An Iterative Solver For The Diffusion Equation

Alan Davidson

April 28, 2006

Abstract

I construct a solver for the time-dependent diffusion equation in one, two, or three dimensions using a backwards Euler finite difference approximation and either the Jacobi or Symmetric Successive Over-Relaxation iterative solving techniques. This solver supports Dirichlet conditions in arbitrary geometries within the region over which we compute.

1 Introduction

This past year, I was on a clinic for Los Alamos National Lab. They had created a three-dimensional computer simulation of avascular (lacking blood vessels) tumor growth, and wanted us to implement a vasculature and then simulate chemotherapy doses flowing in through the blood. In the model, several different chemicals diffused throughout the tumor and surrounding areas. Moreover, the blood vessels needed to be treated as Dirichlet boundary conditions (with constant chemical concentrations). These vessels, however, could run through the middle of the tumor in any geometry. This made solving the diffusion equation quite difficult. We eventually created an iterative solver using the Gauss-Seidel method using a backwards Euler finite difference approximation, written in C++.

In this report, I describe a similar solver I created in Matlab using a backwards Euler FDA. However, I experimented with both the Jacobi and Symmetric Successive Over-Relaxation iterative methods to solve the diffusion equation. My system also allows arbitrary Dirichlet boundary conditions, but can be implemented in one, two, or three dimensions.

2 Background

The diffusion equation, which is also known as the heat equation, can be written as

$$u_t = a + D \cdot u_x x$$

where u is the concentration of chemical, a is a source/sink term describing production or consumption of the chemical, D is a physical diffusion constant, t is time, and x is space. In order to solve this equation, I used a FDA to create a system of linear equations of the form $A\vec{u} = \vec{b}$ to solve instead.

Unfortunately, there is not a useful structure to the matrix A which can be exploited to solve this problem easily (this is especially true in the three-dimensional case). Every way to get an exact answer to this system requires the same amount of work as finding A^{-1}. However, it is possible to more easily find an approximation to the solution by using an iterative method.

In these methods, it is typical to divide the matrix A into three separate parts. Let $A = D + U + L$ where D is a diagonal matrix, U is strictly upper triangular, and L is strictly lower triangular. In iterative methods, the system of equations is solved by starting with an initial guess for $\vec{u}$, called $\vec{u}_0$. From this, we can generate a new vector $\vec{u}_1$ which is intended to be closer to the correct solution. We can then plug $\vec{u}_1$ back into the system to create $\vec{u}_2$, etc. Eventually, we hope that the system will converge on a certain value, $\vec{u}_\infty$, and that this value will be the solution we seek. In both of the methods I explored, we define $\vec{u}_{i+1} = P\vec{u}_i + Q$, where P and Q are constant matrices which we need only compute once.

2.1 The Jacobi Method

The Jacobi method uses the following iterative step:

$$\vec{u}_{i+1} = D^{-1} \cdot (-L - U)\vec{u}_i + D^{-1} \cdot \vec{b}$$

The only matrix inversion required is that of D, and because D is diagonal, finding its inverse is quite simple. Moreover, if this method converges to any solution, the solution will satisfy the property that

$$(D + L + U) \cdot \vec{u}_\infty \;=\; \vec{b}$$

Thus, if this method converges, it converges on the correct solution.

The Jacobi method only converges when the diagonal terms of $D^{-1} \cdot (-L - U)$ are larger than the sum of the other terms in the apposite row. However, this condition is satisfied when the norm of the matrix is less than 1. I have used this latter criterion to determine the stability of the system. Although it is possible that I will interpret some stable systems as unstable, I will correctly classify the vast majority of systems.

2.2 The SSOR Method

The Successive Over-Relaxation (or SOR) method relies on the fact that our problem can be equivalently written as

$$\omega A \vec{u} \;=\; \omega \vec{b}$$

where ω is a non-zero scalar. This allows us to rewrite A in the following way:

$$\omega A \;=\; (D + \omega L) + (\omega U - (1 - \omega) D)$$

We can iteratively solve for $\vec{u}$ by defining

$$\vec{u}_{(i+1)} \;=\; (D + \omega L)^{-1} \cdot \Big((-\omega U + (1 - \omega) D) \vec{u}_{(i)} + \omega \vec{b} \Big)$$

This is known as the forwards SOR method. The only matrix we need to invert is $D + \omega L$, which is lower triangular and therefore simpler to invert than A.

