ECE4960 – Project Assignment 5
A Generic 1D/2D Parabolic PDE Solver for Heat Equation with Finite Difference

1. Internal Document

In this project, a generic parabolic PDE solver is implemented, which could take in any arbitrary initial values of any length, with the finite difference discretization rules of *forward*, *backward*, and *trapezoidal* Euler. This PDE solver is used to simulate and solve the naïve heat conduction equations:

General form: $\frac{\partial T}{\partial t} = \frac{K}{c_\rho \rho} \nabla^2 T$

1D form: $\frac{\partial T}{\partial t} = \frac{K}{c_\rho \rho} \frac{\partial^2 T}{\partial x^2}$

2D form: $\frac{\partial T}{\partial t} = \frac{K}{c_\rho \rho} (\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2})$

Dirichlet boundary conditions are adapted, and $\frac{K}{c_\rho \rho}$ is a constant whose value is around 0.1 - 0.3. In this project, $\frac{K}{c_\rho \rho}$ is set to be 0.15.

- **Program Design Breakdown:**

  The program is broken down into: ***PDESolver.java***, ***DiscretizationRule.java***, ***Main.java***, ***TestHelper.java***, and other inherited helper scripts including *FilIO.java*, *SparseMatrix.java*, *Vector.java*, *Jacobi.java*.

  a) ***PDESolver.java:*** A Java class, a generic 1D/2D parabolic PDE solver that takes in any arbitrary length of input, constant D, grid size h, delta_t, start and end time. It also interfaces with ***DiscretizationRule.java***, which provides different finite difference discretization rules for solving parabolic PDEs.

  b) ***DiscretizationRule.java:*** A Java interface, provides method handlers for different finite difference discretization rules including forward, backward, trapezoid Euler in both 1D and 2D.

  c) ***TestHelper.java:*** Implements white box testing for helper functions used in this project.

d) ***Main.java***: The main entrance for this project, invokes tests for helper functions, and validates heat equations' simulation results.

e) ***FilIO.java***: Imported from previous-written java project, auto-generates output files and test reports.

f) ***Jacobi.java***: Imported from previous-written java project, a generic Jacobi-iterative matrix solver, helps solve PDE equations.

g) ***Vector.java, SparseMatrix.java***: Imported from previous-written java project, the data structures for implementing PDE solvers. Instead of FullMatrix, SparseMatrix is used because it would be more space efficient when the input vector space is tremendously large.

- **Program Implementation:**

a) **Forward Euler rule:** Its implementation is different from the backward the trapezoidal Euler, therefore, it was implemented separately in ***PDESolver.java*** as solveForward(). The **1D** forward Euler PDE could be expressed as below:

$$n_{i,j+1} = n_{i,j}(1 - \frac{2D*\Delta t}{h^2}) + (n_{i+1,j} + n_{i-1,j})\frac{D*\Delta t}{h^2}$$

The left hand side equation represents time = j+1, and the right hand side represents time = j. Therefore, to calculate the temperatures for next time-step, we only need to implement $\boldsymbol{Temperature\ (t = j + 1) = Matrix\ A * Temperature(t = j)}$, where A's diagonal elements are $(1 - \frac{2D*\Delta t}{h^2})$, and $n_{i,j}$'s neighbors having the value of $\frac{D*\Delta t}{h^2}$ in A.

After creating matrix A, $\boldsymbol{Temperature\ (t = j + 1)}$ is calculated by A multiplying $\boldsymbol{Temperature\ (t = j)}$**.**

For **2D PDE**: The diagonal values are $(1 - \frac{4D*\Delta t}{h^2})$, and $n_{i,j}$'s neighbors are $\frac{D*\Delta t}{h^2}$.

b) **Backward and Trapezoidal Euler rule:** require solving matrixes, instead of simple matrix multiplication. The PDE equations of backward and trapezoidal Euler in **1D** could be respectively written as:

$$n_{i,j+1}\left(\frac{1}{\Delta t}+\frac{2D}{h^2}\right)-(n_{i+1,j+1}\ +\ n_{i-1,j+1})\ \frac{D}{h^2}\ =\ n_{i,j}(-\frac{1}{\Delta t})$$

$$n_{i,j+1}\left(\frac{1}{\Delta t}+\frac{D}{h^2}\right)-(n_{i+1,j+1}\ +\ n_{i-1,j+1})\ \frac{D}{2h^2}\ =\ n_{i,j+1}\left(\frac{1}{\Delta t}-\frac{D}{h^2}\right)+$$

