Lab 8

Implicit Methods: the Crank-Nicolson Algorithm

You may have noticed that all of the algorithms we have discussed so far are of the same type: at each spatial grid point j you use present, and perhaps past, values of $y(x,t)$ at that grid point and at neighboring grid points to find the future $y(x,t)$ at j. Methods like this, that depend in a simple way on present and past values to predict future values, are said to be *explicit* and are easy to code. They are also often numerically unstable, and as we saw in the last lab, even when they aren't they can have severe constraints on the size of the time step. *Implicit* methods are generally harder to implement than explicit methods, but they have much better stability properties. The reason they are harder is that they assume that you already know the future.

Implicit methods

To give you a better feel for what "implicit" means, let's study the simple first-order differential equation

$$\dot{y} = -\gamma y \tag{8.1}$$

8.1 (a) Solve this equation using Euler's method:

$$\frac{y_{n+1} - y_n}{\tau} = -\gamma y_n \,. \tag{8.2}$$

Show by writing a simple Matlab script and doing numerical experimentation that Euler's method is unstable for large τ. Show by experimenting and by looking at the algorithm that it is unstable if $\tau > 2/\gamma$. Use $y(0) = 1$ as your initial condition. This is an example of an explicit method.

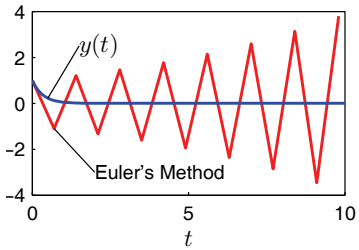

Figure 8.1 Euler's method is unstable for $\tau > 2/\gamma$. ($\tau = 2.1/\gamma$ in this case.)

(b) Notice that the left side of Eq. (8.2) is centered on time $t_{n+\frac{1}{2}}$ but the right side is centered on t_n. Let's center the the right-hand side at time $t_{n+\frac{1}{2}}$ by using an average of the advanced and current values of y,

$$y_n \Rightarrow \frac{y_n + y_{n+1}}{2} \,.$$

Show by numerical experimentation in a modified script that when τ becomes large this method doesn't blow up. It isn't correct because y_n bounces between positive and negative values, but at least it doesn't blow up. The presence of τ in the denominator is the tip-off that this is an implicit method, and the improved stability is the point of using something implicit.

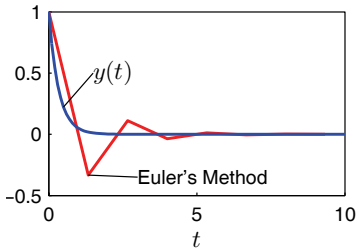

Figure 8.2 The implicit method in 8.1(b) with $\tau = 4/\gamma$.

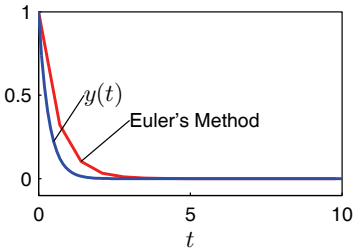

Figure 8.3 The fully implicit method in 8.1(c) with $\tau = 2.1/\gamma$.

(c) Now Modify Euler's method by making it *fully implicit* by using y_{n+1} in place of y_n on the right side of Eq. (8.2) (this makes both sides of the equation reach into the future). This method is no more accurate than Euler's method for small time steps, but it is much more stable. Show by numerical experimentation in a modified script that this fully implicit method damps away very quickly when τ is large. Extra damping is usually a feature of fully implicit algorithms.