A similar method, known as the backwards SOR, defines A as

$$\omega A \;=\; (D + \omega U) + (\omega L - (1 - \omega) D)$$

and the iterative step as

$$\vec{u}_{(i+1)} \;=\; (D + \omega U)^{-1} \cdot \Big((-\omega L + (1 - \omega) D) \vec{u}_{(i)} + \omega \vec{b} \Big)$$

The Symmetric SOR (SSOR) method is a hybrid of these two: each iteration of the SSOR method consists of one forward SOR step and one backward SOR step. Consequently,

$$\begin{aligned}
\vec{u}_{(i+1/2)} &= (D + \omega L)^{-1} \cdot \Big((-\omega U + (1-\omega) D)\vec{u}_{(i)} + \omega \vec{b} \Big) \\
\vec{u}_{(i+1)} &= (D + \omega U)^{-1} \cdot \Big((-\omega L + (1-\omega) D)\vec{u}_{(i+1/2)} + \omega \vec{b} \Big)
\end{aligned}$$

The SSOR method has the same convergence properties as the Jacobi method, namely that it will converge if the norm of the matrix by which we multiply the previous iteration's solution is less than 1, and if it converges, it will converge to the correct solution.

3 The Method

I used the sparse matrix representation in Matlab to create my solver. Although I had separate cases for creating A, based on the number of dimensions of the input, the actual solver was the same regardless of the dimension of the original area. It is worth noting, however, that the dimensions of A, $\vec{u}$, and $\vec{b}$ are not the same as that of the region for which we are solving the diffusion equation. A is always two-dimensional, while $\vec{u}$ and $\vec{b}$ are always one-dimensional. If we are solving the equation over a two-dimensional region, the concentration of the chemical at location (i, j) can be found at $\vec{u}(i + sizeX \cdot (j-1))$, where $sizeX$ is the length of the overall region, in sites. Note that we subtract 1 from j for correct indexing because Matlab begins indexing arrays at 1 instead of 0. Analogously, in the three-dimensional case we have location (i, j, k) mapped onto element $\vec{u}(i + sizeX \cdot (j-1) + sizeX \cdot sizeY \cdot (k-1))$.

4 Results

Throughout all of these runs, I have used a timestep of 0.01 for each iteration. The time denoted in the figures is the total amount of time that has passed, so the higher this number is, the more steps have been taken.

The solver seems to be working quite well for the one-dimensional case. For instance, in Figure 4, chemical is diffusing in from the 3 boundary conditions.