$$(n_{i+1,j+1}\ +\ n_{i-1,j+1})\frac{D}{2h^2}$$

To calculate the temperatures for next time-step, we only need to calculate $\boldsymbol{Matrix\ A * Temperature\ (t=j+1) = Matrix\ B * Temperature(t=j)}$,

where A's diagonal elements are $(\frac{1}{\Delta t}+\frac{2D}{h^2})$, and $n_{i,j}$'s neighbors having the value of $\frac{D}{h^2}$ in A for Backward Euler rule, and $\left(\frac{1}{\Delta t}+\frac{D}{h^2}\right),\frac{D}{2h^2}$ for Trapezoidal Euler rule. B's diagonal value is $-\frac{1}{\Delta t}$ for backward Euler, $\left(\frac{1}{\Delta t}-\frac{D}{h^2}\right)$ for trapezoidal rule's diagonal value, and $\frac{D}{2h^2}$ for trapezoid rule's neighbors.

After creating matrix A and B, $\boldsymbol{Temperature\ (t=j+1)}$ is calculated by solving the matrix with Jacobi Matrix solver.

For **2D PDE**: A's diagonal values are $(\frac{1}{\Delta t}+\frac{4D}{h^2})$ and $\left(\frac{1}{\Delta t}+\frac{2D}{h^2}\right)$ for backward and trapezoidal Euler.

- **Program Validation and Testing strategy:**

The helper functions are tested with white box testing. Their test results and testing strategies are available in "TestHelperReport.txt".

The PDE solver is validated with Wilkinson's Principle, where a problem is approached with different methods, and by checking whether these methods have same or similar outputs, we could tell whether the program is performing correctly.

a) For 1D heat conduction equation:

The 1D heat conduction could be imagined as the heat is conducting through a thin pole. The initial condition is (0, 0, 0, 0, 0, 0, 10, 10, 10, 0, 0, 0, 0, 0, 0), meaning the center of the pole is hot at the beginning. The time and space plots of three discretization rules are as below:

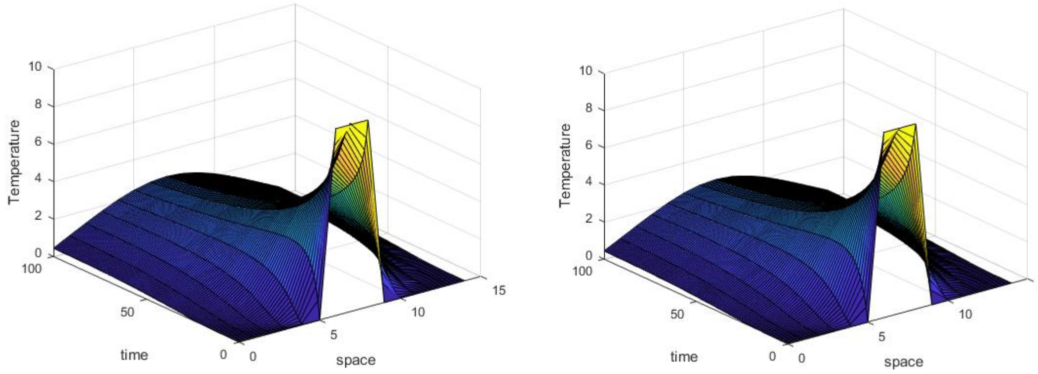


Fig. [1] Left: Backward Euler in 1D; Right: Forward Euler in 1D

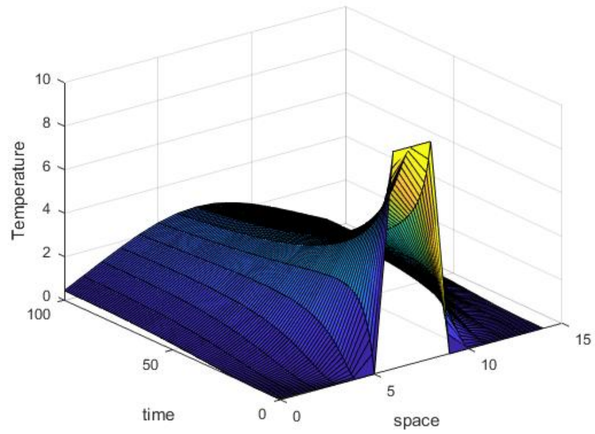


Fig. [2] Trapezoid Euler in 1D

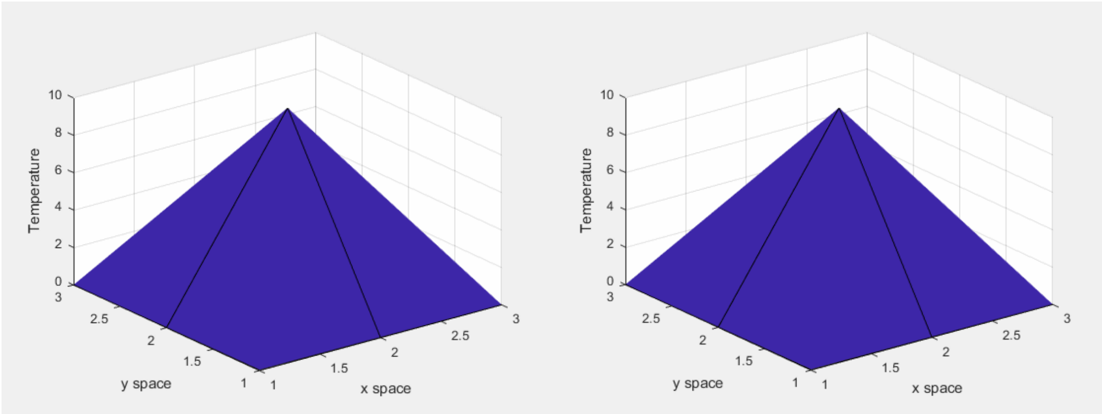
It could be seen that the temperature is attenuating with time, and the heat is conducted from the pole's center to the ends of the pole, which obeys the physical law.

Also, these three rule's outputs are very similar to each other, therefore, the 1D PDE solver is validated.

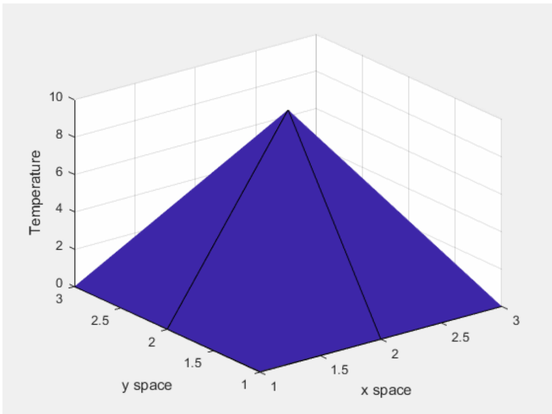
It's also worth mentioning, if you look at the actual output log files, the trapezoidal output has smoother changes in temperature, then the backward Euler, and the forward Euler's output has the least smooth changes.

b) For 2D heat conduction equation:

The 2D heat conduction could be imagined as heat is conducting within a plane. The initial condition is (0, 0, 0, 0, 10, 0, 0, 0, 0), meaning the center of the plane is hot at the beginning. The time and space plots are made into animations, which should be "animating" by looking at the .gif in my git repo:



Animation. [3] Left: Backward Euler in 2D; Right: Forward Euler in 2D
The animation should be animating in my git repo, in directory **Outputs**



Animation. [4] Trapezoid Euler
The animation should be animating in my git repo, in directory **Outputs**

It could be seen through the animations that the temperature is attenuating with time, and the heat is conducted from the center to the edges of the plane, which obeys the physical law.

Also, three rule's outputs are very similar to each other, therefore, the 2D PDE solver is also validated.

It's also worth mentioning, if you look at the actual output log files, the trapezoidal output has smoother changes in temperature, then the backward Euler, and the forward Euler's output has least smooth changes.

2. External Document

This program is designed to solve any arbitrary parabolic PDEs in 1D or 2D with finite difference discretization rules of forward, backward and trapezoidal Euler. The API is:

```
PDESolver.solve(new backward1D(), "1D", initial1D, h1, D1, dt1,
startTime1, endTime1);
```

To invoke the solver, users only need to indicate the discretization rule, the space dimension, the initial conditions, the space and time grid sizes of h and delta_t, the constant D, and the time range.

Then the program would automatically output the solutions into user's directory as .txt format files for further use.