PML for a General Wave Equation

Absorbing boundary conditions for generic waves with an eye towards computational complexity

Abstract

Expanding on some previous work, we investigate how we can add absorbing boundary conditions to a computational domain so that we can pretend our waves propagate from somewhere in the middle and out towards infinity. We discuss a few different ways we can augment our simulations to accomplish this with one specific approach, the perfectly matched layer (PML). MATLAB implementations for each method are provided in the appendices.

Coming from an electrical engineer, you may find the following statement shocking: all of the interesting differential equations come from quantum mechanics. Oh yes, Maxwell's equations are infinitely useful, but c'mon, not even a bucket full of lasers will ever be as interesting as the necessity for a wave equation to be complex-valued. Seriously, read the first ninety pages of [1], with special attention paid to chapter four, section five, and I'm sure you'll agree.

Now, when presented with a partial differential equation, of course the first thing one does is type it into the computer and watch how things bounce around. When I say bounce around, I mean that literally. As quantum wavefunctions—indeed any wave-like thing—move around in a computational domain where the boundary conditions are, say, set to be zero by default, they'll reflect off of the sides (a fun consequence of discontinuities and second derivatives). Since there aren't any sources or sinks in the fundamental equations of quantum mechanics, probability and energy tend to stay conserved in such a situation: a good thing.

However, even in the presence of potentials, if your initial wavefunction spreads out enough it eventually devolves into a bunch of oscillating modes dominated by the modes of the computational domain itself (usually a square). Meaning, if you're trying to explore the modes of a particular scalar potential—not the modes of your computer—you need to find a way to attenuate your wavefunction before it gets to the edge of the computational domain. For some potentials, like the harmonic oscillator, this isn't usually necessary, but for others it is. While not necessarily obvious, in order to do this, if we stare at Schrodinger's equation,

$$j\hbar\frac{\partial u}{\partial t} = -\frac{\hbar^2}{2m}\nabla^2 u + V(\mathbf{x})$$

for too long and realize that the easiest way to do that is to add a negative, purely imaginary potential that exists near the edges of the computational domain. That way as your wavefunction moves outwards towards the edges of your simulation and overlaps, or interacts, with this extra potential, it will attenuate exponentially based on its strength. This is most useful when your wavefunction can't be completely captured by the potential, such as with the standard electric potential, Morse potential or Lennard-Jones potential.

Now, to take things to the next level, you might want to model how wavefunctions also interact with scalar or vector potentials (the same as with Maxwell's equations in Lorenz gauge). To do that, we look at Pauli's equation:

$$j\hbar\frac{\partial \psi}{\partial t} = \left[\frac{1}{2m}\left(-\hbar^2\nabla^2 + q^2\mathbf{A}^2 + j\hbar q\nabla\cdot\mathbf{A} + j\hbar q\mathbf{A}\cdot\nabla - q\hbar\sigma\cdot\mathbf{B}\right) + V\right]\psi$$

Which is a bit of an eyeful, truly. However, if you think about it, everything here is built in to, and optimized for, the computer. I prefer to use MATLAB for computer things, but if your code is using anything that gets down to the Intel Math Kernel Library (MKL), or LAPACK, or BLAS, or OpenBLAS, then you'll probably have access to some hand-crafted algorithms that can easily manage matrix operations, divergences, gradients; all that fun stuff.

Ok fine, but: how do you generate the potentials *V* or **A**? To do that, we take some inspiration from Quantum Electrodynamics[1] to figure out how to build charges or currents from wavefunctions. At this point, in Lorenz gauge, we're just left with four wave equations. One for the scalar potential, V, and three that cover the three components on the vector potential, **A**. Each one more or less taking on the form:

$$\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} = \nabla^2 u + \phi$$

Where *u* is just a stand in for whatever potential. While this isn't the most general form of a hyperbolic partial differential wave equation, this is what we're going to focus on because that's as complicated as we need for now. Thankfully, the physicists know how to solve these things for us given that the wave equation in free space is effectively an LTI system. So, all we do is open up [2], and read off the Green's function (or impulse response or fundamental solution (or propagator)) of the wave equation, to get us a general, three-dimensional solution of:

$$u(\mathbf{x},t) = \frac{\delta\left(t - \frac{|\mathbf{x}|}{c}\right)}{4\pi|\mathbf{x}|} * \phi$$

And that's nice.

This does tell us a couple of things that can help us. First off, we can just pick an arbitrary source charge distribution and convolve it with the fundamental solution to watch how it evolves in time. We don't have to put the differential equations in the computer after all. Except of course convolution is insanely slow, especially in 3D, and just typing the letters `convn` into the MATLAB command window gets the computer fans spinning full tilt. As an aside, the fundamental solution to the wave equation in two dimensions is impossible to work with in this same way.

Second, it makes this obvious that all of our potentials are going to be radiating away from whatever source we have in our computational domain, so we'll definitely have issues as they bounce off the edges. Indeed, even if our sources aren't constant, you will get cool interference effects as things travel back at you with the opposite phase—eventually devolving into a bunch of weird oscillating modes of your computational domain. This is bad! We need to figure out a way to make our potentials attenuate down to zero as they approach our boundary, so that what goes on within our computational domain is a good approximation of what would go on in free space, where the domain is mostly infinite.