The diffusion equation with Crank-Nicolson

Now let's look at the diffusion equation again, and see how implicit methods can help us. Just to make things more interesting we'll let the diffusion coefficient be a function of x:

$$\frac{\partial T}{\partial t} = \frac{\partial}{\partial x}\left(D(x)\frac{\partial T}{\partial x}\right) \tag{8.3}$$

We begin by finite differencing the right side, taking care to handle the spatial dependence of D. Rather than expanding the nested derivatives in Eq. (8.3) let's difference it in place on the grid. In the equation below $D_{j\pm\frac{1}{2}} = D(x_j \pm h/2)$.

$$\frac{\partial T_j}{\partial t} = \frac{D_{j+\frac{1}{2}}\left(T_{j+1} - T_j\right) - D_{j-\frac{1}{2}}\left(T_j - T_{j-1}\right)}{h^2} \tag{8.4}$$

8.2 Show that the right side of Eq. (8.4) is correct by finding a centered difference expressions for $D(x)\frac{\partial T}{\partial x}$ at $x_{j-\frac{1}{2}}$ and $x_{j+\frac{1}{2}}$. Then use these two expressions to find a centered difference formula for the entire expression at x_j. Draw a picture of a grid showing x_{j-1}, $x_{j-\frac{1}{2}}$, x_j, $x_{j+\frac{1}{2}}$, and x_{j+1} and show that this form is centered properly.

Now we take care of the time derivative by doing something similar to problem 8.1(b): we take a forward time derivative on the left, putting that side of the equation at time level $n+\frac{1}{2}$. To put the right side at the same time level (so that the algorithm will be second-order accurate), we replace each occurrence of T on the right-hand side by the average

$$T^{n+\frac{1}{2}} = \frac{T^{n+1} + T^n}{2} \tag{8.5}$$

like this:

$$\frac{T_j^{n+1} - T_j^n}{\tau} = \frac{D_{j+\frac{1}{2}}\left(T_{j+1}^{n+1} - T_j^{n+1} + T_{j+1}^n - T_j^n\right) - D_{j-\frac{1}{2}}\left(T_j^{n+1} - T_{j-1}^{n+1} + T_j^n - T_{j-1}^n\right)}{2h^2} \tag{8.6}$$

If you look carefully at this equation you will see that there is a problem: how are we supposed to solve for T_j^{n+1}? The future values T^{n+1} are all over the place, and

they involve three neighboring grid points (T_{j-1}^{n+1}, T_j^{n+1}, and T_{j+1}^{n+1}), so we can't just solve in a simple way for T_j^{n+1}. This is an example of why implicit methods are harder than explicit methods.

fig:Crank

In the hope that something useful will turn up, let's put all of the variables at time level $n+1$ on the left, and all of the ones at level n on the right.

$$-D_{j-\frac{1}{2}}T_{j-1}^{n+1}+\left(\frac{2h^2}{\tau}+D_{j+\frac{1}{2}}+D_{j-\frac{1}{2}}\right)T_j^{n+1}-D_{j+\frac{1}{2}}T_{j+1}^{n+1}= \\ D_{j-\frac{1}{2}}T_{j-1}^{n}+\left(\frac{2h^2}{\tau}-D_{j+\frac{1}{2}}-D_{j-\frac{1}{2}}\right)T_j^{n}+D_{j+\frac{1}{2}}T_{j+1}^{n} \quad (8.7)$$

We know this looks ugly, but it really isn't so bad. To solve for T_j^{n+1} we just need to solve a linear system, as we did in Lab 2 on two-point boundary value problems. When a system of equations must be solved to find the future values, we say that the method is *implicit*. This particular implicit method is called the *Crank-Nicolson algorithm*.

To see more clearly what we are doing, and to make the algorithm a bit more efficient, let's define a matrix **A** to describe the left side of Eq. (8.7) and another matrix **B** to describe the right side, like this:

$$\mathbf{A}T^{n+1}=\mathbf{B}T^n \quad (8.8)$$

(T is now a column vector). The elements of **A** are given by

$$A_{j,k}=0 \quad \text{except for:}$$

$$A_{j,j-1}=-D_{j-\frac{1}{2}} \quad ; \quad A_{j,j}=\frac{2h^2}{\tau}+(D_{j+\frac{1}{2}}+D_{j-\frac{1}{2}}) \quad ; \quad A_{j,j+1}=-D_{j+\frac{1}{2}} \quad (8.9)$$

and the elements of **B** are given by

$$B_{j,k}=0 \quad \text{except for:}$$

$$B_{j,j-1}=D_{j-\frac{1}{2}} \quad ; \quad B_{j,j}=\frac{2h^2}{\tau}-(D_{j+\frac{1}{2}}+D_{j-1/2}) \quad ; \quad B_{j,j+1}=D_{j+\frac{1}{2}} \quad (8.10)$$

