1.  0.]
 [ 0.  1.  1.  1.  1.  0.  1.  1.  1.  0.]
 [ 0.  0.  0.  0.  1.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]

[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

7.2 Code

Listed here are the five files of of my routine as well as the other python code used in this thesis. The files constituting my main routine are listed last and begin with `System_Parameters.py`. Other than `System_Parameters.py` they are all interdependent. and are therefore listed in the order with which they must be run. Note that the page numbering on the following pages corresponds to the files of code and not the master document.

```
# File name: Point_Charge

# Import major libraries
import matplotlib.pyplot as pl
import numpy as np


# Parameters
SIZE = 100
ArrayDistance = 2
fieldColor = (0,0,0)
x0 = float(SIZE)/2
y0 = float(SIZE)/2


# The electric field and charge arrays
Ex = np.zeros((SIZE,SIZE))
Ey = np.zeros((SIZE,SIZE))
Etotal = np.zeros((SIZE,SIZE))
rho = np.zeros((SIZE,SIZE))

# Set the electric field to obey Coulomb's law
for i in range(SIZE):
    for j in range(SIZE):
        if not(i==x0 and j==y0):

            r = ( (i-x0)**2 + (j-y0)**2 )**(0.5)
            Ex[i][j] = (i-x0)*(r**(-1.5))
            Ey[i][j] = (j-x0)*(r**(-1.5))

            Etotal[i][j] =(Ex[i][j]**(2) + Ey[i][j]**(2))**(0.5)

# Calculate the charge distribution
for i in range(1,SIZE-1):
    for j in range(1,SIZE-1):
        rho[i][j] = (0.5)*( (Ex[i+1][j] - Ex[i-1][j] ) +
        (Ey[i][j+1] - Ey[i][j-1]) )

#Plot the field
for i in range(SIZE):
    for j in range(SIZE):

        fieldColor = (Etotal[i][j],Etotal[i][j], 0)
```

```
        plot1 = pl.scatter(i,j, c=fieldColor )

pl.show()

print "For plot of charge density type 'c'"

if raw_input()== "c":
    pl.close()

    # Plot the charge distribution
    for i in range(SIZE):
        for j in range(SIZE):
            # Note that the factor of 4 multiplying rho is an
            #arbitrary constant which helps make the results of
            #the calculation more visible as a plot.
            plot1 = pl.scatter(i,j, 4*rho[i][j])
pl.show()
```

```python
# File name: Guassian_PulseABC.py

# This program is supposed to model a simple Guassian Pulse
# using Faraday's and Ampere's laws and an ABC.

# Import all necessary libraries
import numpy as np
import pylab as pl
import matplotlib.pyplot as plt
import matplotlib.animation as animation


# Define important parameters
SIZE = 400
SOURCENODE = 100

# Define the field arrays
Ez = [0]*SIZE
By = [0]*SIZE

# The function which updates the field arrays
def update(timeStep):

    # This helps create the ABC
    By[SIZE-2] = By[SIZE-3]
    # This updates the magnetic field array
    for i in range(0,SIZE-2):
        By[i] = By[i] + (Ez[i+1] - Ez[i])

    # This helps create the ABC
    Ez[0] = Ez[1]
    # This updates the electric field array
    for i in range(1,SIZE-1):
        Ez[i] = Ez[i] + (By[i] - By[i-1])

    # This updates the source of the pulse
    Ez[SOURCENODE] = 10*np.exp(-(((timeStep-100)/5)**2))

    # This outputs the new value of Ez
    return Ez
```

```
#################################################################
# The following code animates the pulse

# Initialize the domain x
x = [0]*SIZE
for i in range(1,SIZE-1):
    x[i] = i

# Variables needed for the plot
fig = plt.figure()
ax = plt.axes(xlim=(0, SIZE), ylim=(-10, 10))
EzGraph, = ax.plot(Ez, lw=.5)
pl.ylim([-20,20])


# This is the function that enables the animation
def animate(timeStep):
    EzGraph.set_ydata(update(timeStep))
    if timeStep%10 ==0:
        print timeStep
    return EzGraph,

ani = animation.FuncAnimation(fig, animate, interval=25,
        blit= True )

# Plot the results
pl.show()
```

```
# File name: System_Parameters.py

# Import major libraries
import matplotlib.pyplot as plt
import numpy as np

# Opening statement
print "Enter a size for the initialization grid."


# This defineds the side length of the initialization grid
GridLength = input()

# SIZE is the most important parameter in this routine.
# It defines the size of the system arrays.
SIZE = 150


# Parameters for timeStep loop

# Number of iterations of time loop
Maxtime = 1