Neato, but now what? To get started, we do a lot of reading. Most references on solving partial differential equations, e.g. [5], don't really talk much about absorbing boundary conditions, just about the standard stuff like Dirichlet, Neumann, Cauchy, or Robin. Then you switch gears a bit, read [3] in its entirety, and then get sad. It turns out a lot of the material out there is focused on absorbing electromagnetic waves, but in terms of electric and magnetic *fields,* not the potentials. There's an entire industry surrounding the finite-difference time-domain method, which can take a number of different approaches to absorbing boundary conditions, including that of a perfectly matched layer (PML), or figuring out how to stick some sort of absorbing or conducting material wherever you want the fields to die out.

Then you start to search around the internet, discover the open source software package Meep[2], and deep in their documentation on PMLs they send you over to [4] to read about a good general approach. That's what the rest of this is going to be, following that paper, except we'll go right to the end of a two-dimensional simulation, and make a few adjustments to make things stable and fast.

[1] https://en.wikipedia.org/wiki/Quantum_electrodynamics#Equations_of_motion
[2] https://github.com/NanoComp/meep

For a full introduction, please go ahead and read through [4], I'll just touch on some of the initial stuff so that the rest of the thinking here makes sense. We know from years of intense study, confirmed by the general analytic solution above, that that the wave equation in free space is linear. This means that in the absence of boundary conditions, we can construct solutions from a linear combination of plane waves, such that:

$$u(\mathbf{x},t) = \iint U(\mathbf{k},\omega)\, \mathrm{e}^{j(\mathbf{k}\cdot\mathbf{x}-\omega t)} \mathrm{d}\omega\, \mathrm{d}\mathbf{k}$$

Where *U(***k***, w)* is the Fourier transform of *u(***x***, t)*. Staring at this for a while, we realize that if we want it to attenuate at all, there's not much we can do by way of modifying the **k**s or the *w*, as that would upset the overall balance of things, and the inverse Fourier transform wouldn't work as planned. However, if we make a change to the **x**s or the *t*, then at a specific time, or within a specific region, we can get an exponential attenuation as the wave advances into that spatial region, or as it marches ahead past a specific point in time. The way to do this would be to modify either to have an imaginary component at some point. That way you'd have a real and imaginary part to the exponent, and if the real part is negative, you get exponential attenuation.

Focusing on time for a moment, there is another way to do this is, and that's literally to just start changing values in the equations after you've plugged them into the computer. For example, if we discretize the wave equation in one dimension, and re-arrange things:

$$u^{t+1} = 2d\, u^t - d^2\, u^{t-1} + s(\nabla^2 u + \phi)$$

Where *u*, let's say, is a 3D field representing our potential just to test things out, phi is our source charge distribution, $s = \mathrm{d}t^2\, c^2$, and we're doing a basic per-step integration with the Forward Euler, or explicit, method. Here *d* is just a real-valued constant that we use to play around with. If we set it to be anything less than one, we get an immediate exponential decay *across the entire function*. Meaning that if we were to turn it on at a certain point in time, after that the entire wave would decay exponentially, eventually reaching a steady state balanced by the creation of a potential by the source, phi.

This approach though, since it effects the entire function *u* at all space (you know, due to the nature of time), eventually affects the physics we're trying to simulate above: a proper interaction of potentials with wavefunctions. The next step is to take a look at what we can do with the **x** coordinates. Similarly, if we introduce an imaginary component, we can engineer things in such a way that there is an exponential attenuation in certain regions of our computational domain. That is, we make a coordinate transformation (some say analytic continuation) such that, at least in 1D:

$$x \rightarrow x + j\,\frac{f(x)}{\omega}$$

Where we have complete control over where *f(x)* is defined to be not zero. This means that our original solution for *u* can be described as, in general:

$$u(\mathbf{x},t) \rightarrow \iint \mathrm{e}^{-\omega^{-1}\mathbf{k}\cdot\mathbf{f}(\mathbf{x})}\, U(\mathbf{k},\omega)\, \mathrm{e}^{j(\mathbf{k}\cdot\mathbf{x}-\omega t)}\, \mathrm{d}\omega\, \mathrm{d}\mathbf{k}$$

Since we've already stated that we're operating in free space, we can make this simple, by realizing that the phase velocity, *c*, of our wave is just the frequency divided by the wavenumber, allowing us to simplify things such that:

$$u(\mathbf{x},t) \rightarrow \mathrm{e}^{-c^{-1}\mathbf{f}(\mathbf{x})} \iint U(\mathbf{k},\omega)\, \mathrm{e}^{j(\mathbf{k}\cdot\mathbf{x}-\omega t)}\, \mathrm{d}\omega\, \mathrm{d}\mathbf{k}$$

Which of course is just:

$$u(\mathbf{x},t) \rightarrow \mathrm{e}^{-c^{-1}\mathbf{f}(\mathbf{x})} u(\mathbf{x},t)$$

Meaning that if we can find a decent way to define *f(x)*, we can accomplish our goal: attenuate a wave in some arbitrary region. Or rather, close to the boundary of our computational domain in order to prevent reflections. Now all we have to do is figure out how to put this into our differential equation so that we can plug it into our computer.

Before we get started, we just have to remember two things:

$$\frac{\mathrm{d}u}{\mathrm{d}t} = -j\omega u$$

$$\int_0^t u\, \mathrm{d}t = \frac{j}{\omega} u$$

Which will help us simplify our math as we go along. You know, make a few transformations to/from the time-domain so that we just end up doing fancy algebra. From here on out, we'll be following the logic of [4], and then extending it to fully flesh out the 2D situation. The full 3D solution is left as an exercise for the reader. To keep the notation simple in certain points, we'll be holding on to the 3D vector calculus symbols, but please keep in mind there will only ever be derivatives in terms of *x*, *y*, and *t*. Then, as we go to put things in to the computer we can get away with this for the simple reason that MATLAB has gradient and divergence functions defined for arbitrary dimensions.

