# Lab 7

# The Diffusion Equation and Implicit Methods

## Analytic approach to the Diffusion Equation

Now let's attack the diffusion equation [1]

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2} \quad . \tag{7.1}$$

This equation describes how the distribution $T$ (often temperature) diffuses through a material with a constant diffusion coefficient $D$. The diffusion equation can be approached analytically via separation of variables by assuming that $T$ is of the form $T(x,t) = g(x)f(t)$. Plugging this form into the diffusion equation, we find

$$\frac{1}{D}\frac{\dot{f}(t)}{f(t)} = \frac{g''(x)}{g(x)} \tag{7.2}$$

The left-hand side depends only on time, while the right-hand side depends only on space, so both sides must equal a constant, say $-a^2$. Thus, $f(t)$ must satisfy

$$\dot{f}(t) = -\gamma f(t) \tag{7.3}$$

where $\gamma = a^2 D$ so that $f(t)$ is

$$f(t) = e^{-\gamma t}. \tag{7.4}$$

Meanwhile $g(x)$ must satisfy

$$g''(x) + a^2 g(x) = 0 \tag{7.5}$$

If we specify edge-value boundary conditions so that $T(x=0,t) = 0$ and $T(x = L,t) = 0$ then the solution to Eq. (7.5) is simply

$$g(x) = \sin(ax) \tag{7.6}$$

and the separation constant can take on the values $a = n\pi/L$, where $n$ is an integer. Any initial distribution $T(x, t = 0)$ that satisfies these boundary conditions can be composed by summing these sine functions with different weights using Fourier series techniques. Notice that higher spatial frequencies (i.e. large $n$) damp faster, according to Eq. (7.4). We already studied how to use separation of variables computationally in the first several labs of this manual, so let's move directly to time-stepping methods for solving the diffusion equation.

[1]N. Asmar, *Partial Differential Equations and Boundary Value Problems* (Prentice Hall, New Jersey, 2000), p. 110-129.

## Numerical approach: a first try

Let's try to solve the diffusion equation on a grid as we did with the wave equation. If we finite difference the diffusion equation using a centered time derivative and a centered second derivative in $x$ we get

$$\frac{T_j^{n+1} - T_j^{n-1}}{2\tau} = \frac{D}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{7.7}$$

Solving for $T_j^{n+1}$ to obtain an algorithm similar to leapfrog then gives

$$T_j^{n+1} = T_j^{n-1} + \frac{2D\tau}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{7.8}$$

There is a problem starting this algorithm because of the need to have $T$ one time step in the past ($T_j^{n-1}$), but even after we work around this problem this algorithm turns out to be worthless. We won't make you code it up, but if you did, you'd find that no matter how small a time step $\tau$ you choose, you encounter the same kind of instability that plagues staggered leapfrog when the step size got too big (infinite zig-zags). Such an algorithm is called *unconditionally unstable,* and is an invitation to keep looking. This must have been a nasty surprise for the pioneers of numerical analysis who first encountered it.

For now, let's sacrifice second-order accuracy to obtain a stable algorithm. If we don't center the time derivative, but use instead a forward difference we find

$$\frac{T_j^{n+1} - T_j^{n}}{\tau} = \frac{D}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{7.9}$$

This algorithm has problems since the left side of Eq. (7.9) is centered at time $t_{n+\frac{1}{2}}$, while the right side is centered at time $t_n$. This makes the algorithm inaccurate, but it turns out that it is stable if $\tau$ is small enough. Solving for $T_j^{n+1}$ yields

$$T_j^{n+1} = T_j^{n} + \frac{D\tau}{h^2}\left(T_{j+1}^n - 2T_j^n + T_{j-1}^n\right) \tag{7.10}$$