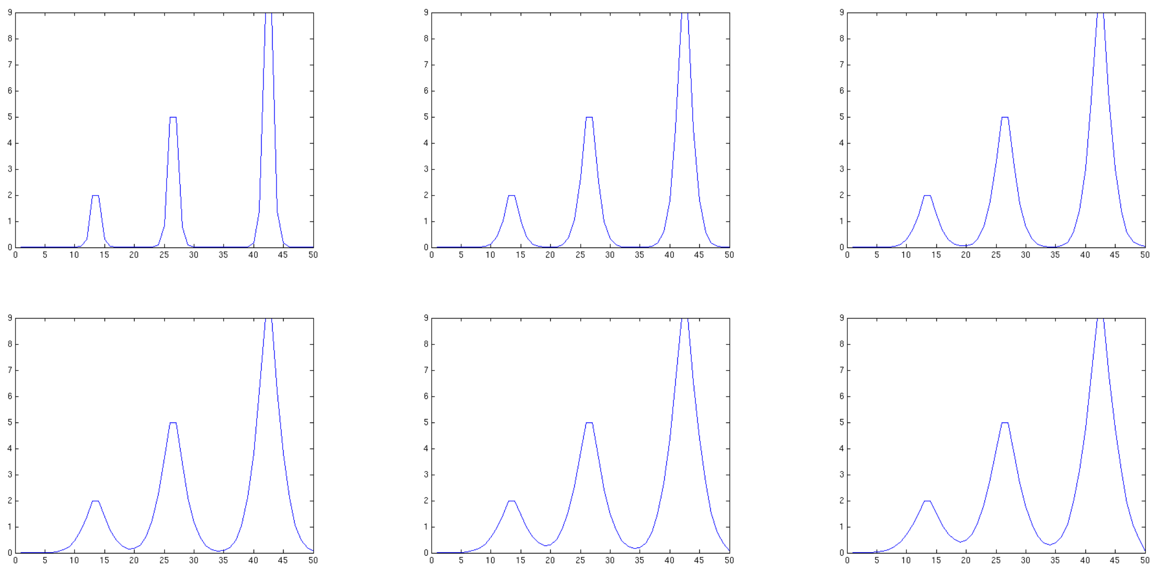

Figure 1: Several snapshots using the SSOR method ($\omega = 0.5$) in the one-dimensional case. The three spikes are Dirichlet BCs, and the chemical is diffusing out of them into the rest of the space.

There is a little variation between the two iterative methods, and between different values of ω in the SSOR method, as can be seen in Figure 4. However, when ω becomes very small, the solution becomes significantly different, as shown in Figure 3. I hypothesize that this is because as ω approaches 0, the solution is no longer stable and is likely to diverge. For instance, the first timestep of this run took over 3000 iterations to converge, while identical runs with larger values of ω took fewer than ten iterations.

Although I believe that the two-dimensional and three-dimensional versions of the solver are working properly, I did not find a useful way to visualize the solutions, so results from them have not been included.

5 Conclusion

Iterative solvers can be used to quickly get approximate solutions to systems of linear equations which may not be easily solved with other methods. My solver can generate realistic

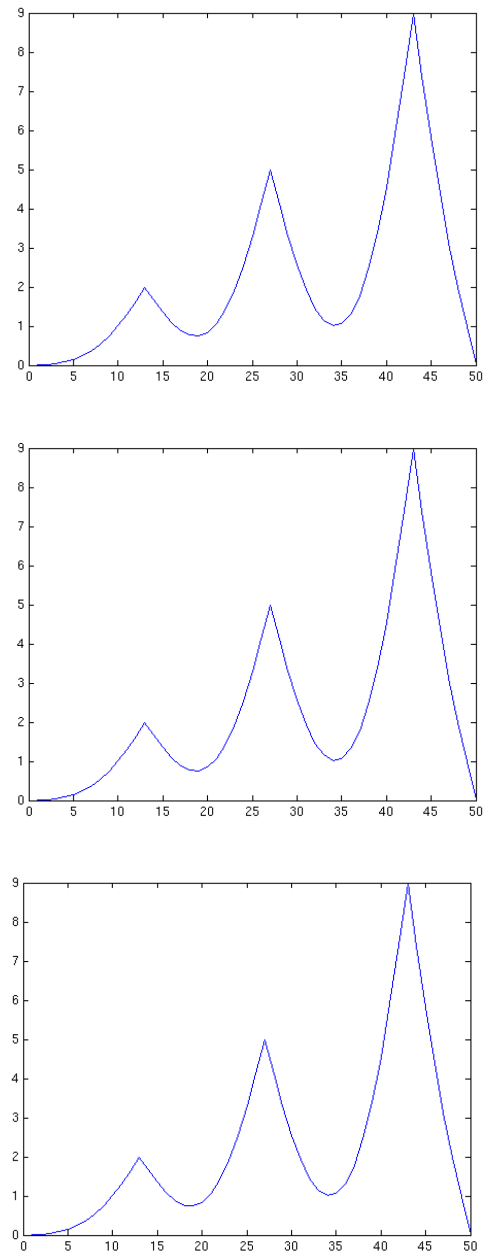

Figure 2: A later timestep (t=1.00) using the Jacobi method (top), and SSOR method with ω equal to 1 (middle) and 0.5 (bottom). There appears to be little difference between the solutions found by these different solving methods.