 #[m] The size of the array is 10cm x 10cm x 10cm
spaceStep = float(0.1)/SIZE

# [s] The time increment for the time loop.
timeStep = 0.1

# The ratio of timeStep to spaceStep
CourantNum = float(timeStep)/spaceStep

# This helps scale between the arrays of size GridLength
# and the arrays of size SIZE
ScaleFactor = float(SIZE)/GridLength


# Physical constants

# [m^-3 kg^-1 s^4 A^2]
eps0 = 8.85418782*(10**(-12))

# [m kg s^-2 A^-2]
```

```
mu0 = 1.25663706*(10**(-6))

# [S/m] of [1/ (Ωm) ]  Set to conductivity of copper
MaterialConduc = 5.95*(10**7)

# Dielectric constant of copper
epsCu = 5.6

# Permeability of coper
muCu = -1.0

# Parameters for grid initialization
gCFrontEdge = np.floor(GridLength*.4)
gCBackEdge = np.ceil(GridLength*.6)
gLeftPlate = np.floor(GridLength*.4)
gRightPlate = np.ceil(GridLength*.6)
gCapLength = abs(gCFrontEdge - gCBackEdge)

# Parameters needed to initialize ConDuc and the
# elctric field array
WireTop = np.ceil(SIZE*.6)
WireBottom = np.floor(SIZE*.4)
CapTop = np.ceil(SIZE*.75)
CapBottom = np.floor(SIZE*.25)
CapHeight = abs(CapTop - CapBottom)
CapBackEdge = np.ceil(gCBackEdge*ScaleFactor)
CapFrontEdge = np.floor(gCFrontEdge*ScaleFactor)
CapLength = np.ceil(gCapLength*ScaleFactor)
LeftPlatePos = np.floor(gLeftPlate*ScaleFactor)
RightPlatePos = np.ceil(gRightPlate*ScaleFactor)

# This is the cross-section of the wire that gets plotted
WireMidPlane = np.floor((CapTop + CapBottom)*.5)

# These parameters control how much of the capacitor
# plates actually get initialized with a non-zero
# field. The idea to *not* fully fill the capacitor
# With an electric field extending to the edges is
# to minimize fringing effects. Note that since
# CapTop/CapBottom = 3/2, TopIndent/BottomIndent
# should be < 2/3 so that the top part edge of the
# field is not the same as the bottom edge. Similarly
# since CapBackEdge/CapFrontEdge = 3/2 then it should
```

```
# be that CapBackIndent/CapBackIndent < 2/3.
#TopIndent = 0.9
#BottomIndent = 1.1
#FrontIndent = 1.1
#BackIndent = 0.9

TopIndent = 1
BottomIndent = 1
FrontIndent = 1
BackIndent = 1

# Declarations necessary to create the plots
fig = plt.figure()
ax = fig.add_subplot(111)
```

```
# File name: Get_Wire_Shape.py

# Import necessary libraries
from numpy import *
from System_Parameters import *
import matplotlib.pyplot as plt



#####################################################################
# This is the array corresponding directly to squares on
# the grid.
gridCircuit = zeros((GridLength+1,GridLength+1))

#####################################################################



#####################################################################
# These lines make and number the grid

# Make the grid
for i in range(0,GridLength+1):
    plt.axhline(i,0,GridLength, color = 'black')
    plt.axvline(i,0,GridLength, color = 'black')

# Number the grid
for i in range(1,GridLength+1):
    for j in range(1,GridLength+1):
            ax.text(i - 0.75, j - 0.5, (i-1)*GridLength + j,
            fontsize=(10)*(1-.015*GridLength))

#####################################################################
# This initializes the capacitor plates

# Define an index for the while loop
tau = gCFrontEdge

# Place the capacitors on the grid
while gCFrontEdge <= tau <= gCBackEdge:
    # The capacitor nodes. LCN/ RCN = Left/right capacitor nodes
    LCN = [gLeftPlate, tau]
    RCN = [gRightPlate, tau]
```

```
    # Place the initial capacitor nodes in the NewWirePos list
    gridCircuit[tau,gLeftPlate] =1
    gridCircuit[tau,gRightPlate] =1

    # This colors in the initial wire nodes on the grid
    for i in [LCN, RCN]:
        # i[0] is the x coordinate and i[1] is the y coordinate
        a = [i[0],i[0]+1,i[0]+1,i[0]]
        b = [i[1],i[1],i[1]-1,i[1]-1]

        # This shades in the square defined by a,b
        plt.fill(a,b, color = 'black')

    # Increment the index
    tau = tau + 1

# Reset the index tau
tau = 0

# This places the beginning of the wire on the grid
while 0 <= tau <= floor(GridLength*.2):
    # The initial wire nodes. LWN/ RWN = Left/right wire nodes
    LWN = [floor(GridLength*.4) - tau, floor(GridLength*.5)]
    RWN = [floor(GridLength*.6) + tau, floor(GridLength*.5)]

