Final Project: Maxwell's Equations

Andrew Emerick

December 21, 2013

1 Maxwell's Equations

In my code, I evolve Maxwell's Equations in a vacuum under a Coulumb gauge using the iterated Crank-Nicholson method. The code works in parallel, and I have tested it successfully on a 2^3 processor grid (more on this later). The code is split into two files, parallel_functions.cpp and maxwell.cpp. The former contains the functions used to set up the simulation volume and assign all of the bookkeeping neighbor matrices, both for the local processor gridpoints and for the overall processor grid (these functions are mostly the same as those that I used in the previous exercises, with possibly some minor tweaks). I will focus on describing functions in maxwell.cpp.

The initial conditions are set in the intial_conditions function. This converts the intial conditions given on A^ϕ and E^ϕ to their cartesian vector field counterparts. The laplacian function takes the laplacian of the passed vector (this is needed to evolve the equations). calculate_B calculates the magnetic field from the vector potential, this is discussed further in Sec. 1.3.

1.1 Iterated Crank-Nicholson Scheme

The equations are evolved in cn_evolve, which simultaneously iterates A and E through the iterated Crank-Nicholson scheme to evolve the vector fields in the simulation volume a specified time dt. The first iteration of this scheme is as follows:

$$ {}^1\tilde{A}_j^{n+1} = A_j^n - E_j^n \Delta t \tag{1} $$

$$ {}^1\tilde{E}_j^{n+1} = -\nabla^2(A^n)\Delta t + E_j^n \tag{2} $$

$$ {}^1\bar{A}_j^{n+1/2} = \frac{1}{2}({}^1\tilde{A}_j^{n+1} + A_j^n) \tag{3} $$

$$ {}^1\bar{E}_j^{n+1/2} = \frac{1}{2}({}^1\tilde{E}_j^{n+1} + A_j^n) \tag{4} $$

∇^2 is the discretized laplacian. This is repeated a second time, with $\bar{A}$ taking place of A in the equation for E, and the $\bar{E}$ taking place of E in the equation for A. Keepign with the same notation, the final step to calculate the n+1 values is:

$$ A_j^{n+1} = A_j^n - {}^2\bar{E}_j^{n+1/2}\Delta t \tag{5} $$

$$ E_j^{n+1} = -\nabla^2({}^2\bar{A}_j^{n+1/2})\Delta t + E_j^n \tag{6} $$

This is done once for each vector component (x,y,z), for a total of three times.

1.2 Periodic Boundary Condition

The above scheme was applied to the periodic boundary condition case. This was fairly simply to employ, as the way the neighbor matrices are designed, they automatically employ the boundary conditions (since my neighbor matrices are based upon the one you wrote for the irl matlab code). This is the case as well for the processor neighbor matrix. In the zipped package, I have included a movie showing the evolution of the electric field energy density, magnetic field energy density, and total energy density run in an xy slice near the center run on a single processor. For comparison, I have included the evolution of the total energy density from the 8 processor run (each processor outputs its data separately, so this movie has 1

plot for each of the 4 processors in the xy slice). dt = 0.01 for these results, and the color map is logscale. In each case, the field reaches the boundary near t=2, with ripple effects becoming very noticeable for t¿4.

1.3 Energy Conservation

Energy is not perfectly conserved. I think it could be argued that this is due to some error (round off) in the algorithm, but there could also be some error in my calculation of the magnetic field. I suspect that the error lies primarily in the discretized curl of the vector field that I use to calculate the magnetic field. I made an attempt to mitigate this error by using an alternative means of calculating the magnetic field energy. I took advantage of the fact that $\nabla \cdot \vec{A}$ is zero in this gauge. Using vector identities, I reduced $\vec{B} \cdot \vec{B}$ to $(\nabla \times \vec{A}) \cdot (\nabla \times \vec{A})$, which ultimately becomes $\nabla^2 \|\vec{A}\|^2$. This should produce the magnetic energy density at each grid point, but this was inconsistent with the curl method (larger by a factor ¡2), and also had similar energy dissipation issues. NOTE: Any results I give (unless specified otherwise) are using the curl method of calculating the magnetic field.