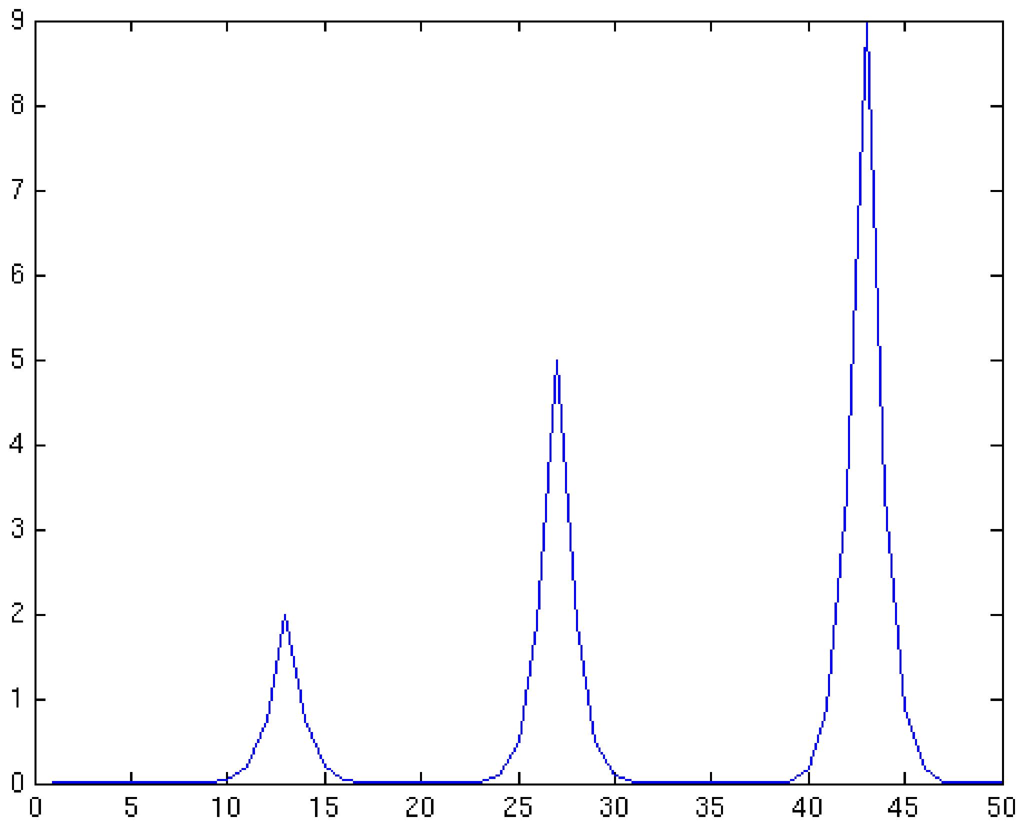

Figure 3: A later step (t=1.00) using the SSOR method ($\omega = 0.001$) in the one-dimensional case. As ω approaches 0, the solver becomes unstable and gives more inaccurate results. If ω is negative, the system never converges to a solution.

results for the time-dependent diffusion equation, even with arbitrary Dirichlet boundaries within the region over which we are solving. For values of ω close to 0, the solution found by the SSOR method appears to be innacurate, but other solutions do not appear to differ significantly from each other.