The first step, is to take the derivative of our 1D coordinate transform so that we can figure out how our derivatives might transform. That is:

$$\partial x \rightarrow \left(1 + j\frac{1}{\omega}\frac{\partial f}{\partial x}\right)\partial x$$

Where also for the sake of simplicity going forward, we redefine *f(x)* such that:

$$\frac{1}{\omega}\frac{\partial f}{\partial x} = \frac{\sigma_x}{\omega}$$

Which then means we're going to go ahead and do everything in rectangular coordinates, so that we transform all of our vector calculus derivatives to be something along the lines of:

$$\frac{\partial}{\partial x} \rightarrow \frac{1}{1 + j\frac{\sigma_x}{\omega}}\frac{\partial}{\partial x}$$

Now, starting with our original wave equation:

$$\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} = \nabla^2 u + \phi$$

We can make a relatively straightforward substitution into two separate equations such that:

$$\frac{1}{c^2}\frac{\partial u}{\partial t} = \nabla \cdot \mathbf{v} + \int_0^t \phi \, \mathrm{d}t$$

$$\frac{\partial \mathbf{v}}{\partial t} = \nabla u$$

Which may look strange, but if you take the time derivative of the first, then substitute the second one back in, you'll get your original wave equation. So now we focus our attention on **v**. In two dimensions, this means that we'll have the time derivative of the x-component of the vector **v** equal to the x-derivative of our wave, *u*, such that:

$$\frac{\partial v_x}{\partial t} = \frac{\partial u}{\partial x}$$

If we then bring this to the frequency domain by way of a straight up Fourier Transform, while also making the substitution for the x-derivative as before, we get:

$$-j\omega v_x = \frac{1}{1 + j\frac{\sigma_x}{\omega}}\frac{\partial u}{\partial x}$$

If we then multiply both sides by the complex denominator that we've added to our derivative, we get:

$$-j\omega v_x \left(1 + j\frac{\sigma_x}{\omega}\right) = \frac{\partial u}{\partial x}$$

And then multiply things out, re-arranging so that all of the imaginary parts are on the left-hand side of our equation we're left with:

$$-j\omega v_x = \frac{\partial u}{\partial x} - v_x \sigma_x$$

Which of course means that when we bring things back into the time-domain, we get another handy differential equation that now takes into account a region where we may want our wave to attenuate:

$$\frac{\partial v_x}{\partial t} = \frac{\partial u}{\partial x} - v_x \sigma_x$$

If we follow the same procedure with the y-component of **v** and the y-derivative of *u*, and then collapse things back into convenient vector calculus notation (even if we're still only in 2D), we can see that:

$$\frac{\partial \mathbf{v}}{\partial t} = \nabla u - \mathbf{v} \odot \sigma$$

Where the dot with the circle around it is the Hadamard, or element-wise, product. All that does is multiply similar elements of two given vectors, e.g. the x-component of the output vector consists of all x-components of the inputs multiplied together. Especially when putting things into the computer, this really is just a fancy way of writing down multiple equations all in one go.

As we can see here, our simple gradient is transformed into something slightly more complicated, but please keep in mind that sigma will only be defined for our region where we want our wave to attenuate. This means that *for the most part*, sigma isn't really doing anything. This is just a mild correction to help us accomplish

our goals in one very specific part of our computational domain. Indeed, if we do it right, it won't even take up much space in our computer at all. We'll discuss this in a little bit.

Now, let's see what we can do with that modified wave equation:

$$\frac{1}{c^2}\frac{\partial u}{\partial t} = \nabla \cdot \mathbf{v} + \int_0^t \phi \, \mathrm{d}t$$

So first, let's move everything over to the frequency domain by way of a Fourier Transform so that we get:

$$-\frac{1}{c^2} j\omega u = \nabla \cdot \mathbf{v} + \frac{j}{\omega}\phi$$

Then we'll expand out our divergence, also making the appropriate substitutions for both the x- and y-derivatives, such that:

$$-\frac{1}{c^2} j\omega u = \frac{1}{1 + j\frac{\sigma_x}{\omega}}\frac{\partial v_x}{\partial x} + \frac{1}{1 + j\frac{\sigma_y}{\omega}}\frac{\partial v_y}{\partial y} + \frac{j}{\omega}\phi$$

Multiplying both sides by both complex denominators then gets us a headache:

$$-\frac{1}{c^2} j\omega u \left(1 + j\frac{\sigma_x}{\omega}\right)\left(1 + j\frac{\sigma_y}{\omega}\right) = \frac{\partial v_x}{\partial x}\left(1 + j\frac{\sigma_y}{\omega}\right) + \frac{\partial v_y}{\partial y}\left(1 + j\frac{\sigma_x}{\omega}\right) + \frac{j}{\omega}\phi\left(1 + j\frac{\sigma_x}{\omega}\right)\left(1 + j\frac{\sigma_y}{\omega}\right)$$

Which we can actually simplify by recognizing, *and requiring*, that there will be no overlap between our source charges and the region where our wave attenuates. That is, if we can guarantee that our source charges stay far away from our boundaries, well-behaved near the center of our computational domain, then we can assume that any sigmas multiplied by any phis will just be zero. When we do this in the computer, this is relatively straightforward—if a bit of a pain—as we can always just have a larger computational domain for our potentials than our wavefunctions / charges.