    # Place the initial wire nodes in the NewWirePos list
    gridCircuit[floor(GridLength*.5),floor(GridLength*.4)-tau]=1
    gridCircuit[floor(GridLength*.5),floor(GridLength*.6)+tau]=1

    # This colors in the initial wire nodes on the grid
    for i in [LWN, RWN]:
        # i[0] is the x coordinate and i[1] is the y coordinate
        a = [i[0],i[0]+1,i[0]+1,i[0]]
        b = [i[1],i[1],i[1]-1,i[1]-1]

        # This shades in the square defined by a,b
        plt.fill(a,b, color = 'black')

    # Increment the index
    tau = tau + 1
```

```
# Show the grid
fig.show()
#####################################################################



#####################################################################
# These lines allow the user to initialize wire positions
# using the grid

# Booleans needed for following code
stdCircuitCheck = False
stillGoing = True
MakeCustomCir = False

# List needed for following code
CircuitSqr = []

# Instructions for giving numbers
print 'Here one can give numbers corresponding to squares so \
as to initialize a wire array.'
print 'If grid is 10x10 type "s" for a standard circuit. \
Otherwise, type 0.'

userPreference = raw_input()

# This unlocks a list of square positions that can be used
# to make a custom circuit
if userPreference == "s":
    MakeCustomCir = True
    CircuitSqr = [15,16,17,18,28,38,48,58,68,78,88,87,86]

# Allow the user to enter square numbers
while stillGoing:
    # Get a number or an array
    print "Give a square number. If finished, type 0."

    # This segment is where the user enters the square numbers
    # that will be part of the wire. If there is an error this
    # is caught by the excpetion.
    try:

        if MakeCustomCir:
```

```
        # Handle each number in the list
        for i in CircuitSqr:

            # This extracts x,y coordinates
            x = (i - i%10)/10
            y = i%10
            if y ==0:
                y = y+10

            # This defines vertices of a grid square from
            # x and y
            a = [x,x+1,x+1,x]
            b = [y,y,y-1,y-1]

            # This shades the square corresponding to the
            # vertices from a,b
            plt.fill(a,b, color = 'black')

            # Add the new grid position to gridCircuit
            gridCircuit[GridLength-y][x] = 1

        # This ensures this part of the loop is not
        # entered again
        MakeCustomCir = False


    # What the user types into the command line
    inputNum = floor(abs( float(raw_input())))

    # Exit the loop or shade in the appropriate square
    if inputNum == 0:
        stillGoing = False

    # This handles the case that a single square number
    # was entered
    elif inputNum <= GridLength**2:

        # This extracts x,y coordinates
        x = (inputNum - inputNum%10)/10
        y = inputNum%10
        if y ==0:
            y = y+10
```

```
            # This defines vertices of a grid square from x
            # and y
            a = [x,x+1,x+1,x]
            b = [y,y,y-1,y-1]

            # This shades the square corresponding to the
            # vertices from a,b
            plt.fill(a,b, color = 'black')

            # Add the new grid position to gridCircuit
            gridCircuit[GridLength-y][x] = 1

    except:
        print "Error. Try again."


# Redraw the canvas to include the newly shaded tiles
fig.canvas.draw()
```

```
# File name: Circuit_Initialization.py

# Import all necessary libraries
import numpy as np
from System_Parameters import *
from Get_Wire_Shape import *


################################################################
# This is where the user is prompted to give a voltage
# accross the capacitor
print 'Enter the charge in coulombs that appears on each \
capacitor plate. (e.g. 10 --> 10C)'

initialCharge = input()

# This uses the initial charge to create the initial capacitor
# field in Newtons per Coulomb
initCapField = (float(initialCharge)/((CapLength*CapHeight)
*eps0)) #[N/C]

################################################################




################################################################