The nature of the lack of energy conservation for t<0.5 worries me somewhat, as there is a slight oscillation. I would have guessed that, if the error was purely numerical roundoff, the energy loss would be roughly linear with time (as it is for larger t). This is shown in Fig. 1 for the periodic boundary condition. Plotted is the electric field energy (red), magnetic field energy calculated with the curl (green), $\sum \vec{A} \cdot \vec{A}$ (dark blue), total energy using the curl magnetic field (purple), magnetic field energy calculated with the laplacian (light blue), and total energy using the laplacian magnetic field (yellow). The loss in the curl method total energy is roughly 8% by t=1.

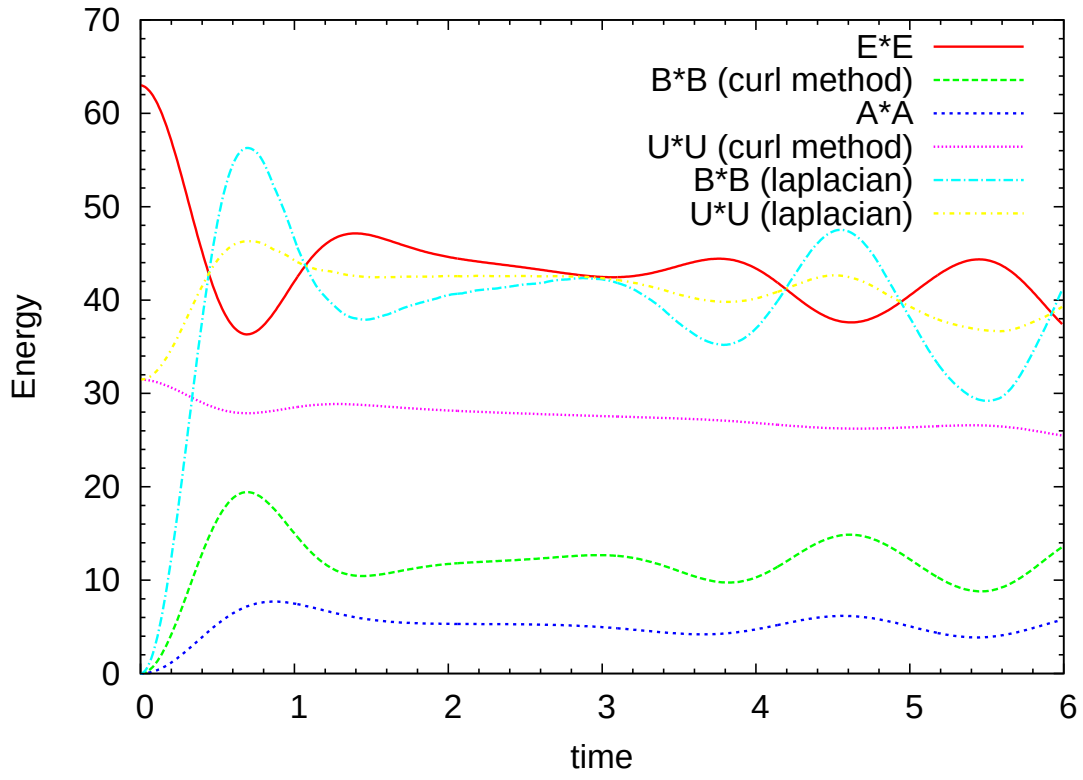

Figure 1: Energy evolution example as a function of time for the periodic boundary conditions, and dt = 0.01.

1.4 Limiting Δt

Fig. 2 demonstrates that the algorithm can very quickly become unstable when using too large dt spacing. The solution quickly blows up for dt = 0.5 (dt = 0.25 still gave a reasonable result). Testing a few dt, I found that (in general) larger time spacings meant worse energy conservation (though not radically), as long as the system stayed away from the boundaries. (Again I am using periodic boundaries here). Once the system begins reflecting across the boundary, the solution generally blows up, faster for larger dt. For the dt = 0.5 example shown, the total energy increases by an order of magnitude or two every time step after the first few.

