Chapter 3

Implicit methods for 1-D heat equation

3.1 Implicit Backward Euler Method for 1-D heat equation

- Unconditionally stable (but usually slower than explicit methods).
- *implicit* because it evaluates difference approximations to derivatives at next time step t_{k+1} and not current time step we are solving for t_k.

$$\begin{aligned} U_{xx}(t_{k+1}, x_j) &= \frac{U_{j+1}^{k+1} - 2U_j^{k+1} + U_{j-1}^{k+1}}{\Delta x^2} \\ U_t(t_{k+1}, x_j) &= \frac{U_j^{k+1} - U_j^k}{\Delta t} \end{aligned}$$

$U_t = \beta U_{xx}$ becomes:

$$\begin{aligned} U_j^k &= U_j^{k+1} - \frac{\beta \Delta t}{\Delta x^2} \quad [U_{j+1}^{k+1} - 2U_j^{k+1} + U_{j-1}^{k+1}] \\ &= (1+2s)U_j^{k+1} - s(U_{j+1}^{k+1} + U_{j-1}^{k+1}) \end{aligned} \tag{3.1}$$

where $s = \frac{\beta \Delta t}{\Delta x^2}$ as before.
We still need to solve for U_j^{k+1} given U_j^k is known $\Rightarrow$ This requires solving a tridiagonal linear system of n equations.
Again we let $U_j^k = U(x_j, t_k)$; $x_j = j\Delta x, j = 0, ..., n+1, \Delta x = \frac{a}{n+1}$; $t_k = k\Delta t, k = 0, ..., m$, and $\Delta t = \frac{T}{m}$.

3.1.1 Numerical implementation of the Implicit Backward Euler Method

Again we are solving the same problem: $U_t = \beta U_{xx}, \quad U(x,0) = U_j^0 = f(x_j)$

3.1.2 Dirichlet boundary conditions

$U(0,t) = 1 = U_0^k, \quad U(a,t) = U(1,t) = 3 = U_{n+1}^k = U_4^k$

For simplicity we consider only 4 elements in x in this example to find the matrix system we need to solve for:

$x_0 = 0 \qquad x_1 = 0.25 \qquad x_2 = 0.5 \qquad x_3 = 0.75 \qquad x_4 = 1$

Rewriting $-s[U_{j+1}^{k+1} + U_{j-1}^{k+1}] + (1+2s)U_j^{k+1} = U_j^k$ as a matrix equation:

$$\underbrace{\begin{pmatrix} 1+2s & -s & 0 \\ -s & 1+2s & -s \\ 0 & -s & 1+2s \end{pmatrix}}_{\text{Tridiagonal matrix}} \underbrace{\begin{pmatrix} U_1^{k+1} \\ U_2^{k+1} \\ U_3^{k+1} \end{pmatrix}}_{\text{Solution } U \text{ at next time step}} = \begin{pmatrix} U_1^k \\ U_2^k \\ U_3^k \end{pmatrix} + s \underbrace{\begin{pmatrix} U_0^{k+1} \\ 0 \\ U_4^{k+1} \end{pmatrix}}_{\text{given from b.c.}}$$

$$\begin{aligned} A\vec{U}^{k+1} &= \vec{U}^k + \vec{b} \\ \Rightarrow \vec{U}^{k+1} &= A^{-1}[\vec{U}^k + \vec{b}] \end{aligned}$$

3.1.3 Mixed boundary conditions

The matlab code for this example is **BackwardEuler.m**.

$U(0,t) = 1 = U_0^k, \quad U_x(1,t) = 2 = \frac{\partial U_{n+1}^k}{\partial x}$

Using a leap-frog approximation for the spatial derivative:

$$\frac{\partial U_j^k}{\partial x} = \frac{U_{j+1}^k - U_{j-1}^k}{2\Delta x} + \mathrm{O}(\Delta x^2)$$

This is more accurate than the forward approximation we used previously (see section 2.2.1):

$$\frac{\partial U_j^k}{\partial x} = \frac{U_{j+1}^k - U_j^k}{\Delta x} + \mathrm{O}(\Delta x)$$

So for the *Neumann* boundary condition we have:

$$U_x(1,t) = \frac{\partial U_{n+1}^k}{\partial x} \quad \approx \quad \frac{U_{n+2}^k - U_n^k}{2\Delta x} = 2 \tag{3.2}$$

With $n = 4$, U_5^k is called a 'ghost point' because it lies outside the bar. So using equation 3.2 we define U_5^k:

$$\Rightarrow U_5^k = 4\Delta x + U_3^k$$

Because we have Neumann boundary conditions at $x = a(= 1)$, $U_{n+1}^k = U_4^k$ is unknown, and given by equation 3.1:

$$\begin{aligned} U_4^k &= -s[U_5^{k+1} + U_3^{k+1}] + (1+2s)U_4^{k+1} \\ &\text{use } U_5^{k+1} = 4\Delta x + U_3^{k+1} \\ \Rightarrow U_4^k &= -s[4\Delta x + 2U_3^{k+1}] + (1+2s)U_4^{k+1} \end{aligned}$$