# This is where I declare all the system arrays
# The conductivity array - essentially defines where the
# wire is
ConDuc = np.zeros((SIZE,SIZE,SIZE))

# The electric field arrays
Ex = np.zeros((SIZE,SIZE,SIZE))
Ey = np.zeros((SIZE,SIZE,SIZE))
Ez = np.zeros((SIZE,SIZE,SIZE))

# The magnetic field arrays
Bx= np.zeros((SIZE,SIZE,SIZE))
By= np.zeros((SIZE,SIZE,SIZE))
Bz= np.zeros((SIZE,SIZE,SIZE))
```

```
# The permittivity array
eps = np.zeros((SIZE,SIZE,SIZE))

# The permeability array
mu = np.zeros((SIZE,SIZE,SIZE))

# The charge array
qDen = np.zeros((SIZE,SIZE,SIZE))

# While the conductivity array is set to a default of
# zero, this loop sets eps and mu to default values of esp0
# and mu0 respectively
for i in range(SIZE):
    for j in range(SIZE):
        for k in range(SIZE):
            eps[i][j][k] = eps0
            mu[i][j][k] = mu0

# This is the version of NewCircuit but scaled up to the
# appropriate size and multiplied by the conductivity of
# the material.
ScaledNewCircuit = np.kron(gridCircuit, np.ones(
(ScaleFactor,ScaleFactor)))
#####################################################################



#####################################################################
# This is where I initialize the appropriate system arrays
for i in range(SIZE):
    for j in range(SIZE):
        for k in range(SIZE):


            if WireBottom <= i <= WireTop:
                # Initialize the conductivity array
                ConDuc[i][j][k] = (ScaledNewCircuit[j][k]
                *MaterialConduc)


                if ScaledNewCircuit[j][k] == 1:

                    # Initialize the permittivity array
```

```
            # (eps[i][j][k] is set to a default
            # of eps0)
            eps[i][j][k] = epsCu*eps0

            # Initialize the permeability array
            # (mu[i][j][j] is set to a defauly
            # of mu0
            mu[i][j][k] = muCu*mu0

    # This check says that if the index i is between
    # the top and bottom of the capacitor plates but
    # not between the top and bottom of the wire then
    # initialize only the part of the circuit
    # corresponding to the capacitor plates.
    elif CapBottom <= i <= CapTop:
        if CapFrontEdge <= j <= CapBackEdge:
            if k == LeftPlatePos or k == RightPlatePos:

                # Initialize the conductivity array
                ConDuc[i][j][k] = (ScaledNewCircuit[j][k]
                *MaterialConduc)

                # ScaledNewCircuit will be 1 whenever
                # there is a wire node
                if ScaledNewCircuit[j][k] == 1:

                    # Initialize the permittivity array
                    eps[i][j][k] = epsCu*eps0

                    # Initialize the permeability array
                    mu[i][j][k] = muCu*mu0
```

```
# File name: Update_Equations.py

# Import all necessary libraries and files
import numpy as np
import pylab as pl
import matplotlib.pyplot as plt
from System_Parameters import *
from Get_Wire_Shape import *
from Circuit_Initialization import *

#####################################################################

pl.close()

#####################################################################

# The coefficient arrays for the update equations
Cbxb = np.zeros((SIZE,SIZE,SIZE))
Cbxe = np.zeros((SIZE,SIZE,SIZE))
Cbyb = np.zeros((SIZE,SIZE,SIZE))
Cbye = np.zeros((SIZE,SIZE,SIZE))
Cbzb = np.zeros((SIZE,SIZE,SIZE))
Cbze = np.zeros((SIZE,SIZE,SIZE))
Cexe = np.zeros((SIZE,SIZE,SIZE))
Cexb = np.zeros((SIZE,SIZE,SIZE))
Ceye = np.zeros((SIZE,SIZE,SIZE))
Ceyb = np.zeros((SIZE,SIZE,SIZE))
Ceze = np.zeros((SIZE,SIZE,SIZE))
Cezb = np.zeros((SIZE,SIZE,SIZE))

# This is a coefficient used in creating and ABC
abcCoef = float(CourantNum)/((eps[i][j][k]*mu[i][j][k])**0.5)

# These "storage" arrays are necessary so that code lines
# computing the ABC can keep track of the past and
# future fields
exOldXLeft = np.zeros((SIZE,SIZE))
exOldXRight = np.zeros((SIZE,SIZE))
eyOldXLeft = np.zeros((SIZE,SIZE))
eyOldXRight = np.zeros((SIZE,SIZE))
ezOldXLeft = np.zeros((SIZE,SIZE))
ezOldXRight = np.zeros((SIZE,SIZE))
```

```
exOldYLeft = np.zeros((SIZE,SIZE))
exOldYRight = np.zeros((SIZE,SIZE))
eyOldYLeft = np.zeros((SIZE,SIZE))
eyOldYRight = np.zeros((SIZE,SIZE))
