CME211 Final Project–CG Solver on Steady-state Heat Equation

Chih-Hsuan Kao
chkao831@stanford.edu

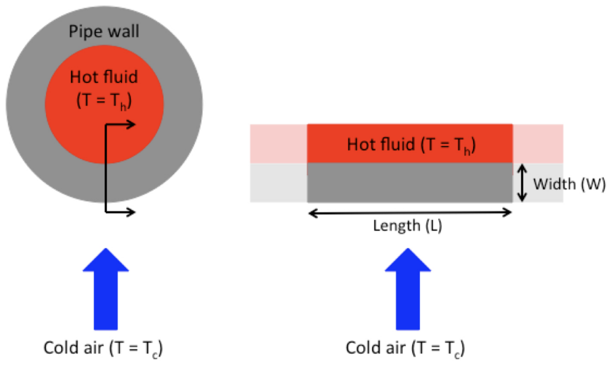
Stanford University

December 9, 2019

1 Introduction

In this project, I implemented a sparse matrix solver in C++ to solve the steady-state heat equation. The chosen iterative algorithm is the Conjugate Gradient (CG) method. Details of the CG Algorithm will be introduced in Section 5 of this writeup. Firstly, I discretized the pipe wall into an equally spaced Cartesian grid in 2D and assembled heat equations, further forming linear system based on the input text file. Details of the heat system are introduced in Section 2 of this writeup. Then, I solved the linear system $(-A)u = -b$ using the CG method. Finally, using the solution, the visualization task of the temperature distribution is taken care by the post-processing python file.

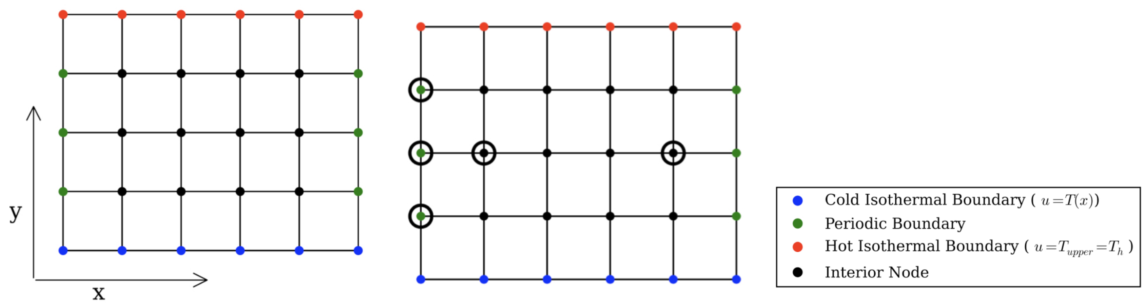
2 Steady-state Heat Geometry and Equations



The problem geometry is illustrated above. In the analysis, I discretized the pipe wall into Cartesian grid in which Δx = Δy = h. Such information is provided in the input text file. The discrete form of 2D heat equation at point (i, j) in the domain is

$$\frac{1}{h^2}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j}) = 0$$

By assembling the above equation at each point (figures are on next page) and by taking care of the boundary condition where periodic points (green points below) are treated as the same unknowns in the system, I formed a linear system $Au = b$ and solved the system using CG Algorithm. The cold (blue) and hot (red) isothermal boundary points are known, so the unknowns in the system would include black interior nodes as well as green periodic boundary points. Since in this form, matrix A is negative definite, I actually solved the system $(-A)u = -b$ instead in the implementation. The ordering of unknowns u in my system is in row-major, starting from the lower left to the upper right.


y
x
Cold Isothermal Boundary ($u = T(x)$)
Periodic Boundary
Hot Isothermal Boundary ($u = T_{upper} = T_h$)
Interior Node