A Source Code - `iterative.m`

The following is the Matlab code which I used. It handles one, two, or three dimensional cases using either the Jacobi or SSOR iterative methods.

```
%iterative(dx,dt,gridSize,initcond,sources,
%          dirichlet,tolsq,Dconst,solver,omega)


% Alan Davidson
% SciComp final project


% This file will use the Jacobi iterative method to solve the
% time-dependent diffusion equation,
%    u_t=D*(u_xx + u_yy + u_zz) + a(x,y,z)
% where a is a source/sink term
%       u is the temperature of that spot
%       t is time
%       x, y, and z are spacial dimensions (possibly not all are used)
%       D is the diffusion constant
%
% We use a first-degree Backward Euler FDA, so we are trying to solve
%
% u(x,y,z,t+dt)+
%      dt*D/dx^2*((2*dim)*u(x,y,z,t+dt)-sum_neighbors(u(x',y',z',t+dt)))
%      =u(x,y,z,t)+dt*a(x,y,z)

% When we have a Dirichlet boundary condition, note that we instead want
%
% u(x,y,z,t+dt) = u(x,y,z,t)
% We use the Jacobi iterative method, which is the following: let us
% suppose we are trying to solve Ax=b, and that x(0) denotes the initial
% condition of the system. Now, let
%
% x(i+1) = D^-1 * (-L-U)*x(i) + D^-1 * b  or in other words
% x(i+1) = P*x(i) + Q
%
% where D, L, and U are the diagonal, lower triangular, and upper
% triangular parts of A, respectively, and P=D^-1*(-L-U) and Q=D^-1*b.
% We repeat this process until ||x(i+1)-x(i)|| is less than our tolerance,
% in which case we stop, and call x(i+1) a good approximation of the
% answer.

% It should be noted that I started this as part of my clinic. Cris Cecka
% did much of the work in getting the clinic code working, which was
% written in C++. I have adapted it myself for Matlab. I have also adapted
% it to work in fewer than 3 dimensions.

% The function prints out the number of iterations required to converge.
% It returns the array of chemical concentrations.

function y=iterative(dx,dt,gridSize,initcond,sources,dirichlet,...
                     tolsq,Dconst,solver,omega)
% dx is the amount by which the grid spaces are separated
% dt is the amount by which we step through time
% gridSize is a vector with 1, 2, or 3 elements, denoting the
%      size/dimension of the grid
% initcond is the initial condition of the system. The number of elements
%      in initcond must be equal to the product of the elements of size.
% dirichlet is a matrix of the Dirichlet conditions, with a
%      negative number for a place without a Dirichlet BC (this is ok,
%      since the concentration of a diffused chemical should never be
%      negative, much less held at that value, so we should never actualy
%      require a Dirichlet BC to actually be negative).
% tolsq is the square of the tolerance to be used at any given timestep
% Dconst is the diffusion constant
% solver is a string - either 'jacobi' or 'ssor' for the different methods
% omega is the scalar in the SSOR (if solver='jacobi' it is ignored)

% temp should be 1
[temp,dim]=size(gridSize);
totalSize=prod(gridSize);

if(temp ~= 1 || dim<1 || dim > 3)
    disp('Too many (or too few?) dimensions.')
    y=0;
    return
end

[numSites,temp]=size(initcond);
if(temp ~= 1 || numSites ~= totalSize)
    disp('Size of initial conditions is wrong.')
    y=0;
    return
end
```

```
[numSites,temp]=size(sources);
if(temp ~= 1 || numSites ~= totalSize)
    disp('Size of source/sink terms is wrong.')
    y=0;
    return
end

[numSites,temp]=size(dirichlet);