ezOldYLeft = np.zeros((SIZE,SIZE))
ezOldYRight = np.zeros((SIZE,SIZE))

exOldZLeft = np.zeros((SIZE,SIZE))
exOldZRight = np.zeros((SIZE,SIZE))
eyOldZLeft = np.zeros((SIZE,SIZE))
eyOldZRight = np.zeros((SIZE,SIZE))
ezOldZLeft = np.zeros((SIZE,SIZE))
ezOldZRight = np.zeros((SIZE,SIZE))


# Initialize the coefficient arrays
for i in range(SIZE):
    for j in range(SIZE):
        for k in range(SIZE):
            Cbxb[i][j][k] = (float(1)-(ConDuc[i][j][k]*timeStep/
            (2*mu[i][j][k])))/(1+ConDuc[i][j][k]*timeStep/(2*
            mu[i][j][k]))

            Cbxe[i][j][k] = (float(1)/(1+ConDuc[i][j][k]*timeStep/
            (2*mu[i][j][k])))*(timeStep/(mu[i][j][k]*spaceStep))

            Cbyb[i][j][k]=(float(1)-(ConDuc[i][j][k]*timeStep/
            (2*mu[i][j][k])))/(1+ConDuc[i][j][k]*timeStep/
            (2*mu[i][j][k]))

            Cbye[i][j][k] = (float(1)/(1+ConDuc[i][j][k]*timeStep
            /(2*mu[i][j][k])))*(timeStep/(mu[i][j][k]*spaceStep))

            Cbzb[i][j][k] = (float(1)-(ConDuc[i][j][k]*timeStep/
            (2*mu[i][j][k])))/(1+ConDuc[i][j][k]*timeStep/
            (2*mu[i][j][k]))

            Cbze[i][j][k] = (float(1)/(1+ConDuc[i][j][k]*timeStep/
            (2*mu[i][j][k])))*(timeStep/(mu[i][j][k]*spaceStep))

            Cexe[i][j][k] = (float(1)-(ConDuc[i][j][k]*timeStep/
            (2*eps[i][j][k])))/(1+ConDuc[i][j][k]*timeStep/
            (2*eps[i][j][k]))
```

```
            Cexb[i][j][k] = (float(1)/(1+ConDuc[i][j][k]*timeStep/
            (2*eps[i][j][k])))*(timeStep/(eps[i][j][k]*spaceStep))

            Ceye[i][j][k] = (float(1)-(ConDuc[i][j][k]*timeStep/
            (2*eps[i][j][k])))/(1+ConDuc[i][j][k]*timeStep/
            (2*eps[i][j][k]))

            Ceyb[i][j][k] = (float(1)/(1+ConDuc[i][j][k]*timeStep/
            (2*eps[i][j][k])))*(timeStep/(eps[i][j][k]*spaceStep))

            Ceze[i][j][k] = (float(1)-(ConDuc[i][j][k]*timeStep/
            (2*eps[i][j][k])))/(1+ConDuc[i][j][k]*timeStep/
            (2*eps[i][j][k]))

            Cezb[i][j][k] = (float(1)/(1+ConDuc[i][j][k]*timeStep/
            (2*eps[i][j][k])))*(timeStep/(eps[i][j][k]*spaceStep))



##################################################################


##################################################################

# This is where I run the update equations through a
# timestep loop
for timeStep in range(0,Maxtime):

#for j in range(int(CapFrontEdge), int(CapBackEdge+1)):
#for k in range(int(LeftPlatePos+1),int(RightPlatePos)):

        #This discharges the capacitor by stepping down the
        # field between the plates
        for i in range(int(BottomIndent*CapBottom),
        int(TopIndent*CapTop)):
            for j in range(int(FrontIndent*CapFrontEdge),
            int(BackIndent*CapBackEdge+(SIZE-1)/10)):
                for k in range(int(LeftPlatePos+(SIZE)/10),
                int(RightPlatePos-2)):

                    Ey[i][j][k]=initCapField*np.exp(
                    -float(Maxtime)/10)
```

```
# This updates the magnetic field as a response to the
# discharging capacitor
for i in range(0,SIZE-1):
    for j in range(0,SIZE-1):
        for k in range(0,SIZE-1):

            # Don't update the field within the capacitor
            #if not(CapBottom < i < CapTop and
            # CapFrontEdge < j < CapBackEdge and
            # LeftPlatePos < k< RightPlatePos):
            Bx[i][j][k] = (Cbxb[i][j][k]*Bx[i][j][k]
            + Cbxe[i][j][k]*((Ey[i][j][k+1]-Ey[i][j][k])
            - (Ez[i][j+1][k]- Ez[i][j][k])))

            By[i][j][k] = (Cbyb[i][j][k]*By[i][j][k]
            + Cbye[i][j][k]*((Ez[i+1][j][k]-Ez[i][j][k])
            - (Ex[i][j][k+1]- Ex[i][j][k])))

            Bz[i][j][k] = (Cbzb[i][j][k]*Bz[i][j][k]
            + Cbze[i][j][k]*((Ex[i][j+1][k]-Ex[i][j][k])
            - (Ey[i+1][j][k]- Ey[i][j][k])))

#This updates the electric field as a response to the
#discharging capacitor
for i in range(1,SIZE):
    for j in range(1,SIZE):
        for k in range(1,SIZE):

            Ex[i][j][k] = (Cexe[i][j][k]*Ex[i][j][k]
            + Cexb[i][j][k]*((Bz[i][j][k] -Bz[i][j-1][k])
            - (By[i][j][k] - By[i][j][k-1])))

            Ey[i][j][k] = (Ceye[i][j][k]*Ey[i][j][k]