3 Breakdown and Summary of the Whole Program

In `main.cpp`, I read in an input text file that contains information describing the geometry (length, width, h, T_c, T_h) of the pipe wall, then, I created a `HeatEquation2D` object. In `heat.cpp`, the `HeatEquation2D` object uses the input data to set up matrix A as `SparseMatrix` object with corresponding right-hand vector b. This is accomplished by utilizing and assembling the heat equation previously specified in section 2. The `SparseMatrix` object, implemented in the file `sparse.cpp`, contains vector attributes that describe a matrix and could be transformed from COO format to CSR format in place by calling the function COO2CSR() within `COO2CSR.cpp`. This is necessary because the CSR format is recommended for matrix in the CG Solver algorithm. Now, to solve the linear system $(-A)u = -b$, I implemented the CG Algorithm in the file `CGSolver.cpp`. In the algorithm, the basic operations on matrix and vectors are taken care by functions in the file `matvecops.cpp` as described in section X of this writeup. Finally, upon successfully solving the system using CG, the number of convergence is printed to the command line and the output series of solutions per every 10 iterations (including the first and last ones) are written into the specified output file from the user. The visualization of the temperature distribution within the pipe wall is implemented separately in the python file `postprocess.py`. The animation of the temperature distribution development during the CG solve (every 10 iteration) is further implemented in the `bonus.py`.

4 The Object-Oriented Programming (OOP) Design for Classes

The `HeatEquation2D` class is implemented in `heat.cpp` and the `SparseMatrix` class is implemented in `sparse.cpp`. For a `HeatEquation2D` object, it has private attributes such as `SparseMatrix() A`, corresponding to vector `b, x`, as well as grid information such as `length, width, h, matrixsize`, T_c, T_h, and T_x of the grid that are obtained and/or computed based on the input file. Those attributes are useful in calling CG-Solver in its Solve() function. In `CGSolver.cpp`, as a `SparseMatrix` object is passed in as a parameter, in order to obtain private attributes of this object, three getters are implemented. The private variables can only be accessed within the same class; hence, I implemented getters for row pointer i, column index j, and CSR value for nonzero entries in the sparse matrix. The concept of encapsulation is used here for better control of class attributes and methods as well as increased security of private data.

5 The Conjugate Gradient (CG) Algorithm in `CGSolver.cpp`

In `CGSolver.cpp`, the solver function takes in arguments of the linear system $(-A)u = -b$ as well as information of the output solution file, including solution prefix and the top/bottom isothermal boundary points (non-unknowns) of the system for system completion. Now, we are given the matrix -A in the form of three vectors in CSR format, the initial starting guess of ones for the solution vector, and the right-hand vector -b of the system $(-A)u = -b$. Also, the designated tolerance (threshold) for the CG algorithm is preliminarily set to be `1.e-5`.

Algorithm 1: The pseudo-code of the CG algorithm for my code

```
Use the passed-in $u$ as initial guess;
Initialize $r_0$ (initial residual) = $b - Au$;
Initialize $r$ (current residual at iteration n) and firstly be $r_0$;
Initialize $p$ as deep copy of the initial residual $r_0$;
Initialize $L2norm_{r0}$ as l2-norm of the initial residual $r_0$;
Initialize $niter$ (number of iteration) to be zero;
Declare $r_{next}$ (for usage at iteration n+1);
Declare $L2norm_r$ as current l2-norm in iteration;
while $niter < size(linearsystem)$ do
    $niter$ = $niter + 1$ ;
    $alpha$ = $(r^T * r)/(p^T * A * p)$ ;
    $u$ = $u + (alpha * p)$ ;
    $r_{next}$ = $r - (alpha * A * p)$ ;
    if $L2norm_r/L2norm_{r0} < threshold$ then
        break;
    end
    $beta$ = $(r_{next}^T * r_{next})/(r^T * r)$;
    $p$ = $r_{next} + beta * p$;
    update $r$ = $r_{next}$ ;
end
return niter;
```