if(temp ~= 1 || numSites ~= totalSize)
    disp('Size of boundary conditions is wrong.')
    y=0;
    return
end

% Every dimension of the grid should have at least 4 elements in it.
if(any(gridSize<4))
    disp('Region is too small; quitting');
    % OK, this is only necessary if there are places along the edge
    % of the region that aren't boundary conditions, but I'm going
    % with it anyway. If the region is too small, the FDA on the boundary
    % won't work correctly.
    y=0;
    return
end

% Where the BC exists, we want elements of x to be identically
% equal to the apposite elements of b.
A=speye(totalSize);
```

```
switch dim
    case 1
        % Where the BC doesn't exist, we want to add 2D*dt/dx^2 to the
        % diagonal term
        [i,j]=find(dirichlet<0);
        A=A+sparse(i,i,2*Dconst*dt/(dx*dx),totalSize,totalSize);
        % We also want the off-diagonal terms to be -D*dt/dx^2 in these
        % cases, but not the ones on the ends!

        % We start out by making an array with the ends marked
        edges=zeros(totalSize,1);
        edges(1)=1;
        edges(totalSize)=1;
        % Now, make temp equal dirichlet, except with 1's forced on the
        % ends. This makes temp be -1 at all interior non-BC points
        temp=dirichlet-(edges.*dirichlet)+edges;
        [i,j]=find(temp<0);
        % Now, we can finally make the off-diagonal terms.
        A=A+sparse(i,i+1,(-Dconst*dt/(dx*dx)),totalSize,totalSize);
        A=A+sparse(i,i-1,(-Dconst*dt/(dx*dx)),totalSize,totalSize);

        % Lastly, we need to worry about non-BC sites on the edges.
        % Here, we want y''(0)=(2y(0)-5y(1)+4y(2)-y(3))/dx^2.
        % We have already added on the 2y(0)/dx^2 term.
        if(dirichlet(1)<0)
            A=A+sparse([1;1;1],[2;3;4],[-5;4;-1].*(Dconst*dt/(dx*dx)),...
                totalSize,totalSize);
        end
        if(dirichlet(totalSize)<0)
            A=A+sparse([totalSize;totalSize;totalSize],...
                       [totalSize-1;totalSize-2;totalSize-3],...
                       [-5;4;-1].*(Dconst*dt/(dx*dx)),totalSize,totalSize);
        end

    case 2
        % The space (x,y) will be indexed into the array at location
        % (x+gridSize(1)*(y-1)).

        % We will handle the interior and the edges of the area separately:
        indices=1:1:totalSize;
        notTheTop=indices>gridSize(1);
        notTheBottom=indices<=(totalSize-gridSize(1));
        notTheLeft=mod(indices,gridSize(1))~=1;
        notTheRight=mod(indices,gridSize(1))~=0;
        interior=notTheTop.*notTheBottom.*notTheLeft.*notTheRight;

        % In places without BCs, we want a total of 1+4Dconst*dt/dx^2 on
        % the diagonal terms.
        [i,j]=find(dirichlet<0);
        A=A+sparse(i,i,4*Dconst*dt/(dx*dx),totalSize,totalSize);
        % In the interior, we subtract Dconst*dt/dx^2 from the off-diagonal
        % terms if we're not on the BC.
        [i,j]=find(interior.*(dirichlet<0));
        A=A+sparse(i,i+1,-Dconst*dt/(dx*dx),totalSize,totalSize);
        A=A+sparse(i,i-1,-Dconst*dt/(dx*dx),totalSize,totalSize);
```

```
A=A+sparse(i,i+gridSize(1),-Dconst*dt/(dx*dx),totalSize,totalSize);
A=A+sparse(i,i-gridSize(1),-Dconst*dt/(dx*dx),totalSize,totalSize);
% This should handle the top edge for non-BCs:
[i,j]=find((indices<=gridSize(1)).*(dirichlet<0));
A=A+sparse(i,i+gridSize(1),-5*Dconst*dt/(dx*dx),...
           totalSize,totalSize);
A=A+sparse(i,i+2*gridSize(1),4*Dconst*dt/(dx*dx),...