This lets us do an easy simplification so now we have:

$$-\frac{1}{c^2} j\omega u \left(1 + j\frac{\sigma_x}{\omega}\right)\left(1 + j\frac{\sigma_y}{\omega}\right) = \frac{\partial v_x}{\partial x}\left(1 + j\frac{\sigma_y}{\omega}\right) + \frac{\partial v_y}{\partial y}\left(1 + j\frac{\sigma_x}{\omega}\right) + \frac{j}{\omega}\phi$$

We can now expand out our brackets, collecting terms on both the right-hand and left-hand sides of our equation. This gives us:

$$-\frac{1}{c^2} j\omega u + \frac{1}{c^2}(\sigma_x + \sigma_y)u + \frac{j}{c^2\omega}\sigma_x\sigma_y u = \nabla \cdot \mathbf{v} + \frac{j}{\omega}\left(\sigma_y\frac{\partial v_x}{\partial x} + \sigma_x\frac{\partial v_y}{\partial y} + \phi\right)$$

Where we've also collapsed straight up derivatives of **v** back into a divergence. if we now re-organize things so that we isolate what will become our first order time derivative to only be on the left-hand side of the equation, again collecting terms we can see that:

$$-\frac{1}{c^2} j\omega u = \nabla \cdot \mathbf{v} - \frac{1}{c^2}(\sigma_x + \sigma_y)u + \frac{j}{\omega}\left(\sigma_y\frac{\partial v_x}{\partial x} + \sigma_x\frac{\partial v_y}{\partial y} - \frac{1}{c^2}\sigma_x\sigma_y u + \phi\right)$$

In order to help us better see what's going on, and then eventually put this into the computer, we now make a substitution, whereby we define:

$$\psi = \frac{j}{\omega}\left(\sigma_y \frac{\partial v_x}{\partial x} + \sigma_x \frac{\partial v_y}{\partial y} - \frac{1}{c^2}\sigma_x \sigma_y u + \phi\right)$$

Meaning that if we divide both sides by *j/w* we can get an equation that is:

$$-j\omega\psi = \sigma_y \frac{\partial v_x}{\partial x} + \sigma_x \frac{\partial v_y}{\partial y} - \frac{1}{c^2}\sigma_x \sigma_y u + \phi$$

Which when we bring back into the time domain, gives us:

$$\frac{\partial \psi}{\partial t} = \sigma_y \frac{\partial v_x}{\partial x} + \sigma_x \frac{\partial v_y}{\partial y} - \frac{1}{c^2}\sigma_x \sigma_y u + \phi$$

… another differential equation we have to solve! Oh brother. In two dimensions, in order to allow our potential to attenuate as it radiates away from a source and as it approaches the boundary of our computational domain, we now have to solve three differential equations all at once. My preferred way to do it, would be to set up the initial conditions—fully defining sigmas and phi, everything else starting at zero—and then march through time with the following steps:

1) At each time step, I would first take the gradient of *u*, then use a straight up explicit method (Forward Euler with forward differences in time is easiest) to solve for both components of **v** (v_x and v_y) using:

$$\frac{\partial \mathbf{v}}{\partial t} = \nabla u - \mathbf{v} \odot \sigma$$

2) We then use these values to update the next step for psi and *u*. But first we have to find the x-derivative of v_x, then the y-derivative of v_y. Then we can again use a straight up explicit method to solve for the current value of psi:

$$\frac{\partial \psi}{\partial t} = \sigma_y \frac{\partial v_x}{\partial x} + \sigma_x \frac{\partial v_y}{\partial y} - \frac{1}{c^2}\sigma_x \sigma_y u + \phi$$

3) The last thing to do is then take the current value of psi, the current values of the derivatives of **v**, and the current value of our wave, *u*, to explicitly step forward in time once more using:

$$\frac{1}{c^2}\frac{\partial u}{\partial t} = \nabla \cdot \mathbf{v} - \frac{1}{c^2}(\sigma_x + \sigma_y)u + \psi$$

And that's it! This works! Please see some MATLAB code in Appendix A that goes through this.

If you think through it, when compared to integrating a straight up wave equation there's not that much additional overhead here. Sure, there's some extra calculation with gradients instead of a single Laplacian, but in terms of space requirements there's not much additional memory required. Why? Because the sigmas can be represented (in 2D) by matrices that are mostly zero. This means that if you represent them as sparse matrices in MATLAB, they only take up space where elements are non-zero (unlike full matrices), and they only perform calculations where elements are non-zero (unlike full matrices). In practical terms, if our PML or absorbing boundary represented by the sigmas is less than say, 5% of our total computational domain, then

the only large additional space requirements added are by the intermediary values represented by the derivatives of *u,* **v** and its derivatives, and psi.

But... there is at least one problem with it. Don't get me wrong, for small problems, you can really quickly jam this into your computer, leave it as is, and get about your business. But what if you want to make your computational domain bigger, or have better resolution, or have your waves move faster? Well, at that point you'll realize that the problem is:

1) This approach is not very stable.

As we know from [5], chapter 5, a straight up explicit Euler method for a parabolic partial differential equation (like above) is half as stable (and accurate) when compared to a hyperbolic partial differential equation (like the wave equation), requiring a time step at least half as small. Also, since the stability of the system is determined by the distance between grid points—and our PML adds distance via an imaginary part—the overall design of our PML needs to take this into account. That is, the relative values of each of our sigmas need to be small enough to account for the stability of the system as a whole.

