**Using Forwards and Backwards Euler Finite Difference Schemes to Solve the 2D Heat Conduction Equation**

Jorge Luis Mares Zamora

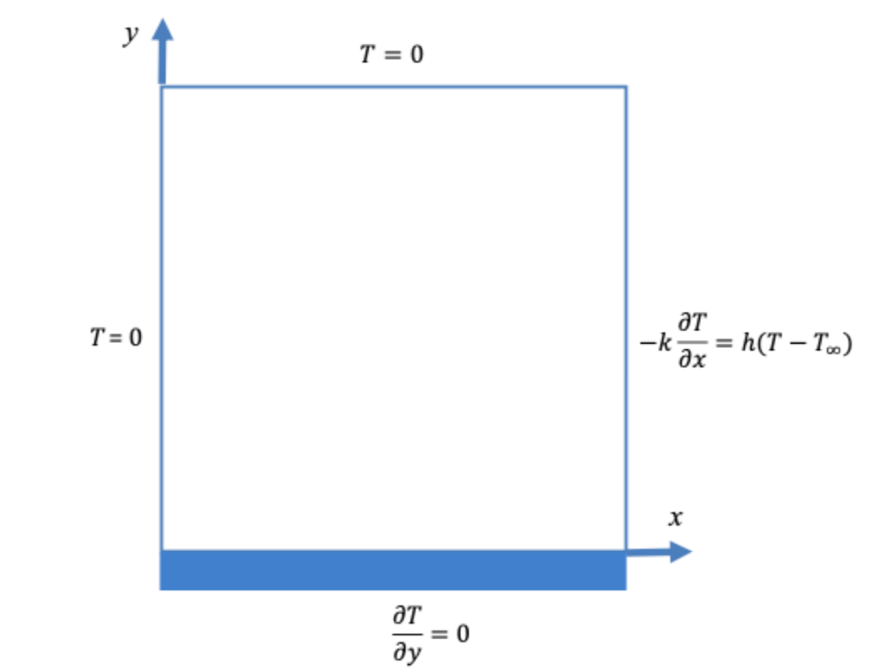
The University of Alabama in Huntsville

This project report presents the development of a solution for the 2D Heat Conduction equation using Forward Euler (Explicit), and Backward Euler (Implicit) time schemes. The approximations were calculated using code developed in MATLAB. The 2D Heat Conduction equation solved in this project is solved in a square domain with a side length, , discretized using a uniform grid, for which the boundary conditions were known at each of the edges of the domain. All of the known values and variables present in the equations are input parameters in the solver, so that the same code can be used to solve different problems with the same domain and boundary conditions, but different parameters and initial conditions.

1. **Introduction**

The Heat Conduction Equation describes how temperature changes over time and space within a material due to heat flow. The two-dimensional Heat Conduction equation then describes the Temperature changes within an idealized bar with infinite length.

The domain for this problem is a square (Figure 1) with side length, , where the initial temperature at each discrete point in the domain is known to be (this will serve as the Initial Conditions to our problem). The boundaries of our domain (each one of the edges of the Square) each behave differently. The North and West boundaries are kept at a fixed temperature, and , respectively, which means that their temperature stays constant throughout all of our time iterations. The East boundary has a convective boundary condition, described by Equation 2, where is the heat transfer coefficient and is the freestream temperature. The South boundary is kept insulated at all time, which means that the rate of change of the temperature normal to the boundary is equal to zero, described by Equation 3.



**Figure 1: Schematic of the Domain for the PDE.**

It is important to note that the Heat Conduction equation that we are trying to solve is transient, since the time derivative term is not equal to zero. Then, both time marching approaches to follow will be run iteratively either until a final specified time or a steady state solution is reached, depending on what the user requests in the input parameters.

1. **Methodology**

Arriving to the analytical solution of the Heat Conduction equation is usually reserved for special cases, as finding a solution for different boundary conditions can become very complicated very quickly. Then, a numerical approach can be used to find a solution to the equation, where the result is an approximation rather than an exact solution. This approximation can be found using finite difference techniques, where the first and second derivatives of a function over a discretized grid can be approximated with the following equations (the equation presented for the discretization of the first derivative of an arbitrary function, , is using a forward difference; however, it is important to note that similar approximations can be obtained for backwards and central differences).