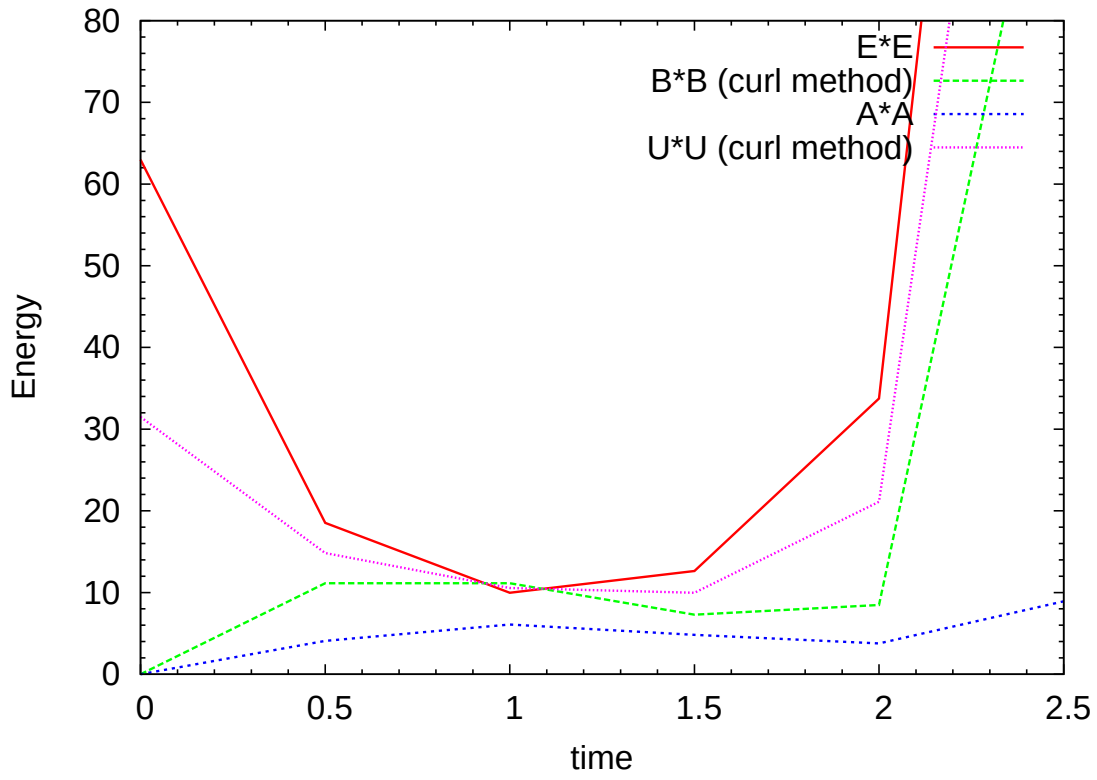

Figure 2: Energy evolution instability for large time spacing. Shown is the result for dt = 0.5. The solution is reasonable (though with large energy dissipation) while the system stays away from the boundary. Once the boundary is hit, the solution blows up drastically.

1.5 Sommerfeld Condition

The Sommerfeld radiation condition is supposed to model a real system, where the simulation region is just a slected 'view' into a larger volume, and information that would be sent past the boundary would be diffused out, as opposed to reflected in the periodic case. Ensuring this, while preventing reflections, is challenging. I have given some thought to implementing a naive case where the phase velocity of the wave is exactly Δx / Δt. In this case, my thought was to calculate the n+1 values at the boundary as:

$$U^{n+1}_{bndry} = U^{n}_{bndry-1} \tag{7}$$

where U is a vector field (A or E in this case). This would be applied separately for each component of U (x,y,z). For example, for the grid points sitting on the right hand side x boundary (maximum x), only the x-component of U will be modified according to this scheme. The y and z components will be solved for normally. This may have been a naive implementation, and in actually all three components should be modified in a similar fashion. An initial test of this did not prevent reflection at the boundary.

The failure of this implementation could be that it works only for the wave velocity Δx / Δt. I found [1], which gives a more robust implementation that makes an estimation of the wave velocity at every boundary point. This algorithm requires knowledge of grid points two spaces away from the boundary however. In addition, to calculate the n+1 values, it requires knowledge of the n and n-1 values. Implementing this scheme will take some more coding, though should not be exceedingly challenging. The main adjustments would be doubling the size of the neighbor grid point matrix (to contain knowledge of values two grid spacings away), and also some editing to preserve the two previous time steps. A smarter way of doing this would be to only preserve values near the boundary (leaving most of the simulation volume alone), but this requires some more thought in order to be done properly.

References

[1] I. Orlanski, *A Simple Boundary Condition for Unbounded Hyperbolic Flows* Journal of Computational Physics, Vol. 21, No. 3, Jul. 1976