            + Ceyb[i][j][k]*((Bx[i][j][k] -Bx[i][j][k-1])
            - (Bz[i][j][k] - Bz[i-1][j][k])))

            Ez[i][j][k] = (Ceze[i][j][k]*Ez[i][j][k]
            + Cezb[i][j][k]*((By[i][j][k] -By[i-1][j][k])
            - (Bx[i][j][k] - Bx[i][j-1][k])))


#Attempt at absorbing boundary conditions for the edge
```

```
#of the cube

#This updates all three field components at the beginning
#and end of the x direction
for i in [0,SIZE]:
    for j in range(0,SIZE-1):
        for k in range(0,SIZE-1):
            if i==0:
                Ex[i][j][k]=(exOldXLeft[j][k]+((abcCoef
                -1)/(abcCoef+1)*(Ex[1][j][k] -Ex[0][j][k])))

                Ey[i][j][k]=(eyOldXLeft[j][k]+((abcCoef
                -1)/(abcCoef+1)*(Ey[1][j][k] -Ey[0][j][k])))

                Ez[i][j][k]=(ezOldXLeft[j][k]+((abcCoef
                -1)/(abcCoef+1)*(Ez[1][j][k] -Ez[0][j][k])))

                #This updates the old field at the boundary
                #to the new field at that point
                exOldXLeft[j][k] = Ex[1][j][k]
                eyOldXLeft[j][k] = Ey[1][j][k]
                ezOldXLeft[j][k] = Ez[1][j][k]

            elif i==SIZE-1:
                Ex[i][j][k] = (exOldXRight[j][k]+((abcCoef
                -1)/(abcCoef+1)*(Ex[SIZE-2][j][k] -
                Ex[SIZE-1][j][k])))

                Ey[i][j][k] = (eyOldXRight[j][k]+((abcCoef
                -1)/(abcCoef+1)*(Ey[SIZE-2][j][k] -
                Ey[SIZE-1][j][k])))

                Ez[i][j][k] = (ezOldXRight[j][k]+((abcCoef
                -1)/(abcCoef+1)*(Ez[SIZE-2][j][k] -
                Ez[SIZE-1][j][k])))

                # This updates the old field at the
                # boundary to the new field at that point
                exOldXRight[j][k] = Ex[SIZE-2][j][k]
                eyOldXRight[j][k] = Ey[SIZE-2][j][k]
                ezOldXRight[j][k] = Ez[SIZE-2][j][k]

# This updates all three field components at the beginning
```

```
# and end of the y direction
for j in [0,SIZE]:
    for i in range(0,SIZE-1):
        for k in range(0,SIZE-1):
            if j==0:
                Ex[i][j][k]=(exOldYLeft[j][k] + (abcCoef
                -1)/(abcCoef+1)*(Ex[i][1][k]-Ex[i][0][k]))

                Ey[i][j][k]=(eyOldYLeft[j][k] +(abcCoef
                -1)/(abcCoef+1)*(Ey[i][1][k] -Ey[i][0][k]))

                Ez[i][j][k]=(ezOldYLeft[j][k] +(abcCoef
                -1)/(abcCoef+1)*(Ez[i][1][k] -Ez[i][0][k]))

                #This updates the old field at the boundary
                #to the new field at that point
                exOldYLeft[i][k] = Ex[i][1][k]
                eyOldYLeft[i][k] = Ey[i][1][k]
                ezOldYLeft[i][k] = Ez[i][1][k]



            elif j==SIZE-1:
                Ex[i][j][k]=(exOldYLeft[j][k]+((abcCoef
                -1)/(abcCoef+1)*(Ex[i][SIZE-2][k]
                - Ex[i][SIZE-1][k])))

                Ey[i][j][k]=(eyOldYLeft[j][k]+((abcCoef
                -1)/(abcCoef+1)*(Ey[i][SIZE-2][k]
                - Ey[i][SIZE-1][k])))

                Ez[i][j][k]=(ezOldYLeft[j][k]+((abcCoef
                -1)/(abcCoef+1)*(Ez[i][SIZE-2][k]
                - Ez[i][SIZE-1][k])))

                #This updates the old field at the boundary
                #to the new field at that point
                exOldYLeft[i][k]= Ex[i][SIZE-2][k]
                eyOldYLeft[i][k]= Ey[i][SIZE-2][k]
                ezOldYLeft[i][k]= Ez[i][SIZE-2][k]

# This updates all three field components at the beginning
# and end of the z direction
```

```
        for k in [0,SIZE]:
            for i in range(0,SIZE-1):
                for j in range(0,SIZE-1):
                    if k==0:
                        Ex[i][j][k]=(exOldZLeft[i][j]+((abcCoef
                        -1)/(abcCoef+1)*(Ex[i][j][1]-Ex[i][j][0])))

                        Ey[i][j][k]=(eyOldZLeft[i][j]+((abcCoef
                        -1)/(abcCoef+1)*(Ey[i][j][1]-Ey[i][j][0])))

                        Ez[i][j][k]=(ezOldZLeft[i][j]+((abcCoef
                        -1)/(abcCoef+1)*(Ez[i][j][1]-Ez[i][j][0])))

                        #This updates the old field at the boundary
                        #to the new field at that point
                        exOldZLeft[i][j] = Ex[i][j][1]
