

1. Task: Heat diffusion on a rectangular plate

Bogy Balázs

2019 July 03.

1 Task

The task was to simulate heat diffusion on a rectangular plate with the given boundary conditions:

- on the left edge temperature is given, linear
- on the bottom there is a constant heat flux
- on the right and top edge there is perfect thermal insulation

The initial condition was that the plate has a linear temperature dependence along the x axis too in addition to the first boundary condition.

2 Theory

$$j = -\kappa \frac{\partial T}{\partial x}$$
$$\frac{\partial T}{\partial t} = \frac{\kappa}{C\rho} \Delta T \quad (1)$$

Where T is the temperature, j is the heat flux, κ is the thermal conductivity, C is the heat capacity, and ρ is density. For simplicity's sake in the simulation I set ρ , C , and κ to 1.

Taking a rectangular grid mesh we can discretize T .

$$T_{i,j} = T(i \times dx, j \times dx)$$

Where i and j are the indices of the grid points, dx being the distance between two neighbouring point. Then the Laplace operator (in infinitesimal limit) turns into:

$$\Delta T_{i,j} \approx \frac{T_{i-1,j} + T_{i,j-1} - 4T_{i,j} + T_{i+1,j} + T_{i,j+1}}{dx^2} \quad (2)$$

Of course on the boundary we have a special case, as at least one of these neighbouring grid points don't exist.

2.1 Insulation

There is no heat transfer between the point outside the mesh and the one inside. One could imagine that point being at the same temperature, so we could just remove that point from the Laplacian calculation along with a $-T_{i,j}$.

$$T_{-1,j} = T_{0,j}$$

$$\Delta T_{0,j} \approx \frac{T_{0,j-1} - 3T_{0,j} + T_{1,j} + T_{0,j+1}}{dx}$$

The same can be done in case multiple points are outside (corner).

2.2 Given temperature

This is the easiest, we don't calculate the Laplace operator for this point as its temperature is already given and there is no need to calculate anything to get its value in the next time-step.

2.3 Given heat flux

Simply ignoring the outside point and adding a constant heat/second to the system is not going to make a constant heat flux, as the point on the opposing side is going to modify it.

Instead lets calculate the temperature the imaginary grid point should have to make the the heat flux constant.

$$j = -\kappa \frac{\partial T}{\partial x}$$

$$\frac{j}{\kappa} \approx \frac{T_{1,j} - T_{-1,j}}{2dx}$$

$$T_{-1,j} = T_{1,j} - \frac{j2dx}{\kappa} \tag{3}$$

So we can just add a new line of grid points ($T_{-1,j}$) with this temperature and calculate for the real edge as if it was an inner part of the grid.

3 The program

I've written the program in C++, it requires `linbcg` from the Numerical Recipes codes. The plate's height/width ratio was set to 2. I used $dx=10^{-2}$, $dt=0.1 \times 10^{-4}$. I implemented both the explicit and implicit Euler-method.

| | |
|------------------------------|--|
| <code>T_left_0</code> | temperature in the top left corner |
| <code>T_left_gradient</code> | temperature gradient along left edge |
| <code>T_x_gradient</code> | temperature gradient along x axis in initial state |
| <code>j_bottom</code> | heat flux on bottom |
| <code>realWidth</code> | width of plate |
| <code>realHeight</code> | height of plate |
| <code>dx</code> | spatial discretization length |
| <code>dt</code> | time discretization length |
| <code>saveInterval</code> | Interval at which data file saves are made of the system |

Table 1: The variable parameters of the program

3.1 Explicit Euler-method solution

For the explicit Euler method we just discretize (1), use forward time and calculate the discretized Laplacian as seen in (2).

The solution is rather trivial, I used vectors of vectors (typedefed as `Grid`) for the temperature data, set it to the initial state, then stepped in dt (set to less than $\frac{dx^2}{2}$) units until the time reached the preset goal.