           totalSize,totalSize);
A=A+sparse(i,i+3*gridSize(1),-Dconst*dt/(dx*dx),...
           totalSize,totalSize);
% This should handle the bottom edge for non-BCs:
[i,j]=find((indices>totalSize-gridSize(1)).*(dirichlet<0));
A=A+sparse(i,i-gridSize(1),-5*Dconst*dt/(dx*dx),...
           totalSize,totalSize);
A=A+sparse(i,i-2*gridSize(1),4*Dconst*dt/(dx*dx),...
           totalSize,totalSize);
A=A+sparse(i,i-3*gridSize(1),-Dconst*dt/(dx*dx),...
           totalSize,totalSize);
% This should handle the left edge for non-BCs:
[i,j]=find((mod(indices,gridSize(1))==1).*(dirichlet<0));
A=A+sparse(i,i+1,-5*Dconst*dt/(dx*dx),totalSize,totalSize);
A=A+sparse(i,i+2,4*Dconst*dt/(dx*dx),totalSize,totalSize);
A=A+sparse(i,i+3,-Dconst*dt/(dx*dx),totalSize,totalSize);
% This should handle the right edge for non-BCs:
[i,j]=find((mod(indices,gridSize(1))==0).*(dirichlet<0));
A=A+sparse(i,i-1,-5*Dconst*dt/(dx*dx),totalSize,totalSize);
A=A+sparse(i,i-2,4*Dconst*dt/(dx*dx),totalSize,totalSize);
A=A+sparse(i,i-3,-Dconst*dt/(dx*dx),totalSize,totalSize);
```

```
case 3
    % The space (x,y,z) will be indexed into the array at location
    % (x+gridSize(1)*(y-1)+gridSize(1)*gridSize(2)*(z-1)).

    % We will handle the interior and the edges of the area separately:
    indices=1:1:totalSize;
    notTheTop=indices>gridSize(1)*gridSize(2);
    notTheBottom=indices<=(totalSize-gridSize(1));
    notTheLeft=mod(indices,gridSize(1))~=1;
    notTheRight=mod(indices,gridSize(1))~=0;
    notTheFront=mod(indices,gridSize(1)*gridSize(2))<=gridSize(1);
    notTheFront=mod(indices,gridSize(1)*gridSize(2))>=...
                gridSize(1)*gridSize(2)-gridSize(1);
    interior=notTheTop.*notTheBottom.*notTheLeft.*notTheRight.*...
             notTheFront.*notTheBack;
    % In places without BCs, we want a total of 1+6Dconst*dt/dx^2 on
    % the diagonal terms.
    [i,j]=find(dirichlet<0);
    A=A+sparse(i,i,6*Dconst*dt/(dx*dx),totalSize,totalSize);
    % In the interior, we subtract Dconst*dt/dx^2 from the off-diagonal
    % terms if we're not on the BC.
    [i,j]=find(interior.*(dirichlet<0));
    A=A+sparse(i,i+1,-Dconst*dt/(dx*dx),totalSize,totalSize);
    A=A+sparse(i,i-1,-Dconst*dt/(dx*dx),totalSize,totalSize);
    A=A+sparse(i,i+gridSize(1),-Dconst*dt/(dx*dx),totalSize,totalSize);
    A=A+sparse(i,i-gridSize(1),-Dconst*dt/(dx*dx),totalSize,totalSize);
    A=A+sparse(i,i+gridSize(1)*gridSize(2),...
               -Dconst*dt/(dx*dx),totalSize,totalSize);
    A=A+sparse(i,i-gridSize(1)*gridSize(2),...
               -Dconst*dt/(dx*dx),totalSize,totalSize);
    % This should handle the top face for non-BCs:
    [i,j]=find((indices<=gridSize(1)*gridSize(2)).*(dirichlet<0));
    A=A+sparse(i,i+gridSize(1)*gridSize(2),-5*Dconst*dt/(dx*dx),...
               totalSize,totalSize);
    A=A+sparse(i,i+2*gridSize(1)*gridSize(2),4*Dconst*dt/(dx*dx),...
               totalSize,totalSize);
    A=A+sparse(i,i+3*gridSize(1)*gridSize(2),-Dconst*dt/(dx*dx),...
               totalSize,totalSize);
    % This should handle the bottom face for non-BCs:
    [i,j]=find((indices>totalSize-gridSize(1)*gridSize(2))...
               .*(dirichlet<0));
    A=A+sparse(i,i-gridSize(1)*gridSize(2),-5*Dconst*dt/(dx*dx),...
               totalSize,totalSize);
    A=A+sparse(i,i-2*gridSize(1)*gridSize(2),4*Dconst*dt/(dx*dx),...
               totalSize,totalSize);
    A=A+sparse(i,i-3*gridSize(1)*gridSize(2),-Dconst*dt/(dx*dx),...