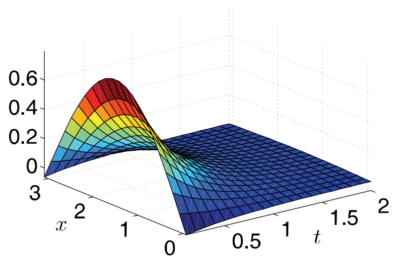

**Figure 7.1** Diffusion of the $n = 1$ sine temperature distribution given in Problem 7.1(a).

**P7.1** (a) Modify one of your staggered leapfrog programs that uses a cell-center grid to implement Eq. (7.10) to solve the diffusion equation on the interval $[0, L]$ with initial distribution

$$T(x,0) = \sin(\pi x/L) \tag{7.11}$$

and boundary conditions $T(0) = T(L) = 0$. Use $D = 2$, $L = 3$, and $N = 20$. You don't need to make a space-time surface plot like Fig. 7.1. Just make a line plot that updates each time step as we've done previously. This algorithm has a CFL condition on the time step $\tau$ of the form

$$\tau \le C\frac{h^2}{D} \tag{7.12}$$

Determine the value of $C$ by numerical experimentation.

Test the accuracy of your numerical solution by overlaying a graph of the analytic solution. Plot the numerical solution as points and the exact solution as a line so you can tell the difference. Show that your grid solution matches the exact solution with increasing accuracy as the number of grid points $N$ is increased from 20 to 40 and then to 80. You can calculate the RMS error using something like

```
error = np.sqrt( np.mean( (T - exact)**2 ))
```

(b) Get a feel for what the diffusion coefficient does by trying several different values for $D$ in your code. Give a physical description of this parameter to the TA.

(c) Now switch your boundary conditions to be insulating, with $\partial T/\partial x = 0$ at both ends. Explain what these two types of boundary conditions mean by thinking about a watermelon that is warmer in the middle than at the edge. Tell physically how you would impose both of these boundary conditions (specifying the value and specifying the derivative) on the watermelon and explain what the temperature history of the watermelon has to do with your plots of $T(x)$ vs. time.

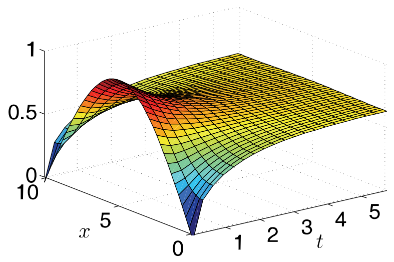

**Figure 7.2** Diffusion of the Gaussian temperature distribution given in Problem 7.1(c) with insulating boundary conditions.

Even though this technique can give us OK results, the time step constraint for this method is onerous. The constraint is of the form $\tau < Ch^2$, where $C$ is a constant. Suppose, for instance, that to resolve some spatial feature you need to decrease $h$ by a factor of 5; then you will have to decrease $\tau$ by a factor of 25. This will make your code take forever to run, which motivates us to find a better way.

## Implicit Methods: the Crank-Nicolson Algorithm

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

$$\frac{dy}{dt} = -y \tag{7.13}$$

**P7.2** (a) Write a program to solve this equation using Euler's method:

$$\frac{y_{n+1} - y_n}{\tau} = -y_n\,. \tag{7.14}$$

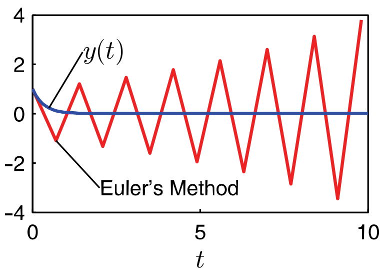

**Figure 7.3** Euler's method is unstable for $\tau > 2$. ($\tau = 2.1$ in this case.)

The program to solve for y using Euler's method is only a few lines of code (like less than 10 lines, including the plot command). Here are the first few lines:

```
tau = 0.5
tmax = 20.
t = np.arange(0,tmax,tau)
y = np.zeros_like(t)
```

Show by numerical experimentation that Euler's method is unstable for large $\tau$. You should find that the algorithm that is unstable if $\tau > 2$. Use $y(0) = 1$ as your initial condition. This is an example of an explicit method.