                        eyOldZLeft[i][j] = Ey[i][j][1]
                        ezOldZLeft[i][j] = Ez[i][j][1]


                    elif k==SIZE-1:
                        Ex[i][j][k]=(exOldZLeft[i][j] + ((abcCoef
                        -1)/(abcCoef+1)*(Ex[i][j][SIZE-2]
                        - Ex[i][j][SIZE-1])))

                        Ey[i][j][k]=(eyOldZLeft[i][j] + ((abcCoef
                        -1)/(abcCoef+1)*(Ey[i][j][SIZE-2]
                        - Ey[i][j][SIZE-1])))

                        Ez[i][j][k]=(ezOldZLeft[i][j] + ((abcCoef
                        -1)/(abcCoef+1)*(Ez[i][j][SIZE-2]
                        - Ez[i][j][SIZE-1])))

                        #This updates the old field at the boundary
                        #to the new field at that point
                        exOldYLeft[i][j] = Ex[i][j][SIZE-2]
                        eyOldYLeft[i][j] = Ey[i][j][SIZE-2]
                        ezOldYLeft[i][j] = Ez[i][j][SIZE-2]



########################################################################
# This is where I use Gauss's law to calcualte the charge density
```

```
# of the system by taking the divergence of the electric field.
for i in range(1,SIZE-1):
    for j in range(1,SIZE-1):
        for k in range(1,SIZE-1):
            qDen[i][j][k]= (((float(1)/(2*spaceStep))*eps[i][j][k]
            *((Ex[i+1][j][k] - Ex[i-1][j][k]) + (Ey[i][j+1][k]
            - Ey[i][j-1][k]) + (Ez[i][j][k+1] - Ez[i][j][k-1]))
            )*(1*10**(-12)))
```

```
# File name: Color_Plot.py

import time
start_time = time.clock()


# Import all necessary libraries and files
import numpy as np
import matplotlib.pyplot as plt
import pylab as py # matplotlib

from System_Parameters import *
from Get_Wire_Shape import *
from Circuit_Initialization import *
from Update_Equations import *


# This closes the existing plot, which, in this case, is the
# initialization grid
py.close()

######################################################################
# The tuple which holds rgb values
chargeColor = (0,0,0)

# New array to store the scaled version of q
scaledLogQ = np.zeros((SIZE,SIZE))


######################################################################


# Take the log of all the numbers in qDen. Note that the cross-
# section used is the *top* of the wire. This is because we want
# to look at *surface* charge
for i in range(SIZE):
    for j in range(SIZE):
        # This avoids misasigning such entries to enormous values.
        # Instead they are approximated as zero
        if -1<= qDen[WireTop][i][j] <=1:
            scaledLogQ[i][j] = 0
        elif qDen[WireTop][i][j] < -1:
            scaledLogQ[i][j] = -1*np.log(abs(qDen[WireTop][i][j]))
```

```
        elif qDen[WireTop][i][j] > 1:
            scaledLogQ[i][j] = np.log(qDen[WireTop][i][j])

# These are the max and min values used to resize each element
# in scaledLogQ to be bewtween 0 and 1.
Max = np.amax(scaledLogQ)
Min = np.amin(scaledLogQ)


# Divide everything by the maximum value so that every entry is
# between -1 and 1.
for i in range(SIZE):
    for j in range(SIZE):
        if scaledLogQ[i][j] <0:
            scaledLogQ[i][j] = (float(scaledLogQ[i][j])/abs(Min))
        elif scaledLogQ[i][j] >0:
            scaledLogQ[i][j] = (float(scaledLogQ[i][j])/Max)



# This makes a color plot of scaleQ
for i in range(SIZE):
    for j in range(SIZE):

        # If the charge is negative plot it as blue and
        # if the charge is positive plot it as red. In
        # both cases, add some green.
        if scaledLogQ[i][j] < 0:
            chargeColor = (0, 0.75, abs(scaledLogQ[i][j]))
        elif scaledLogQ[i][j] > 0:
            chargeColor = (abs(scaledLogQ[i][j]), 0.75,0)
        else:
            chargeColor = (0.9,0.9,0.9)

        #Shade the pixel if it corresponds the edge of the wire
        if MaterialConduc<=MaterialConduc*(ScaledNewCircuit[i+1][j]
            + ScaledNewCircuit[i][j+1] + ScaledNewCircuit[i+1][j+1]
            + ScaledNewCircuit[i][j+1] ) < 4*MaterialConduc:
            chargeColor = (0.4,0.4,0.4)

        # Plot a dot with the appropriate color at the point (i,j)
        py.scatter(i,j,color= chargeColor)
```

