

C++ for Numerical Methods

Session 12

In this session you should modify your code from previous weeks to implement implicit methods, such as Backward-Euler and Crank-Nicolson. Your code should now be able to solve the heat equation in one space dimension with Dirichlet boundary conditions using Backward-Euler, Forward-Euler and Crank-Nicolson.

Objectives for Session 12

By the start of the next session, you should have completed the following:

- Implement and build an implicit PDE solver.
- Modify your basic PDE solver to solve the heat equation using the *theta* method. This means that by setting the value of *theta* we can choose the weighting of the implicit and explicit methods, where

$\theta = 0$ explicit

$\theta = 1$ implicit

$\theta = 0.5$ Crank-Nicolson

For our PDE solver code, we have already created both a `TriMatrix` class and a `Field` class. Our `TriMatrix` class stores a tridiagonal structure and is the discretized differential operator and the `Field` class stores the initial condition and solution. Now what we want to do is overload the `'/'` operator. Having been familiar with the back-slash operator in Matlab, where it directly solves the matrix inverse times a vector, i.e $A^{-1} * b$, we have no such routine in C++ and we need to create and solve this system ourselves - note however that we have to overload the forward-slash operator as we are not allowed to overload the back-slash operator in C++.

To do this, we need to implement a tridiagonal solve which operates on four arrays and an integer N (the size of the arrays). Three of the four arrays are the arrays stored in our `TriMatrix` class which represent the diagonal bands, and the fourth array is our `Field` object, i.e the solution. Our tridiagonal solve then solves for the unknown vector x where $Ax = b$. This is implemented at every time step. So you can clearly see that this does not differ much from the explicit case. In the explicit case, we implemented a matrix-vector **multiply** at each time step but for the implicit case we implement a matrix-vector **solve**.

Below is code to perform the tridiagonal solve and also shows the overloading of the `'/'` operator. You may incorporate this into your own program. It is not essential to understand the full details of the Thomas algorithm but you should completely understand what you are passing as arguments into the solver and the output, etc. The code is available for download from the usual webpage.

```

// This "/" operator is really going to be a tridiagonal solve
// as we are not going to really invert the matrix but solve it
// efficiently using the thomas algorithm/Tridiagonal solver
// @param x The RHS of the linear system to solve.

Field TriMatrix::operator/(const Field& x) const {
    return triSolve(x);
}

Field TriMatrix::triSolve(const Field &x) const {
    Field y(N);

    int j;
    double beta;
    double *gamma = new double[N];

    if (mDiag[0] == 0.0) {
        cout << "Error 1 in Tridag()" << endl;
    }

    beta = mDiag[0];
    y[0] = x.data[0] / beta;
    for (j = 1; j < N; j++) {
        gamma[j] = mUpper[j - 1] / beta;
        beta = mDiag[j] - mLower[j - 1] * gamma[j];
        if (beta == 0.0) {
            cout << "Error 2 in Tridag()" << endl;
        }
        y[j] = (x.data[j] - mLower[j - 1] * y[j - 1]) / beta;
    }

    for (j = N-2; j >= 0; j--) {
        y[j] -= gamma[j + 1] * y[j + 1];
    }

    delete[] gamma;
    return y;
}

```

The implicit scheme is derived as follows for time step $n + 1$ with ν is defined in the previous session:

$$\frac{V_i^{k+1} - V_i^k}{\Delta t} = \frac{V_{i-1}^{k+1} - 2V_i^{k+1} + V_{i+1}^{k+1}}{\Delta x^2}$$

$$-\nu V_{i-1}^{k+1} + (1 + 2\nu)V_i^{k+1} - \nu V_{i+1}^{k+1} = V_i^k$$

Therefore, at each time step we must solve the following linear system. Notice it is almost identical to the matrix introduced last week!

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ -\nu & 1 + 2\nu & -\nu & 0 & \dots & 0 \\ 0 & -\nu & 1 + 2\nu & -\nu & \dots & 0 \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & -\nu & 1 + 2\nu & -\nu & 0 \\ 0 & \dots & 0 & -\nu & 1 + 2\nu & -\nu \\ 0 & \dots & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \gamma_0 \\ V_1^{k+1} \\ \vdots \\ V_{N-1}^{k+1} \\ \gamma_N \end{pmatrix} = \begin{pmatrix} \gamma_0 \\ V_1^k \\ \vdots \\ V_{N-1}^k \\ \gamma_N \end{pmatrix}$$

Exercises

- Make sure your explicit Euler scheme is working from last week. Test it out by running your code for various values of ν , thus analysing the CFL stability condition of $\nu < \frac{1}{2}$.
- Derive the discretisation for the θ -scheme using the same approach as above for the implicit.
- Modify your program to solve the heat equation in one dimension using the θ -scheme. The Thomas algorithm for solving a tridiagonal matrix system is available for download from the usual website.
- Modify and play around with the value of ν for the implicit and Crank-Nicolson scheme. Compare this to the explicit case.