(b) Notice that the left side of Eq. (7.14) is centered on time $t_{n+\frac{1}{2}}$ but the right side is centered on $t_n$. Fix this by centering the right-hand side at time $t_{n+\frac{1}{2}}$ by using an average of the advanced and current values of $y$,

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

Modified your program to implement this fix, then show by numerical experimentation that when $\tau$ 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 $\tau$ in the denominator is the tip-off that this is an implicit method, and the improved stability is the point of using something implicit.

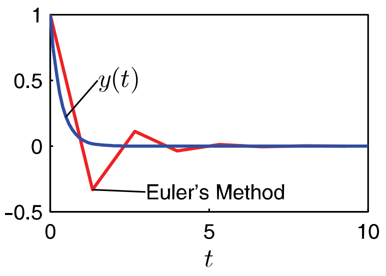

**Figure 7.4** The implicit method in 7.2(b) with $\tau = 4$.

(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. (7.14) (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 and it doesn't bounce between positive and negative values.

Show by numerical experimentation in a modified program that this fully implicit method damps even when $\tau$ is large. For instance, see what happens if you choose $\tau = 5$ with a final time of 20 seconds. The time-centered method of part (b) would bounce and damp, but you should see that the fully implicit method just damps. It's terribly inaccurate, and actually doesn't even damp as fast as the exact solution, but at least it doesn't bounce like part (b) or go to infinity like part (a). Methods like this are said to be "absolutely stable". Of course, it makes no sense to choose really large time steps, like $\tau = 100$ when you only want to run the solution out to 10 seconds.

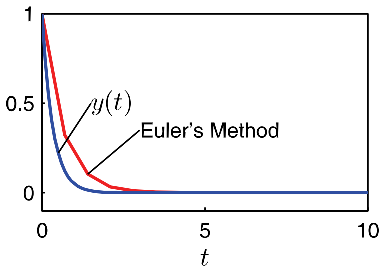

**Figure 7.5** The fully implicit method in 7.2(c) with $\tau = 2.1$.

## The diffusion equation with Crank-Nicolson

Now let's look at the diffusion equation again and see how implicit methods can help us find a stable numerical algorithm. Here's the equation again:

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2} \tag{7.15}$$

We begin by finite differencing the right side as usual:

$$\frac{\partial T_j}{\partial t} = D\frac{T_{j+1} - 2T_j + T_{j-1}}{h^2} \tag{7.16}$$

Now we discretize the time derivative by taking a forward time derivative on the left:

$$\frac{T_j^{n+1} - T_j^n}{\tau} = D\frac{T_{j+1}^n - 2T_j^n + T_{j-1}^n}{h^2} \tag{7.17}$$

**Phyllis Nicolson** (1917–1968, English)

**John Crank** (1916–2006, English)

This puts the left side of the equation at time level $t_{n+\frac{1}{2}}$, while the right side is at $t_n$. 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

$$T^{n+\frac{1}{2}} = \frac{T^{n+1} + T^n}{2} \tag{7.18}$$

like this:

$$\frac{T_j^{n+1} - T_j^n}{\tau} = D\frac{T_{j+1}^{n+1} + T_{j+1}^n - 2T_j^{n+1} - 2T_j^n + T_{j-1}^{n+1} + T_{j-1}^n}{2h^2} \tag{7.19}$$

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.