So, is there a way to make things slightly more stable? Yes, and that's to reformulate things so that at least part of it is a second order differential equation. What that means is that we'll still have our first intermediary or auxiliary differential equation:

$$\frac{\partial \mathbf{v}}{\partial t} = \nabla u - \mathbf{v} \odot \sigma$$

But instead of needing to solve a third partial differential equation, i.e. psi, we go straight ahead with:

$$\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} = \nabla \cdot \frac{\partial \mathbf{v}}{\partial t} + \phi$$

As before, we expand out the divergence, making substitutions for our derivatives to include the PML, while at the same time bringing everything to the frequency domain so that we get:

$$-\frac{1}{c^2}\omega^2 u = \frac{1}{1 + j\frac{\sigma_x}{\omega}}\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x} - \sigma_x v_x\right) + \frac{1}{1 + j\frac{\sigma_y}{\omega}}\frac{\partial}{\partial y}\left(\frac{\partial u}{\partial y} - \sigma_y v_y\right) + \phi$$

Now if we multiply everything by both of the complex denominators, following the same logic as above, and remembering that our source, phi, should never overlap with the PML represented by the sigmas, we're left with:

$$-\frac{1}{c^2}\omega^2 u\left(1 + j\frac{\sigma_x}{\omega}\right)\left(1 + j\frac{\sigma_y}{\omega}\right) = \nabla^2 u - \nabla \cdot (\mathbf{v} \odot \sigma) + \phi$$

Expanding everything and collecting terms gets us:

$$-\frac{\omega^2}{c^2}u - j\frac{\omega}{c^2}(\sigma_x + \sigma_y)u + \frac{1}{c^2}\sigma_x\sigma_y u = \nabla^2 u - \nabla \cdot (\mathbf{v} \odot \sigma) + \phi$$

And then if we bring everything back into the time-domain, we're left with:

$$\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} + \frac{1}{c^2}(\sigma_x + \sigma_y)\frac{\partial u}{\partial t} + \frac{1}{c^2}\sigma_x\sigma_y u = \nabla^2 u - \nabla \cdot (\mathbf{v} \odot \sigma) + \phi$$

Similar to the above procedure, we would follow the following steps to simulate our potential:

1) At each time step, I would first take the gradient of *u*, then use a straight up explicit method (Forward Euler with forward differences in time is easiest) to solve for both components of **v** (v_x and v_y) using:

$$\frac{\partial \mathbf{v}}{\partial t} = \nabla u - \mathbf{v} \odot \sigma$$

2) We use the intermediary values we found for **v** and use them to solve for the next step in time for *u:*

$$\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} + \frac{1}{c^2}(\sigma_x + \sigma_y)\frac{\partial u}{\partial t} + \frac{1}{c^2}\sigma_x\sigma_y u = \nabla^2 u - \nabla \cdot (\mathbf{v} \odot \sigma) + \phi$$

This also eliminates one step, and the additional space required by psi. While it's not obvious here—even though this is more stable (and accurate) than the first order solution—this actually runs much slower than the previous method. Depending on your setup, about 50% - 100% slower than you'd expect. The reason can more readily be seen if we fully expand things out using central differences for the time derivatives such that:

$$\frac{\partial^2 u}{\partial t^2} = \frac{1}{\Delta t^2}\left(u^{t+1} - 2u^t + u^{t-1}\right)$$

$$\frac{\partial u}{\partial t} = \frac{1}{2\Delta t}\left(u^{t+1} - u^{t-1}\right)$$

And then collect like terms after using just a basic explicit, Forward Euler method:

$$\left(1 + \frac{\Delta t}{2}(\sigma_x + \sigma_y)\right)u^{t+1} + \left(\sigma_x\sigma_y(\Delta t)^2 - 2\right)u^t + \left(1 - \frac{\Delta t}{2}(\sigma_x + \sigma_y)\right)u^{t-1} = (\Delta t)^2 c^2\left(\nabla^2 u - \nabla \cdot (\mathbf{v} \odot \sigma) + \phi\right)$$

Re-arranging this to solve for u^{t+1}, we can iterate through time to update our potential, without anything too complicated. Please refer to Appendix B for some MATLAB code that goes through an example implementation.

In terms of computational efficiency, the reason that things slow down is that we're no longer taking advantage of the time savings of multiplying things by sparse matrices. That is, wherever a combination of sigmas appears, we're adding or subtracting a constant value, converting them to full matrices before multiplying by current or previous versions of the wave (i.e. u^t, u^{t-1}) to solve for the next wave at the next time step, u^{t+1}. Since the sigmas never change with time, we can pre-compute some of these matrices for a marginal speedup, but at the cost of additional memory.