Once the boundary conditions are added to these matrices, Eq. (8.8) can easily be solved symbolically to find T^{n+1}

$$T^{n+1}=\mathbf{A}^{-1}\mathbf{B}T^n\,. \quad (8.11)$$

However, since inverting a matrix is computationally expensive we will use Gauss elimination instead when we actually implement this in Matlab (see Matlab help on the \ operator). Here is a sketch of how you would implement the Crank-Nicolson algorithm in Matlab.

(i) Load the matrices **A** and **B** as given in Eq. (8.9) and Eq. (8.10) above for all of the rows except the first and last. As usual, the first and last rows involve the boundary conditions. Usually it is a little easier to handle the boundary conditions if we plan to do the linear solve in two steps, like this:

Phyllis Nicolson, English (1917–1968) See this web site for a brief biographical sketch of Nicolson.

John Crank (1916–2006, English) See this Wikipedia entry for a brief biographical sketch.

```
% compute the right-hand side of the equation
r=B*T;

% load r(1) and r(N+2) as appropriate
% for the boundary conditions
r(1)=...;r(N+2)=...;

% load the new T directly into T itself
T=A\r;
```

Notice that we can just load the top and bottom rows of **B** with zeros, creating a right-hand-side vector r with zeros in the top and bottom positions. The top and bottom rows of **A** can then be loaded with the appropriate terms to enforce the desired boundary conditions on T^{n+1}, and the top and bottom positions of r can be loaded as required just before the linear solve, as indicated above. (An example of how this works will be given in the Crank-Nicolson script below.) Note that if the diffusion coefficient $D(x)$ doesn't change with time you can load **A** and **B** just once before the time loop starts.

(ii) Once the matrices **A** and **B** are loaded finding the new temperature inside the time loop is easy. Here is what it would look like if the boundary conditions were $T(0) = 1$ and $T(L) = 5$ using a cell-centered grid.

The top and bottom rows of **A** and **B** and the top and bottom positions of r would have been loaded like this (assuming a cell-center grid with ghost points):

$$A(1,1) = \frac{1}{2} \quad A(1,2) = \frac{1}{2} \quad B(1,1) = 0 \quad r(1) = 1 \tag{8.12}$$

$$A(N+2,N+2) = \frac{1}{2} \quad A(N+1,N+2) = \frac{1}{2} \quad B(N+2,N+2) = 0 \quad r(N+2) = 5 \tag{8.13}$$

so that the equations for the top and bottom rows are

$$\frac{T_1 + T_2}{2} = r_1 \quad \frac{T_{N+1} + T_{N+2}}{2} = r_{N+2} \tag{8.14}$$

The matrix **B** just stays out of the way (is zero) in the top and bottom rows.

The time advance would then look like this:

```
% find the right-hand side for the solve at interior points
r=B*T;

% load T(0) and T(L)
r(1)=1;r(N+2)=5;

% find the new T and load it directly into the variable T
```

```
% so we will be ready for the next step
T=A\r;
```

8.3 (a) Below is a Matlab program that implements the Crank-Nicolson algorithm. Download it from the class web site and study it until you know what each part does. Test `cranknicholson.m` by running it with $D(x) = 2$ and an initial temperature given by $T(x) = \sin(\pi x/L)$. As you found in Lab 7, the exact solution for this distribution is:

$$T(x,t) = \sin(\pi x/L)e^{-\pi^2 Dt/L^2} \tag{8.15}$$

Try various values of τ and see how it compares with the exact solution. Verify that when the time step is too large the solution is inaccurate, but still stable. To do the checks at large time step you will need to use a long run time and not skip any steps in the plotting, i.e., use a skip factor of 1.