1. **Forward Euler Time Scheme**

Using these discretizations for the first and second derivatives, we can substitute our exact differentials from Equation 1 with our approximated differentials for a given grid point at a time point . Additionally, since our grid uniformly spaced (i.e. ), we can simplify the resulting expression. Then, since all of the values of are known at the current time (since we are given an initial uniform T distribution equal to in the entire domain), it is possible to solve our discretized algebraic equation for , which is T at the next time interval. Additionally, we can define our Fourier number to simplify the equation even further.

This approach is the Forward Euler (Explicit) approach to finding the solution, as the T value of every grid point at the future time step is obtained using known values of T in the current time step (i.e. explicitly solving for the future time step).

In MATLAB, two matrices and , of size can be defined, where the latter are the x and y resolutions of the grid (which in our case should be the same). T is then initialized to have the value of the initial temperature for each one of its entries. A while loop is then started to iteratively solve for T, iterating through time. Initially, is solved for all the interior nodes (i.e all of the grid points that are not in the surface. Then the boundary nodes are modified one by one in order for them to satisfy the boundary conditions. On the West and North boundaries, we had a fixed temperature BC, so the equation at each of those nodes was the following:

The South boundary had an insulated boundary condition, mathematically expressed in Equation 3. Getting the approximation for this derivative using a forward difference scheme, the following expression can be derived:

Similarly, approximating Equation 2 using a backward difference scheme, we can arrive to the following expression for the convective boundary condition used for the nodes in the East boundary:

After modifying all of the values of T in the boundary nodes so that they are governed by their respective Boundary Condition equation rather than the interior node discretization of the PDE, the current iteration step is completed. At each iteration step a conditional is ran in order to check if the code should keep iterating. In the case where we are wanting to run the code for a specified time, , the conditional checks if the current is greater than or equal to . If true, it stops the iteration. If not true, it updates the current value using Equation 14, the current matrix using Equation 15, and runs the loop for another iteration. The conditional is similar for the case of looking for a steady state solution rather than the solution at a given time, but now the condition is expressed in Equation 16. This essentially checks that the difference between the elements of the old and new matrices is below a tolerance (i.e. the approximation is no longer changing and thus reached steady state conditions).

1. **Backward Euler Time Scheme**

The discretization of the original PDE presented in Equation 6, can also be attained by evaluating all the temperatures at the time step rather than at time , whilst keeping the discretization of the time derivative using a forward difference, as follows:

Solving for the current temperature, , and using the term we previously defined, the resulting expression is the following:

Here, we have an equation for each of the current grid points with terms of T that are currently unknown; however, since each one of our grid points in the domain has its own equation, we can arrive to the conclusion that we have an system of equations with unknowns, where is the number of grid points in our domain. However, since each of these equations is using information from all the neighbors of the current grid point, we must come up with new Equations for those grid points that do not have some neighbors (i.e. those points that lay on our boundary). Similar to the approach followed in the Explicit method, a discretization for the boundary condition equations can be obtained:

In MATLAB, the approach to solving the implicit method is slightly different than the one followed in the explicit approach. Since there is a need to solve a system of equations, all the equations coefficients will have to be stored in a matrix to then be solved. The coefficient matrix can be defined as , and to solve for the future temperature values, we will then have to solve a matrix equation. Following the Handout convention, in this equation is the matrix with the values at time step , and is the values with the values of T at the time step .

Since the matrix is of size , the values of and must be stored in a column matrix rather than a square matrix. The matrix to column matrix conversion followed the index conversion presented in Table 3.2 of the Computational Methods for Fluid Dynamics Textbook, where the index of the column matrix, , is defined as follows:

Then, a column matrix with all the initial values of was created, and the column coefficients were calculated. The coefficient matrix, , is a sparse matrix, which according to the textbook is better off stored as just the diagonals rather than a full matrix. Given MATLAB’s ability to work with matrices though, I decided to store my coefficient matrix as a sparse matrix, which managed all the storage issues whilst providing the flexibility to use the same workflow as with normal matrices (i.e. solving matrix equations and indexing using regular notation). With the equation coefficients matrix created (using the MATLAB function *spdiag*), the coefficients of the boundary conditions equations needed to be modified for the rows of the matrix that corresponded to the nodes in the boundary of our column matrix. The index of the nodes in the boundary change depending on the size of our square domain, so they were calculated using the following logic:

Arrays with the index elements of these sets were created and looping through each row of the matrix, when the row index matched a boundary index, the governing equation coefficients were changed from those in Equation 18 to those in Equations 19 through 22. Having set our coefficient matrix, our solver iteratively solved for new values of T, using the previously known values of C. At the beginning of each iteration, the C coefficients were changed to satisfy the boundary conditions, using the same indexing logic described above, and then the matrix equation was solved for using MATLAB’s backslash operator. After each iteration, the same logic used for breaking out of the loop in the Explicit method was used to determine if the iterations should continue or not.