Is there any way we can get better stability and also better performance? First, let's look at our three first order equations again:

$$\frac{\partial \mathbf{v}}{\partial t} = \nabla u - \mathbf{v} \odot \sigma$$

$$\frac{\partial \psi}{\partial t} = \sigma_y\frac{\partial v_x}{\partial x} + \sigma_x\frac{\partial v_y}{\partial y} - \frac{1}{c^2}\sigma_x\sigma_y u + \phi$$

$$\frac{1}{c^2}\frac{\partial u}{\partial t} = \nabla \cdot \mathbf{v} - \frac{1}{c^2}(\sigma_x + \sigma_y)u + \psi$$

Normally if you're concerned about stability you flip things over from using explicit methods to implicit methods. But looking above, where can we insert implicit methods into these equations? Looking at the differential equation that describes the evolution of psi, it's not even possible. Why? Because there is no psi on the right-hand side of the equation, the implicit and explicit method would be identical. Ideally, we'd want to describe *u* implicitly throughout the full integration, but we can't for that same reason. That is, looking at the equation that describes the time evolution of **v**, we can create an implicit form of this for each component of **v**, but not one for *u*. However, we *can* follow through an implicit formulation of the third equation, the one that finally describes the evolution of *u* itself; simply because it has *u* on both sides of the differential equation.

To do this, we'll first pick a Crank-Nicolson scheme since when solving a regular PDE it gives us better accuracy. When we use forward differences in time, discretization gives us an equation that look like:

$$\frac{1}{c^2}\frac{u^{t+1}-u^t}{\Delta t} = \nabla\cdot\mathbf{v} - \frac{1}{2c^2}(\sigma_x+\sigma_y)(u^{t+1}+u^t) + \psi$$

The interesting thing to note here is that we don't have to worry at all about spatial discretization since when it comes to *u*, that's handled by one of our other differential equations, namely that for **v**. This also means that when it comes to solving things, we don't have to worry about linear algebra at all, since our system of equations forms a series of problems that are solely diagonal. No penta-diagonal or two-step tridiagonal solvers are needed here; we just use regular division. This means that we can then re-arrange everything such that:

$$\left(1+\frac{\Delta t}{2}(\sigma_x+\sigma_y)\right)u^{t+1} = \left(1-\frac{\Delta t}{2}(\sigma_x+\sigma_y)\right)u^t + c^2\Delta t(\nabla\cdot\mathbf{v}+\psi)$$

Which allows us to solve for u^{t+1} in a relatively straightforward fashion. If we take a similar approach to solving **v**—since it has **v** on both sides of the equation—then we get two semi-implicit steps; with psi still being solved explicitly. Leaving the differential equation for psi as it is, this would mean that the differential equation for each component of **v** would be, e.g.:

$$\left(1+\frac{\Delta t}{2}\sigma_x\right)v_x^{t+1} = \Delta t\frac{\partial u}{\partial x} + \left(1-\frac{\Delta t}{2}\sigma_x\right)v_x^t$$

And similarly, for v_y. Now, even though we've solved two of the three steps implicitly, we can't make any claims to absolute stability. I wouldn't even know how to go through the stability analysis for this. But, if we play around with the code itself, we will notice that it seems more stable than the second-order approach above, but also performs just as well as the three step first-order approach.

Please see Appendix C for some MATLAB code that works. Playing around with a certain set of problems, I was able to get a stable time step about 15% higher than that of the second-order approach. For others, it was approximately as stable as the second-order approach. For all cases, the per-step, single-pass calculation time was almost equivalent to the first-order approach. In order to save time on calculation, we've pre-calculated all of the constants (i.e. everything with sigmas in the brackets above). However, if you want to save memory as your computational domain increases, you can trade out memory for computation by realizing that some of the constants are the same, just rotated by 90 degrees, so you can calculate them on the fly if needed using `rot90` (faster than transposing in this case).

Now, switching gears a bit, if you take a look at all of the equations and code so far, we never actually defined any of the parameters that define the perfectly matched layer: the sigmas. So, how do we go about designing something that works? First, we go back to the start, and look at how our wave or potential or whatever *u* represents is defined in the presence of absorbing boundary conditions:

$$u(\mathbf{x},t) \rightarrow \mathrm{e}^{-c^{-1}\mathbf{f}(\mathbf{x})}u(\mathbf{x},t)$$

Where *c* is the free space velocity of your waves, and *f* is a function of space that defines how your potential, *u*, attenuates as it enters into a region upon which *f* is defined to be something other than zero. What we want is for our potential to attenuate as it traverses the PML so that before it can reflect off the side of the computational domain, it's been reduced to some arbitrary small value, say *R*:

$$\exp\left\{-c^{-1}f(x)\right\} = R$$

But then we remember how we previously defined our sigmas in terms of the derivative of *f*:

$$\frac{\partial f}{\partial x} = \sigma_x(x)$$

So that we can re-write things in terms of sigma, first taking the natural log of things, giving us:

$$\int_{x_0}^{x_{max}} \sigma_x(x)\,\mathrm{d}x = -c\,\log(R)$$

We're now left with a choice. How should we define sigma so that it's useful? In order to minimize reflections at the beginning of the PML, we want it to slowly turn on and then increase increasingly quickly so that we can get the amount of attenuation that we want. There are a couple of ways to do this, but one easy way would be with a simple polynomial, that is:

$$\sigma_x(x) = \sigma_{max}(x - x_0)^m$$

With x_0 the point at which the PML turns on, sigma_{max} the maximum value needed to get the correct attenuation, and *m* the order of the polynomial. If we then follow through with the above integration, setting all integration constants to zero to eliminate spurious reflection right at the PML boundary, we get:

$$\frac{\sigma_{max}}{m+1}(x - x_0)^{m+1}\Big|_{x_0}^{x_{max}} = -c\,\log(R)$$

Then if we substitute everything in, we can figure out what sigma_{max} should be:

$$\sigma_{max} = -\frac{c\,\log(R)(m+1)}{(x_{max} - x_0)^{m+1}}$$

And then to make things simple, we can also make the substitution:

$$\text{PML width} = n\,\Delta x = x_{max} - x_0$$

Where *n* is the width of the PML in terms of grid points in your computational domain. We now have everything we need to build up our PML:

1) We decide how much we want our potential or wave to be attenuated throughout the PML by setting *R* to be, e.g. 10^{-4},

2) We decide how wide we want our PML to be in terms of the number of grid points, *n*, e.g. 15,

3) We decide the order of the polynomial, *m*, that we're using for sigma, e.g. 4.