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.

$$-T_{j-1}^{n+1} + \left(\frac{2h^2}{\tau D} + 2\right)T_j^{n+1} - T_{j+1}^{n+1} = T_{j-1}^n + \left(\frac{2h^2}{\tau D} - 2\right)T_j^n + T_{j+1}^n \tag{7.20}$$

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. (7.20) and another matrix **B** to describe the right side, like this:

$$\mathbf{A}T^{n+1} = \mathbf{B}T^n \tag{7.21}$$

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

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

$$A_{j,j-1} = -1 \quad ; \quad A_{j,j} = \frac{2h^2}{\tau D} + 2 \quad ; \quad A_{j,j+1} = -1 \tag{7.22}$$

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

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

$$B_{j,j-1} = 1 \quad ; \quad B_{j,j} = \frac{2h^2}{\tau D} - 2 \quad ; \quad B_{j,j+1} = 1 \tag{7.23}$$

Once the boundary conditions are added to these matrices, Eq. (7.21) could be solved symbolically to find $T^{n+1}$

$$T^{n+1} = \mathbf{A}^{-1}\mathbf{B}T^n \ . \tag{7.24}$$

However, since inverting a matrix is computationally expensive we will use Gauss elimination instead as we did in lab 2 (with SciPy's `linalg.solve` function). Here is a sketch of how you would implement the Crank-Nicolson algorithm in Python.

- Load the matrices **A** and **B** as given in Eq. (7.22) and Eq. (7.23) for all of the rows except the first and last. Since the diffusion coefficient $D$ doesn't change with time you can load **A** and **B** just once before the time loop starts.

- The first and last rows involve the boundary conditions. Usually it is easier to handle the boundary conditions if we plan to do the linear solve of our matrix equation $\mathbf{A}T^{n+1} = \mathbf{B}T^n$ in two steps, like this:

```
import scipy.linalg as la

# matrix multiply to get the right-hand side
r = B@T

# set r as appropriate for the boundary conditions
r[0] = ...
r[-1] = ...

# Solve AT = r.  The T we get is for the next time step.
# We don't need to keep track of previous T values, so just
# load the new T directly into T itself
T = la.solve(A,r)
```

  With this code 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.

For example, if the boundary conditions were $T(0) = 1$ and $T(L) = 5$, the top and bottom rows of **A** and the top and bottom positions of $r$ would have been loaded like this (assuming a cell-center grid with ghost points):

```
# Set the A portion up where you define the matrix
A[0,0] = 0.5
A[0,1] = 0.5
A[-1,-1] = 0.5
A[-1,-2] = 0.5

... # skipped code

# Set the r portion of the boundary condition down in the time
# loop for each iteration
r[0] = 1
r[-1] = 5
```

so that the equations for the top and bottom rows are

$$\frac{T_0 + T_1}{2} = r_0 \qquad \frac{T_N + T_{N+1}}{2} = r_{N+1} \tag{7.25}$$

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

(iii) Once the matrices **A** and **B** are loaded, finding the new temperature inside the time loop is accomplished in the time loop by solving the matrix equation using the code fragment listed above.

**P7.3** (a) Write program that implements the Crank-Nicolson algorithm with fixed-edge boundary conditions, $T(0) = 0$ and $T(L) = 0$. Test your program by running it with $D = 2$ and an initial temperature given by $T(x) = \sin(\pi x/L)$. Try various values of $\tau$ 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.

(b) Now study the accuracy of this algorithm by using various values of the cell number $N$ and the time step $\tau$. For each pair of choices run for $t = 5$ s and find the maximum difference between the exact and numerical solutions. You should find that the time step $\tau$ matters less than $N$. The number of cells $N$ is the more important parameter for high accuracy in diffusion problems solved with Crank-Nicolson.

(c) Modify the Crank-Nicolson program 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. Zoom in on the plots early in time to see what happens in the first few grid points during the first few time steps.

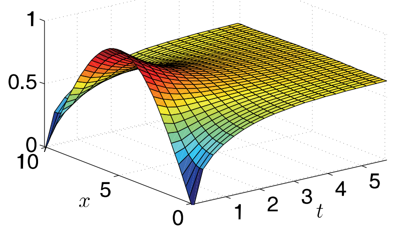

**Figure 7.6** Solution to 7.3(c)