6 The Decomposition of `matvecops.cpp`

As mentioned, on one hand, the main task of performing CG algorithm is included in the file `CGSolver.cpp` along with header file `CGSolver.hpp`. On the other hand, to support common vector and matrix operations, the breakdown of the task is included in the file `matvecops.cpp` along with the header file `matvecops.hpp`. There are five functions in the file.
The first function is `calculateL2Norm()`. This function takes in a vector and calculates its l2-norm as the return value. The second function is `calculateVecTrans()`. This function takes in two vectors and calculates $vec1^T * vec2$ and returns the value. The third function is called `matvecmult()`. The passed-in parameters are three CSR vectors of matrix A, right hand vector of $Ax = b$, and an integer that represents the row size of the square matrix. The function would eventually return the solution vector x as a double vector of the system $Ax = b$. The fourth function is `vecScalarMult()`. It takes in a (double) scalar and a vector and scales the vector. It eventually returns the vector. Last but not least, the fifth function is `vecvecAddition()`. It takes in two vectors and a boolean that indicates if it's an addition or a subtraction. If true, we perform vec1+vec2; if false, we perform vec1-vec2. The return value is a vector.

7 Users Guide

A makefile is implemented for the user's usage. To firstly compile source codes, run

```
$make
```

Then, to run the C++ and generate output, do

```
$./main <inputfile> <soln prefix>
```

After this, a series of solution files corresponding to the solution prefix would be generated in the directory. Also, if succeeded, the number of convergence would be printed out to the command line, such as

```
SUCCESS: CG solver converged in xx iterations.
```

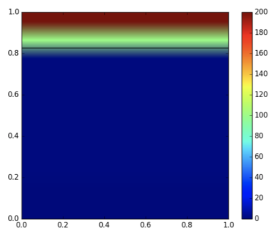
With the output text files, we could now perform visualization:

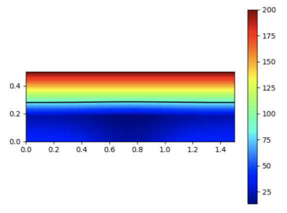
```
$python3 postprocess.py <inputfile> <soln file>
```
Here, the solution file refers to a single specific solution file that we just generated, for example,
```
$python3 postprocess.py input0.txt output9.txt
```
If succeeded, the pseudocolor plot with mean temperature isoline would then be shown and saved as <soln file>.png.
```
Input file processed:  XX.txt
Mean Temperature:  xx.xxxxx
```
would be printed to console.
Finally, I've also implemented animation of the temperature distribution development during the CG solve for every 10 iteration. To run the animation, do
`$python3 bonus.py <inputfile> <soln prefix>` where the argument pair is identical to the one while previously running `$./main <inputfile> <soln prefix>`.
Finally, one could remove `main`, all objects and editor files by executing `make clean`.

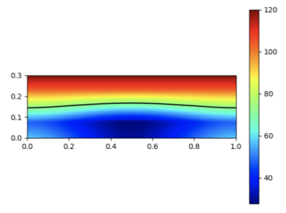
8 Output Showcase

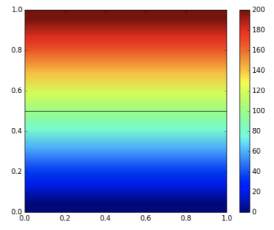
```
$make
$./main input0.txt output
$./main input1.txt output
$./main input2.txt output
$python3 postprocess.py input0.txt output0_1.txt
```
(...ommitted...)
```
$python3 postprocess.py input0.txt output2_157.txt
```

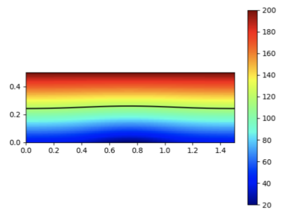
Selected output images:

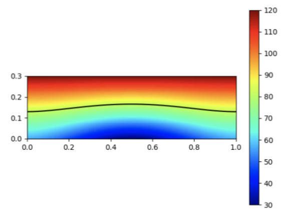

output0_1.png


output1_40.png


output2_60.png


output0_9.png


output1_132.png


output2_157.png

9 References

[1] LeGresley, P 2019, Final Project: Part 1 for CME 211: Software Development for Scientists and Engineers. http://coursework.stanford.edu. Stanford University, delivered 8 Nov 2019.
[2] LeGresley, P 2019, Final Project: Part 2 for CME 211: Software Development for Scientists and Engineers. http://coursework.stanford.edu. Stanford University, delivered 19 Nov 2019.