Please see Appendix D for some sample MATLAB code that defines a potential 2D PML, the one we use for all of our other code. I haven't found any good references on this, but the order of polynomial you use for sigma, *m*, will affect where reflections occur within your PML. Because the polynomial is effectively normalized based on the amount of total attenuation, a higher order polynomial will be steeper near the back of the PML, and much shallower at the front. This means that there will be less attenuation at the front section of the PML, and more reflection at the back; however, overall attenuation throughout should be the same. There aren't really any good metrics on this—I don't think—so you'll have to play around to see what works for you.

Keep in mind, that while this works, you do have to be careful of stability as you adjust the parameters—especially near the corners. Since the sigmas add an imaginary part to the x- and y-components of our 2D computational domain, the magnitude of each coordinate will no longer be identical throughout the domain (as in a finite difference scheme). That means that you'll have to find the largest change in magnitude to determine stability. That is, when you go to figure out the largest time step you need to maintain stability, you have to take into account:

$$\Delta x \rightarrow \left| \Delta x + j \max\left(\frac{\partial \sigma_x}{\partial x} \right) \right|$$

One other thing we haven't discussed here is that total attenuation by the PML is dependent on the angle of incident of the waves, such that maximum attenuation is at normal incidence. For my purposes, and general behaviour, this isn't such a huge deal as every time the wave doesn't get attenuated enough, and bounces off one of the sides, it'll eventually get to another boundary for further attenuation. But if you need any proper accuracy to a significant degree, you might have to take additional precautions.

We can also speed up the code in the appendices a bit (and also reducing memory requirements) by re-writing things to be less verbose. That is, instead of having individual variables for each step, we can use temporary or dummy variables (e.g. using a single `temp` variable instead of `dudx` and `dvxdx`). We can also swap out some of the MATLAB built-in functions like `gradient` or `divergence` for ones that are optimized for our specific case. MATLAB functions handle the general case incredibly well, but that creates some overhead that we can eliminate if we know exactly what we're up to.

The last thing to discuss is extending things into 3D. Following the prescription laid out in [4] as we've done above it's relatively straightforward, though with one complication: we now have to extend a third coordinate into the complex domain. What this means is that after multiplying everything, then expanding out all of the algebra, we'll now get an auxiliary differential equation that's second-order in time. This means we have to keep track of at least two additional variables represented as three-dimensional, full matrices. And that's where things get interesting: 3D matrices take up a lot of space. When we go ahead and actually go through with the work of bringing everything into the third dimension, we have to take extra care to take advantage of sparse matrices as much as possible.

References

[1] D. Bohm, "*Quantum Theory,"* Dover Publications, Inc.; New York, NY; 1951.

[2] J. D. Jackson, "*Classical Electrodynamics, Third Edition,"* John Wiley & Sons, Inc.; Hoboken, NJ; 1999.

[3] A. Taflove, S. C. Hagness, "*Computational Electrodynamics, The Finite-Difference Time-Domain Method, Third Edition,*" Artech House Inc.; Norwood, MA; 2005.

[4] S. G. Johnson, "*Notes on Perfectly Matched Layers (PMLs),"* March 10, 2010; Online at: `https://math.mit.edu/~stevenj/18.369/pml.pdf`

[5] S. Mazumder, *"Numerical Methods for Partial Differential Equations,"* Elsevier Inc.; Waltham, MA; 2016.

Appendix A: First Order, Explicitly Solved PML Implementation

```
clear;   close all;   showpotential = true;

%% Initialize computational domain:
n = 256;   c = 1;   tmax = 2^10;

xmin = -50; xmax = 50;   x = linspace(xmin, xmax, n);   dx = x(2) - x(1);
[x, y] = meshgrid(x, x);

phi = exp(-(x.^2+y.^2)/2)/(2*pi);

%% Initialize perfectly matched layer & simulation variables:
[sigmax, sigmay] = setupPML(x, dx);

dt = 0.25 * dx / c;
s_xplusy = 1/c^2*(sigmax + sigmay);
s_xtimesy = 1/c^2*(sigmax.*sigmay);

psi = zeros(size(x));
u_now = zeros(size(x));

vx = zeros(size(x));
vy = zeros(size(x));

s = c^2 * dt^2;

%% Run the simulation:
if showpotential
    fig = mesh(x, y, u_now);
    axis([xmin xmax xmin xmax -0.2 0.2]);
    xlabel('x');   ylabel('y');
    zlabel('Wave Amplituide');
    title('Generic Wave with PML at Boundary');
end

tavg = 0;
for t = 1:tmax
    tic;

    [dudx, dudy] = gradient(u_now, dx);
    vx = vx + dt * (dudx - vx.*sigmax);
    vy = vy + dt * (dudy - vy.*sigmay);

    [dvxdx, ~] = gradient(vx, dx);
    [~, dvydy] = gradient(vy, dx);
    psi = psi + ...
        dt * (sigmay.*dvxdx + sigmax.*dvydy - s_xtimesy.*u_now + cos(dt*t) * phi);

    u_now = u_now + dt * c^2 * (dvxdx + dvydy - s_xplusy.*u_now + psi);

    tavg = tavg + toc;

    if showpotential && ~mod(t, 4)
        fig.ZData = u_now;
        drawnow;
    end
end

disp(['Average single-pass calculation time: ', num2str(tavg / tmax * 1e3), ' ms']);
```