References

[1] The engineering toolbox. http://www.engineeringtoolbox.com/resistivity-conductivity-d_418.html.

[2] Magnetic susceptibilities of paramagnetic and diamagnetic materials at 20 degrees celcius. http://hyperphysics.phy-astr.gsu.edu/hbase/tables/magprop.html.

[3] A. K. T. Assis and A. J. Mania. Surface charges and electric field in a two-wire resistive transmission line. *Revista Brasileira de Ensino de Fisica*, 21(4), 1999.

[4] Jean-Pierre Berenger. Three-dimensional perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics*, 127(0181):363–379, 1996.

[5] John Broskie. Loudspeaker cables. http://www.tubecad.com/2012/09/blog0242.htm, September 2012.

[6] Richard Fitzpatrick. Helmholtz's theorem. http://farside.ph.utexas.edu/teaching/em/lectures/node37.html.

[7] Griffiths and Partovi. Equilibrium charge density on a thin curved wire. *American Journal of Physics*, 77(1173), 2009.

[8] David Griffiths. *Introduction to Electrodynamics, 4th edition*. Pearson, 2013.

[9] R. Victor Jones. Drude model. http://people.seas.harvard.edu/~jones/es154/lectures/lecture_2/drude_model/drude_model.html.

[10] Tom Moore. *Electric and Magnetic fields are Unified*. McGraw-Hill.

[11] Mueller. A semiquantitative treatment of surface charges in dc circuits. *American Journal of Physics*, 80(782), 2012.

[12] Preyer. Surface charges and fields of simple circuits. *American Journal of Physics*, 68(1002), 2000.

[13] John B. Schneider. *Understanding the Finite-Difference Time-Domain Method*. June 22, 2015.

[14] VEGA. Dielectric constants list. `http://www.vega-americas.com/downloads/Forms-Certificates/Dielectric_Constants_List.pdf`.