               totalSize,totalSize);
    % This should handle the left face for non-BCs:
    [i,j]=find((mod(indices,gridSize(1))==1).*(dirichlet<0));
    A=A+sparse(i,i+1,-5*Dconst*dt/(dx*dx),totalSize,totalSize);
    A=A+sparse(i,i+2,4*Dconst*dt/(dx*dx),totalSize,totalSize);
    A=A+sparse(i,i+3,-Dconst*dt/(dx*dx),totalSize,totalSize);
    % This should handle the right face for non-BCs:
    [i,j]=find((mod(indices,gridSize(1))==0).*(dirichlet<0));
    A=A+sparse(i,i-1,-5*Dconst*dt/(dx*dx),totalSize,totalSize);
```

```
        A=A+sparse(i,i-2,4*Dconst*dt/(dx*dx),totalSize,totalSize);
        A=A+sparse(i,i-3,-Dconst*dt/(dx*dx),totalSize,totalSize);
        % This should handle the front face for non-BCs:
        [i,j]=find((mod(indices,gridSize(1)*gridSize(2))<=gridSize(1))...
                   .*(dirichlet<0));
        A=A+sparse(i,i+gridSize(1),-5*Dconst*dt/(dx*dx),...
                   totalSize,totalSize);
        A=A+sparse(i,i+2*gridSize(1),4*Dconst*dt/(dx*dx),...
                   totalSize,totalSize);
        A=A+sparse(i,i+3*gridSize(1),-Dconst*dt/(dx*dx),...
                   totalSize,totalSize);
       % This should handle the back face for non-BCs:
        [i,j]=find((mod(indices,gridSize(1)*gridSize(2))<=...
                   gridSize(1)*gridSize(2)-gridSize(1))...
                   .*(dirichlet<0));
        A=A+sparse(i,i-gridSize(1),-5*Dconst*dt/(dx*dx),...
                   totalSize,totalSize);
        A=A+sparse(i,i-2*gridSize(1),4*Dconst*dt/(dx*dx),...
                   totalSize,totalSize);
        A=A+sparse(i,i-3*gridSize(1),-Dconst*dt/(dx*dx),...
                   totalSize,totalSize);


    otherwise
        disp('Not yet implemented');
        y=0;
        return
end
```

```
% used for debugging purposes
%full(A)

% b is the boundary conditions if they exist, and the other
% conditions, otherwise.
b=(dirichlet>=0).*dirichlet+(dirichlet<0).*(initcond+dt*sources);

D=speye(totalSize).*A;
L=tril(A,-1);
U=triu(A,1);

switch lower(solver)
    case 'jacobi'
        P=D^-1*(-L-U);
        Q=D^-1*sparse(b);

        if(norm(full(P))>=1)
            disp('System is not stable.')
            norm(full(P))
            y=0;
            return
        end
    case 'ssor'
        E=-L;
        F=-U;
        P=(D-omega*F)^-1*(omega*E+(1-omega)*D)*...
          (D-omega*E)^-1*(omega*F+(1-omega)*D);
        Q=omega*(2-omega)*(D-omega*F)^-1*D*(D-omega*E)^-1*b;
    otherwise
        disp('Invalid solving method');
        y=0;
        return
end

% Used for debugging purposes
%fullD=full(D)
%fullL=full(L)
%fullU=full(U)
%fullP=full(P)
%fullQ=full(Q)

if(norm(full(P))>=1)
    disp('System is not stable.')
    % It's possible in a few cases that the system is actually stable, but
    % I don't want to take chances on this sort of thing.
    norm(full(P))
    y=0;
    return
end

% Now, we actually want to compute the new concentrations.
xold=initcond;
% square of the magnitude of xold-xnew is ressq
ressq = tolsq+1;
iterations=0;
```

```
while(ressq>tolsq)
    xnew=P*xold+Q;
    ressq=dot(xnew-xold,xnew-xold);
    xold=xnew;
    iterations=iterations+1;

    % For debugging purposes
    %if(mod(iterations,50)==0)
    %    iterations
    %    ressq
    %end
end
% print out the number of iterations we needed.
iterations=iterations
y=xnew;
return
```

References

[1] Saad, Yousef, *Iterative Methods for Sparse Linear Systems*, (SIAM, 2000).

[2] `http://mathworld.wolfram.com`