Appendix B: Second Order, Explicitly Solved PML Implementation

```
clear;   close all;   showpotential = true;

%% Initialize computational domain:
n = 256;   c = 1;   tmax = 2^12;

xmin = -50; xmax = 50;   x = linspace(xmin, xmax, n);   dx = x(2) - x(1);
[x, y] = meshgrid(x, x);

phi = exp(-(x.^2+y.^2)/2)/(2*pi);

%% Initialize perfectly matched layer & simulation variables:
[sigmax, sigmay] = setupPML(x, dx);

dt = 0.5 * dx / c;
s_xplusy = dt/2 * (sigmax + sigmay);
s_xtimesy = dt^2 * sigmax .* sigmay;

u_prev = zeros(size(x));
u_now = zeros(size(x));

vx = zeros(size(x));
vy = zeros(size(x));

s = c^2 * dt^2;

%% Run the simulation:
if showpotential
    fig = mesh(x, y, u_now);
    axis([xmin xmax xmin xmax -0.2 0.2]);
    xlabel('x');   ylabel('y');
    zlabel('Wave Amplituide');
    title('Generic Wave with PML at Boundary');
end

tavg = 0;
for t = 1:tmax
    tic;

    [dudx, dudy] = gradient(u_now, dx);
    vx = vx + dt*(dudx - sigmax.*vx);
    vy = vy + dt*(dudy - sigmay.*vy);

    u_next = (s *(4 * del2(u_now, dx) - divergence(x, y, sigmax.*vx, sigmay.*vy) ...
        + cos(dt*t) * phi) - (s_xtimesy - 2).*u_now - (1 - s_xplusy).*u_prev)./(1 + s_xplusy);

    u_next(1,:) = 0; u_next(:,1) = 0; u_next(end,:) = 0; u_next(:,end) = 0;
    u_prev = u_now;
    u_now = u_next;

    tavg = tavg + toc;

    if showpotential && ~mod(t, 4)
        fig.ZData = u_now;
        drawnow;
    end
end

disp(['Average single-pass calculation time: ', num2str(tavg / tmax * 1e3), ' ms']);
```

Appendix C: First Order, Semi-Implicitly Solved PML Implementation

```
clear;   close all;   showpotential = true;

%% Initialize computational domain:
n = 256;   c = 1;   tmax = 2^12;

xmin = -50; xmax = 50;   x = linspace(xmin, xmax, n);   dx = x(2) - x(1);
[x, y] = meshgrid(x, x);

phi = exp(-(x.^2+y.^2)/2)/(2*pi);

%% Initialize perfectly matched layer & simulation variables:
[sigmax, sigmay] = setupPML(x, dx);

dt = 0.5 * dx / c;
s_xtimesy = 1/c^2*(sigmax.*sigmay);

psi = zeros(size(x));

Avx = 1./(1 + dt/2 * sigmax);   Bvx = 1 - dt/2 * sigmax;
Avy = 1./(1 + dt/2 * sigmay);   Bvy = 1 - dt/2 * sigmay;
Au = 1./(1 + dt/2 * (sigmax + sigmay));   Bu = 1 - dt/2 * (sigmax + sigmay);

u_now = zeros(size(x));

vx = zeros(size(x));
vy = zeros(size(x));

s = c^2 * dt^2;

%% Run the simulation:
if showpotential
    fig = mesh(x, y, u_now);
    axis([xmin xmax xmin xmax -0.2 0.2]);
    xlabel('x');   ylabel('y');
    zlabel('Wave Amplituide');
    title('Generic Wave with PML at Boundary');
end

tavg = 0;
for t = 1:tmax
    tic;

    [dudx, dudy] = gradient(u_now, dx);
    vx = Avx.*(Bvx.*vx + dt*dudx);
    vy = Avy.*(Bvy.*vy + dt*dudy);

    [dvxdx, ~] = gradient(vx, dx);
    [~, dvydy] = gradient(vy, dx);
    psi = psi + dt * (sigmay.*dvxdx + sigmax.*dvydy - s_xtimesy.*u_now + cos(dt*t) * phi);

    u_now = Au.*(Bu.*u_now + dt*c^2 * (dvxdx + dvydy + psi));

    tavg = tavg + toc;

    if showpotential && ~mod(t, 4)
        fig.ZData = u_now;
        drawnow;
    end
end

disp(['Average single-pass calculation time: ', num2str(tavg / tmax * 1e3), ' ms.']);
```

Appendix D: Sample PML Implementation

```
function [sigmax, sigmay] = setupPML(x, dx)
    PMLwidth = 15;   m = 4;

    PMLwidth = PMLwidth * dx;
    sigmax_max = log(1e4)*(m+1) / PMLwidth^(m+1);
    PMLwidth = max(max(x)) - PMLwidth;

    sigmax = zeros(size(x));
    idx = abs(x) > PMLwidth;
    sigmax(idx) = sigmax_max * (abs(x(idx)) - PMLwidth).^m;
    sigmay = sigmax.';

    sigmax = sparse(sigmax);
    sigmay = sparse(sigmay);
end
```