The tricky part is setting the derivatives properly. I created a function called `ExplicitEulerInnerStep`, this just calculates $T_{neighbour} - T_{i,j}$ in a given direction (it is templated, so optimizations are already made at compile time). After this I just had to divide the grid into 9 areas, the middle, the middle-edges, and the corners and give each one the directions it has neighbours in.

Of course we have to take into account the boundary conditions, at the left edge its trivial, $\delta_{left} = 0$. On the right and top it is already taken into account by only using directions that exist from there. And on the bottom simply add the `Up` direction instead of `Down` and the $-\frac{j^2 dx}{\kappa}$ term as seen in (3) .

3.2 Implicit Euler-method solution

For the implicit Euler-method first I needed to make the proper algebraic transformations to bring the equations into matrix form.

$$T^{n+1} = T^n + dt \frac{\kappa}{C\rho} \Delta T^{n+1}$$

$$T^{n+1} = T^n + \frac{dt}{dx^2} \frac{\kappa}{C\rho} (T_{i-1,j}^{n+1} + T_{i,j-1}^{n+1} - 4T_{i,j}^{n+1} + T_{i+1,j}^{n+1} + T_{i,j+1}^{n+1})$$

$$T^{n+1}(1 + 4S) - S(T_{i-1,j}^{n+1} + T_{i,j-1}^{n+1} + T_{i+1,j}^{n+1} + T_{i,j+1}^{n+1}) = T^n$$

Where $S = dt \frac{\kappa}{C\rho}$. This is a linear matrix equation, if we think of $T_{i,j}$ as T_n , where $n = jn_W + i$, where n_W is the number of cells along the width of the plate.

For the boundary we can use the previous methods, using 3 (or 2 for corners) instead 4 neighbours (1-3S instead of 1-4S and ignoring the appropriate $T_{x,y}$ term), and adding imaginary cells on the bottom for constant heat flux. Thus N , the number of cells is actually $n_W \times (n_H + 1)$ including the imaginary cells. n_W is `W`, n_H is `H` in the program.

So all that's left is creating the matrix, where the diagonal elements are $1 + n_{neigh}S$, and the neighbouring cells have $-S$ for each row (cell). This is a rather sparse matrix so I used `linbcg` to solve it.

The most difficult part was creating the proper matrix representation for `linbcg`. `InitGlobalSparseMatrix` initializes the matrix, it calculates the number of non-zero elements, to allocate the proper array sizes for `ija` and `sa` then one by one adds each cell's (that is a single row of the matrix) non-zero members to the sparse matrix. By default the `addCellToMatrix` function handles this, it sets its own diagonal value to $1 + n_{neigh}S$ and the value of each neighbouring cell in its row to $-S$ in `sa`, also setting the proper index into `ija`. The proper directions are set for each part of the grid (middle, edges, corners). Also of course the imaginary bottom edge cells don't need to be calculated as they will be re-assigned according to (3), and the left edge is constant so its derivative is simply 0.

After creating the matrix all that is left is for each time-step creating the right-hand side vector `B`, that is `T` with the properly modified imaginary bottom, and solving the linear equation with `linbcg`, and of course saving the data periodically.

It worth noting that the implicit solution is much slower using the same time-step but it is stable even with much larger time steps.

4 Output and visualization

I decided to make the visualization apart from the C++ program. This way it is not necessary to wait for the program to simulate the system again just to re-watch the simulation, other algorithms/programs can be used on the data without the need to reimplement them in the program. Also this way the playback speed can be properly set.

That is why I made a Python script to visualize the data I got from the program (saved as `<time>.dat` files at equal time intervals). It is important to mention that for the output to be saved there needs to be an "explicit_euler" and "implicit_euler" named folder next to the program. The data is saved in

simple tab separated matrix form.

The python script loads the data from the "explicit_euler" and "implicit_euler" folder and crates an mp4 animation out of it (the script has to be placed next to the folders/program). The script has dx as a parameter inside it to properly rescale the axes.