1. **Results**

The developed program was run to solve the same problem using both discretization techniques discussed in the methodology section of this report. The parameters and initial conditions for the all of the results discussed on this report are the same, and are the following:

Additionally, to guarantee stability in the Explicit method, the value of was arbitrarily set to , and the value of our time step, , was calculated using the Fourier Number definition.

1. **Forward Euler Time Scheme**

The developed program was used to compute the solution of the PDE at time t = 0.5 and at the converged steady state. The program was ran using different grid sizes, of 10x10, 20x20, and 40x40 grid points each. The temperatures were plotted against x and y at the centerlines of the opposite axis, at a time t = 0.05 and steady state. These plots are presented in Figure 2 through Figure 5.

A graph of values and values

Description automatically generated

**Figure 2: T vs y for x = 0.5 at t = 0.05 (Explicit)**

**A graph of values and values

Description automatically generated**

**Figure 3: T vs y for x = 0.05 at steady state (Explicit)**

**A graph of values and values

Description automatically generated**

**Figure 4: T vs x for y = 0.5 at t = 0.05 (Explicit)**

**A graph of values and values

Description automatically generated**

**Figure 5: T vs x for y = 0.05 at steady state (Explicit)**

These results look accurate, as the values around the boundary conditions are satisfied for all of the boundaries. Furthermore, there is a big gap between the lines in the 10x10 grid and the 20x20 grid, but a smaller one between the 20x20 grid and the 40x40 grid, which may hint to a mesh convergence where the values slowly approach a solution as the grid gets larger and larger. Finally, it is important to note how it appears that both curves are “flattening out” as the simulation runs to a steady state solution. This flattening of the curve is a good signal as it points to the physical behavior that we would expect to see in a heat conduction problem.

1. **Backward Euler Time Scheme**

The program was used to solve the PDE at time t = 0.5 and at the converged steady state at the same grid sizes as the case for the Explicit Scheme. Similar plots to those presented in the previous section were created for this scheme, and the results are presented in Figure 6 through Figure 9.

A graph of values and values

Description automatically generated

**Figure 6: T vs y for x = 0.5 at t = 0.05 (Implicit)**

**A graph of values and values

Description automatically generated**

**Figure 7: T vs y for x = 0.5 at steady state (Implicit)**

**A graph of values and values

Description automatically generated**

**Figure 8: T vs x for y = 0.5 at t = 0.05 (Implicit)**

**A graph of values and values

Description automatically generated**

**Figure 9: T vs x for y = 0.5 at steady state (Implicit)**

Similar to the analysis to the explicit results, the plots show that the results satisfy the boundary conditions at all the boundaries. The flattening of the curves and the reduction of the gap between grid sizes are also a good sign as to the convergence to an accurate approximation. And the fact that the solutions for all the grid sizes basically converge on y = 0.5 is also a signal that our steady state solution is in fact accurate.

1. **Conclusions**

The 2D Heat Conduction equation is really complicated to solve analytically (if not outright impossible in some cases). However, using finite difference approximations allow us to get a close approximation, trading time in getting an exact solution with the computational power required to compute all these approximations. These finite difference methods do not give an exact solution to the problem; however, if implemented correctly, retain the physical information conveyed by the exact partial differential equations, and can be used to solve real life problems that may be too complicated to solve analytically. As it was practiced in this project, running different grid sizes is ideal in a case like this, as it can show that the accuracy of the approximations is converging to a point as the gap between grid points becomes smaller and smaller. Additionally, it is important to note that both the explicit and implicit methods both arrive to an acceptable solution, and that each of these methods has their own trade offs and pros, so knowing the strengths and weaknesses of each is vital to being able to do a good numerical analysis.

**References**

[1] Incroprera, F., *Principles of Heat and Mass Transfer*, Wiley, Chap. 5.

[2] Ferziger. J. H., Peric M., Street R. L., *Computational Methods for Fluid Dynamics,* 4th Ed., Springer, Chap. 3.

[3] Shotorban B, Lecture Notes MAE 623, Fall 2025