CME 211 Final Project Writeup

Mika Limcaoco

December 14, 2016

Summary of Overall Project

The goal of the overall project was to use a CG Solver algorithm to solve a steady-state heat equation. To do this, I was required to consider a system wherein hot fluid (with temperature T_h) would be transferred within a pipe. The pipe's exterior is kept cool and has a constant temperature of T_c, which is maintained by a series of cold air jets equally distributed throughout the pipe exterior. This project aims to solve for the mean temperature within the pipe walls. To do this, I analyzed one periodic section of the pipe wall.

A Conjugate Gradient method is used to solve the equation $Ax = b$ through multiple iterations. This is because the matrix A is symmetric positive definite and sparse.

Description of the CG solver implementation

The CG Solver Algorithm is implemented through 3 main programs: main.cpp, CGSolver.cpp, and matvecops.cpp. The main file runs the CGSolver and the COO2CSR.cpp file (this translates a matrix read from COO to CSR to be better used by CGSolver). It does this by getting an input matrix in COO format and translating it to CSR, then applying the CGSolver on it to solve for the matrix equation AX=B (x being the unknown). Matvecops has all the functions referenced in CGSolver. Once the X is solved for, it is written to a file (which is named after the second user-typed input) that serves as the solution. The number of iterations needed to get the solution is recorded and printed on the console.

I implemented my CG solver by implementing two different classes: a class that writes the solution (WriteSoln), and a class that solves the matrix through the given

CG Algorithm. WriteSoln merely writes the solution line by line through reading the values of x which are attained through the CGSolver class.

The goal is to solve $Ax = b$ for some sparse matrix A (which represents the heat equation) using the CG algorithm. In the CGSolver class, I first initialized variables to correspond to the pseudocode: vector r, vector Ax, and vector b. I then used the following functions, which are implemented in CGSolver but coded for in matvecops.cpp. I figured that these are the minimal number of functions I could use in order to make my code run most efficiently:

- lnormer: returns the square root of the dot product of two vectors
- multiplyvec: returns the product of two vectors when multiplied. One vector is the sparse matrix read from the matrix file – this means that it is processed through iterating the row pointers of the vector and appending to the product vector the value at that row pointer at the specified iteration multiplied by a value at the second vector (vector x) as defined by the first vector's column index-th (col idx) value in vector x. To clarify, the pseudocode of this formula would look like: Product Vector += ValueVector[iteration] * XVector[column index[iteration]].
- dotproduct: This takes in two vectors of equal length and iterates through the different nth elements of them, and multiplies these values through every iteration. A number is then returned.
- multiplycoeff: Takes a coefficient and multiples it with every value in a matrix, resulting in a new vector.
- addvec: iterates through every value of a vector and adds the nth value of both vectors together, resulting in a new vector.
- subtactvec: does the same ass addvec, but subtracts instead of adds.

I implemented the following functions throughout the CGSolver algorithm when they were necessary. For instance, when the algorithm called to multiple two vectors (e.g. to get the product of A and p, I would call multiplyvec on vectors A and p). It should be noted that these functions don't return the result vector directly – they take the result and modify it. This would be efficient for large matrices. To add to the efficiency, new variables such as the intialized U_n aren't used after one iteration. Thus, these values are replaced after each iteration.

Two classes were used to use the OOP design concept of C++. The first was the sparse class, which contained the structure of a sparse matrix. This class performed many operations and held several data structures in order for the program to work with sparse matrices. These operations included setting the matrix dimension, multiplying sparse matrices by a vector, etc. Another class was the HeatEquation class that served to set up and solve the heat equation system. This classes uses the sparse class, sets up the sparse matrix system through the input of a file outlining the pipe's characteristics, and solves Ax=B through the CG algorithm.

The pseudocode of the CG algorithm is decribed below:

CG Algorithm Pseudocode

initialize u_0;
$r_0 = b - Au_0$;
2-$norm(r0)$ = 2-$norm(r_0)$;
$p_0 = r_0$;
$niter = 0$;
while $niter < maxnumberofiterations$ **do**
  $niter = niter + 1$;
  $\alpha_n = (r_n^T r_n)/(p_n^T A p_n)$;
  $u_{n+1} = u_n + \alpha_n p_n$;
  $r_{n+1} = r_n - \alpha_n A p_n$;
  2-$norm(r)$ = 2-$norm(r_{n+1})$;
  if 2-$norm(r)$/2-$norm(r0) < threshold$ **then**
    break;
  end
  $\beta_n = (r_{n+1}^T r_{n+1})/(r_n^T r_n)$;
  $p_{n+1} = r_{n+1} + \beta_n p_n$;
end

Algorithm 1: Pseudocode of Conjugate Gradient (CG) Algorithm

User's Guide

To run the program, follow the following steps:

- 1. After implementing the `makefile` , you can compile through the following usage on the command line:

```
make
./main [inputfile name] [solution_prefix]
```

The solution files are the solutions for every guess (for every 10 iterations). The names of the solution files are in the following pattern:

```
solution_prefix * 1000 + number of iterations.txt
```

Running the program will lead to the console printing the convergence information.

A sample output of the C++ program is given below:

```
$ ./main input1.txt solution1
The CG algorithm does not converge..
```

- 2. The postprocessing and graphing of the solution are written in Python file `postprocessing.py`. This file finds the mean temperature in the pipe and visualizes the temperature distribution through a pseudocolor plot.

- The usage of the postprocessing procedure from the command line is: `python postprocessing.py [input filename] [solution filename]`

  A sample output should be something like (my file was unable to work):

```
Input file processed:  input1.txt
Mean Temperature:  123.456
```

Example of Visualization of the Heat Distribution

Unfortunatelly, I was unable to output a visualization of the heat distribution. But if it worked, it would be a square with an x axis from 0.0 to 1.0, and a y axis from around -0.2 to 0.8. There would be a scale on the side to show what colors correspond to what temperatures. The temperatures would be graphed, as color gradients, with a line overlaying the graph to represent the mean.

Reference

Nick Henderson, *CME 211: Project Part 1*(2015)
Nick Henderson, *CME 211: Project Part 2*(2015)