Our system of equations becomes:

$$\underbrace{\begin{pmatrix} 1+2s & -s & 0 & 0 \\ -s & 1+2s & -s & 0 \\ 0 & -s & 1+2s & -s \\ 0 & 0 & \underbrace{-2s} & 1+2s \\ & & \text{Neumann b.c.} & \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} U_1^{k+1} \\ U_2^{k+1} \\ U_3^{k+1} \\ U_4^{k+1} \end{pmatrix}}_{\vec{U}^{k+1}} = \underbrace{\begin{pmatrix} U_1^k \\ U_2^k \\ U_3^k \\ U_4^k \end{pmatrix}}_{\vec{U}^k}$$

$$+ \underbrace{\begin{pmatrix} \text{Dirichlet b.c.} \\ \overbrace{sU_0^{k+1}} \\ 0 \\ 0 \\ \underbrace{4s\Delta x} \\ \text{Neumann b.c.} \end{pmatrix}}_{\vec{b}}$$

$$\begin{aligned} A\vec{U}^{k+1} &= \vec{U}^k + \vec{b} = \vec{c} \\ \Rightarrow \vec{U}^{k+1} &= A^{-1}\vec{c} \end{aligned}$$

Using an implicit solver means we have to invert the matrix A to solve for $\vec{U}^{k+1}$ which is a lot more computationally expensive than the matrix multiply operation in equation 2.7 to find $\vec{U}^{k+1}$ for explicit solvers in section 2.2.3. This method is stable for $s \geq 0$ so larger time steps can be used for implicit methods than explicit methods.

The matlab code for this example is **BackwardEuler.m**.

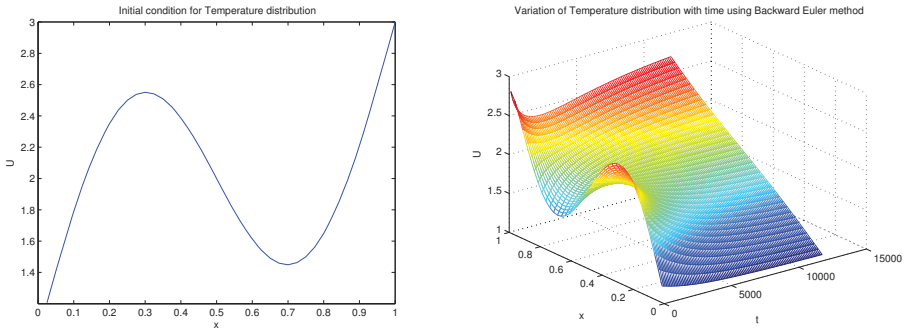

Figure 3.1: Initial conditions in (a) and matlab solution using Backward Euler method for temperature distribution along rod with time in (b)

Figure 3.1 shows the initial conditions in (a) and matlab solution for temperature distribution along rod with time in (b). The solution using the Backward Euler method in figure 3.1 is stable even for large time steps and this matlab code uses a time step 6× greater than the solution using the Forward Euler method in figure 2.1. So even though there is more work in each time step (inverting a matrix) using the implicit Backward Euler method it allows larger time steps than the explicit Forward Euler method.

3.2 Crank-Nicolson Scheme

- Average of the explicit (forward Euler) and implicit (backward Euler) schemes.

- Uses:

$$U_t = \frac{U_j^{k+1} - U_j^k}{\Delta t}$$

$$U_{xx} = \frac{1}{2}\left[\underbrace{\frac{U_{j+1}^{k+1} - 2U_j^{k+1} + U_j^{k+1}}{\Delta x^2}}_{\text{implicit}} + \underbrace{\frac{U_{j+1}^{k} - 2U_j^{k} + U_{j-1}^{k}}{\Delta x^2}}_{\text{explicit}}\right]$$

- Often used for simple diffusion problems.

Chapter 4

Iterative methods

Implicit methods are stable - however they can take much longer to compute than explicit methods. We saw that the Backward Euler method requires a system of linear equations to be solved at each time step:

$$A\vec{U}^{k+1} = \vec{c} \quad (\text{where:} \quad \vec{c} = \vec{U}^k + \vec{b})$$

where A is a tridiagonal matrix. How can we speed up this calculation? By using *iterative methods*.

Iterative methods:

- improve the solution of $A\vec{x} = \vec{b}$.
- use a direct method or a 'guess' for an initial estimate of the solution.
- useful for solving large, sparse systems (eg. tridiagonal matrix A in Backward Euler scheme).
- many different methods such as Jacobi, Gauss-Seidel, relaxation methods.
- iterative methods are not always applicable and convergence criteria need to be met before they can be applied. However they are ideal for finite difference methods (involving solution of large sparse matrices).

Iterative methods begin with an initial guess for the solution $\vec{x}^0$ to the matrix equation we are trying to solve: $A\vec{x} = \vec{b}$. Each iteration updates the new k^{th} estimate ($\vec{x}^k$) which converge on the exact solution $\vec{x}$. Different methods have different convergence times and for big inverse matrix problems are much faster than direct matrix inverse methods.