Also study the accuracy of this algorithm by using various values of the cell number N and the time step τ. For each pair of choices run for 5 seconds and find the maximum difference between the exact and numerical solutions. You should find that the time step τ hardly matters at all. The number of cells N is the main thing to worry about if you want high accuracy in diffusion problems solved with Crank-Nicolson.

(b) Modify the Crank-Nicolson script to use boundary conditions $\partial T/\partial x = 0$ at the ends. Run with the same initial condition as in part (a) (which does not satisfy these boundary conditions) and watch what happens. Use a "microscope" on the plots early in time to see what happens in the first few grid points during the first few time steps.

(c) Repeat part (b) with $D(x)$ chosen so that $D = 1$ over the range $0 \le x < L/2$ and $D = 5$ over the range $L/2 \le x \le L$. Explain physically why your results are reasonable. In particular, explain why even though D is completely different, the final value of T is the same as in part (b).

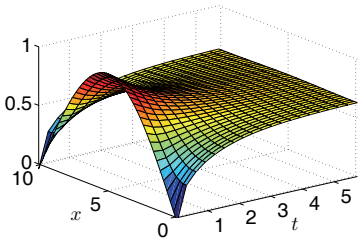

Figure 8.4 Solution to 8.3(b)

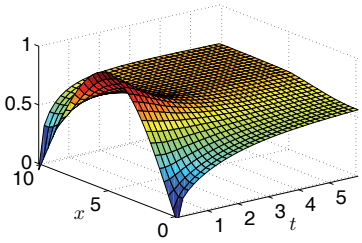

Figure 8.5 Solution to 8.3(c)

cranknicholson.m

```
clear;close all;

% Set the number of grid points and build a cell-center grid
N=input(' Enter N, cell number - ')
L=10;
h=L/N;
x=-.5*h:h:L+.5*h;
x=x';  % Turn x into a column vector.

%  Load the diffusion coefficient array (make it a column vector)
D=ones(N+2,1); % (just 1 for now--we'll change it later)

% Load Dm with average values D(j-1/2) and Dp with D(j+1/2)
```

```
Dm=zeros(N+2,1);Dp=zeros(N+2,1);  % Make the column vectors
Dm(2:N+1)=.5*(D(2:N+1)+D(1:N));   % average j and j-1
Dp(2:N+1)=.5*(D(2:N+1)+D(3:N+2)); % average j and j+1

% Set the initial temperature distribution
T=sin(pi*x/L);

% Find the maximum of T for setting plot limits
Tmax=max(T);Tmin=min(T);

% Choose the time step tau.
% The max tau for explicit stability is a reasonable choice
fprintf(' Maximum explicit time step: %g \n',h^2/max(D))
tau = input(' Enter the time step - ')

% Create the matrices A and B by loading them with zeros
A=zeros(N+2);
B=zeros(N+2);

% load A and B at interior points
const = 2*h^2 / tau;
for j=2:N+1
   A(j,j-1)= -Dm(j);
   A(j,j)  = const + (Dm(j)+Dp(j));
   A(j,j+1)= -Dp(j);

   B(j,j-1)=  Dm(j);
   B(j,j)  = const-(Dm(j)+Dp(j));
   B(j,j+1)=  Dp(j);
end

% load the boundary conditions into A and B
A(1,1)=0.5; A(1,2)=0.5; B(1,1)=0.;            % T(0)=0
A(N+2,N+1)=0.5; A(N+2,N+2)=0.5; B(N+2,N+2)=0; % T(L)=0

% Set the number of time steps to take.
tfinal=input(' Enter the total run time - ')
nsteps=tfinal/tau;

% This is the time advance loop.
for mtime=1:nsteps

    % define the time
    t=mtime*tau;

    % find the right-hand side for the solve at interior points
    r=B*T;

    % apply the boundary conditions
    r(1)=0;   % T(0)=0
```

```
    r(N+2)=0; % T(L)=0

    % do the linear solve to update T
    T=A\r;

    % Make a plot of T every once in a while.
    if(rem(mtime,5) == 0)
        plot(x,T)
        axis([0 L Tmin Tmax])
        pause(.1)
    end

end
```