CME 211 Project: Part 2

Joshua Barnett

December 3, 2019

Introduction

The goal of the project is to create an OOP code that can solve the heat equation on a specific geometry under specified boundary conditions and the specifications are provided in [2]. This requires creating a sparse matrix class to form our block diagonal system in a efficient CSR format suitable for a conjugate gradient method to solve the system $Ax = b$. Finally, we save these results and process and visualize with a python script.

Description

Conjugate Gradient method

The conjugate gradient (CG) method for solving the linear system $Ax = b$ requires a few matrix and vector operations that can be generalized into a few operations and is fully specified in [1]. The most important function is a General Matrix Multiply (GEMM) that takes the form $c = \alpha Ax + \beta b$ where c, x, and b are vectors, α and β are scalars, and A is a CSR matrix. We can use this function for all of the matrix multiplication operations required in the CG method. Additionally, we also implement a weighted vector sum called daxby which is the expression $d = \alpha x + \beta y$. Finally, we have simple operations where we calculate the dot product and L_2 norm. These form the basis for all of the basic operations required to perform the CG method whose pseudo-code is defined below.

Data: Matrix A and vector b to solve $Ax = b$
Result: Solution for vector x in $Ax = b$
$u_n = \mathbf{1}$;
$r_n = b - Au_n$;
$l_{2,0} = \text{norm}_2(r_n)$;
$p_n = r_n$;
$n = 0$;
while $n < n_{max}$ **do**
  n++;
  $\alpha = r_n^T r_n / \left(p_n^T A p_n\right)$;
  $u_{n+1} = u_n + \alpha p_n$;
  $r_{n+1} = r_n - \alpha A p_n$;
  $l_{2,r} = \text{norm}_2\left(r_{n+1}\right)$;
  if $l_{2,r}/l_{2,0} < \mathit{tol}$ **then**
    break;
  end
  $\beta = r_{n+1}^T r_{n+1} / (r_n^T r_n)$;
  $p_n = r_{n+1} + \beta p_n$;
  $u_n = u_{n+1}$;
  $r_n = r_{n+1}$;
end

Algorithm 1: Conjugate Gradient Method for solving $Ax = b$

In order to facilate an OOP design, a few changes were made from the original code. Most importantly, the GEMM operation was altered to accept a new class called `SparseMatrix` instead of the list of `vector` objects.

`SparseMatrix` Design

The `SparseMatrix` class is broken down into three `vector` objects containing `int`, `int`, and `double` respectively as well as two `int` members to hold the size of the matrix and finally a `bool` value to specify if the matrix is in

CSR format or not. There a few methods implemented, including `Resize` which checks to make sure the new matrix sizes are compatible with current data. The `AddEntry` method ensures that the matrix is not already in CSR format before adding. The `ConvertToCSR` method simply calls the `COO2CSR` method and sets the `bool` for if it is CSR to `true`. In order to use the original GEMM method, a new method `CSR_GEMM` is defined as a `friend` of the `SparseMatrix` class so it can access the private members when calling the original GEMM method.

`HeatEquation` Design

The `HeatEquation` class contains the `SparseMatrix` defining the system of equations to solve $Ax = b$ as well as a few `vector` objects of type `double` to hold the RHS and proposed solution vector. There are two public methods, `Setup` and `Solve` that perform the basic operations of the class. `Setup` loads data from the provided input file and creates the sparse matrix A and vector b. The `Solve` class solves the system by calling the private function `CGIterate` which is copied from the `CGSolver` file inside the while-loop that way we have access to each iteration. It is too cumbersome to reuse the `CGSolver` method alone since it only provides the final answer and no intermediate results (like the r and p vectors).
The matrix we create unrolls the 2D grid of points into a linear vector as the variable x. Each gridpoint in general will depend on five points total: the gridpoint itself, one to the left and right, and one up and down. This is encapsulated in the discretization of the heat equation given in [2]

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

Thankfully, in constructing the matrix, we can simply add an entry for each term for the single point. This results in a banded sparse matrix. The boundary conditions are statisfied by only evaluating one side along the periodic boundary and otherwise only evaluating interior points (not on the isothermal boundaries). When a gridpoint references a boundary point, the value is instead added to the b vector when solving $Ax = b$ in the CG method.

User Guide

Compiling the `C++` code

After pulling the entire `project` directory, perform the following command:

```
$ [cd to project directory]
$ make
```

This will produce a `main` executable that we can use to product a solution given an input file and prefix. The input file has the following format: it consists of two lines—3 numbers in the first and 2 in the second.

```
[Length] [Width] [h]
[T_c] [T_h]
```

These define the physical extent of the system in the first line with the discretization given by h. T_c and T_h define the temperature scales for the isothermal layers at the bottom and top of the domain respectively. The program arguments are as follows: `./main [input file] [solution prefix]` Using the example inputs, we can run the following:

```
$ ./main input1.txt solution
SUCCESS: CG Solver converged in 132 iterations.
$ ls solution*
solution000.txt  solution040.txt  solution080.txt  solution120.txt
solution010.txt  solution050.txt  solution090.txt  solution130.txt
solution020.txt  solution060.txt  solution100.txt  solution132.txt
solution030.txt  solution070.txt  solution110.txt
```

To visualize these solution files, we can run the provided `postprocess.py` script which has the syntax `python3 postprocess.py [input file] [solution file]` and an example is given below.

```
$ python3 postprocess.py input1.txt solution132.txt
Input file processed: input1.txt
Mean Temperature: 116.286638
```

This also produces an image visualizing the temperature across the domain at the final solution which is shown in figure 1.

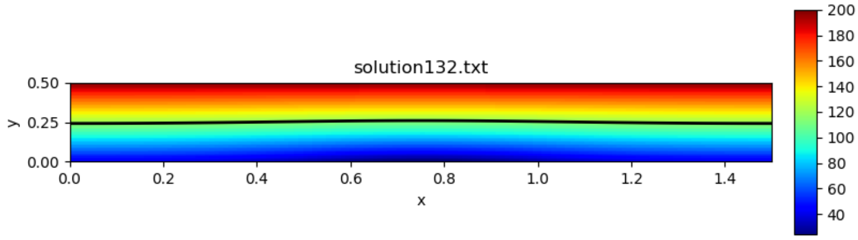

Figure 1: The solution along the domain at the 132nd iteration. The average temperature iso-line is drawn in black.

We can also get a video of the solution using the provided `bonus.py` script which has the syntax `python3 bonus.py [input file] [solution prefix]`. An example is shown below:

```
$ python3 bonus.py input1.txt solution
file: solution000.txt, average temp: 5.521208154785093
file: solution010.txt, average temp: 26.71285983378782
file: solution020.txt, average temp: 43.33173597065316
file: solution030.txt, average temp: 59.75282142319179
file: solution040.txt, average temp: 82.22304735748604
file: solution050.txt, average temp: 97.59372842487988
file: solution060.txt, average temp: 110.10907276977017
file: solution070.txt, average temp: 116.27863777431502
file: solution080.txt, average temp: 116.28282478898845
file: solution090.txt, average temp: 116.28504182573693
file: solution100.txt, average temp: 116.2860486560187
file: solution110.txt, average temp: 116.28645882352941
file: solution120.txt, average temp: 116.28661615374627
file: solution130.txt, average temp: 116.28663556680951
file: solution132.txt, average temp: 116.28663773535905
```

It also reports the files loaded and the associated average temperature reported.

References

[1] CME211, *Final Project: Part 1*, 2019.

[2] ———, *Final Project: Part 2*, 2019.