CME 211: Project Part 1

Due Tuesday, December 6, 2022 at 5:00 p.m. PST

This project was designed by Patrick LeGresley and modified for the purposes of this course.

Background

For the final project you will be developing a program to solve the 2D heat equation on a simple geometry using a sparse matrix solver written in C++. As a first step you will be implementing a sparse matrix solver in C++. In Part 2, you will be forming the system of equations for a specified geometry, solving the system, and writing Python code to visualize your results.

When solving partial differential equations, a commonly used method to solve the equation numerically is the *finite-difference* method. In this technique, the solution domain of interest is discretized into a grid of points and derivative terms are replaced by appropriate finite difference approximations, using stencils that involve information from neighboring points. Consider the steady-state heat equation in two dimensions:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0.$$

To discretize this equation on a Cartesian grid, we would use the following finite-difference approximations to obtain the 2$^{\text{nd}}$ derivatives at any grid location (indexed with integers i and j):

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_{i,j} = \frac{1}{\Delta x^2}(u_{i-1,j} - 2u_{i,j} + u_{i+1,j})$$

$$\left(\frac{\partial^2 u}{\partial y^2}\right)_{i,j} = \frac{1}{\Delta y^2}(u_{i,j-1} - 2u_{i,j} + u_{i,j+1})$$

Substituting these approximations into the PDE and assuming constant grid spacing ($\Delta x = \Delta y = h$) gives the following equation for each point (i, j) in the solution domain:

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

Assembling the equations for the grid points in your domain will lead to a linear system of equations $Au = b$.

Due to the compact nature of the finite difference stencils used in the discretization, systems of equations derived using FD are generally quite sparse. Additionally, for certain geometries (like the one you will be dealing with in Part 2), the resulting system will be symmetric positive/negative definite. To solve systems with this structure, an efficient iterative algorithm to use is the Conjugate Gradient (CG) method. The pseudo-code for the CG algorithm is given in Figure 1.

Part 1

For this first part of the project your task is to implement the CG algorithm in C++ as a function within the file `CGSolver.cpp`. A prototype of this function can be found in the provided `CGSolver.hpp` header file. You should use CSR as the sparse matrix format in your solver, as this format is better suited to the matrix-vector operations that arise in CG.

However, building a matrix in COO format is much more straightforward and that is the format you will use in the second part of the project. To assist you with the required conversion, a header file `COO2CSR.hpp` has been provided for you which contains a function `COO2CSR()` that can be used to convert an existing COO matrix to CSR format in-place (i.e. the provided function will overwrite your input vectors and fill them in with appropriate data for the CSR format).

```
initialize $u_0$
$r_0$ = b – A $u_0$
L2normr0 = L2norm($r_0$)
$p_0$ = $r_0$
niter = 0
while (niter < nitermax)
   niter = niter + 1
   alpha = ($r_n^T$ $r_n$)/($p_n^T$ A $p_n$)
   $u_{n+1}$= $u_n$ + $alpha_n$ $p_n$
   $r_{n+1}$= $r_n$ - $alpha_n$ A $p_n$
   L2normr = L2norm($r_{n+1}$)
   if L2normr/L2normr0 < threshold
      break
   $beta_n$ = ($r_{n+1}^T$ $r_{n+1}$)/($r_n^T$ $r_n$)
   $p_{n+1}$ = $r_{n+1}$ + $beta_n$ $p_n$
```

Figure 1: Conjugate Gradient pseudo-code

When implementing your solver you should develop additional functions to perform common vector and matrix operations that occur in the CG algorithm. These functions should be implemented in a file `matvecops.cpp` with a corresponding include file `matvecops.hpp` for the prototypes.

To test and debug your solver implementation two test matrices in COO format are provided in the files `matrix1.txt` and `matrix2.txt`. In each file the first line contains the number of rows and columns in the matrix. For subsequent lines, the values are the row index, column index, and the value of the matrix entry.

To demonstrate the use of your solver write a `main()` in `main.cpp` that loads a matrix from a file specified at the command line, converts the matrix to CSR format, runs your CG solver function with a starting guess of ones for the solution and zeros for the right hand side, and writes the solution to the specified file name. Use a tolerance of `1.e-5` and if your solver is running properly, the solution should be a vector of zeros:

```
$ ./main matrix2.txt solution2.txt
SUCCESS: CG solver converged in 25 iterations.
```

Adherence to Prototypes

We provide you with header files, and we ask that you use them as a starting place. You are encouraged to consider adding use of keyword `const` to arguments when appropriate.

Preliminary write up

In LaTeX, create a `writeup.tex` that uses the algorithm environment to provide the pseudo-code for the CG algorithm. Also write a short (no more than one page) discussion of how you implemented your CG solver in terms of functions to eliminate redundant code.

Summary of requirements

1. Create a directory called `project` in your CME 211 homework repository. All source files and documentation will go in this directory.
2. Write a function in `CGSolver.cpp` that solves a linear system in CSR format using the CG method. The implementation of this function must be consistent with the provided prototype in `CGSolver.hpp`.
3. For common operations in your solver algorithm, develop vector and matrix functions in `matvecops.cpp` and create the associated header file `matvecops.hpp` with the prototypes.
4. Write a `main()` in `main.cpp` that loads a matrix from a file in COO format, converts the matrix to CSR format using the provided `COO2CSR()` function, and solves the system using your `CGSolver()` function. Write the solution vector to the specified solution file: one value per line, scientific notation, 4 decimal places.
5. Write a `makefile` to compile all source code and produce the `main` executable.
    - `$ make`: must compile all source code and produce `main`
    - `$ make clean`: must remove all object (`*.o`) files and remove `main`

Command line interface

Your build system must produce an executable named `main` that operates according to the following examples:

- provide a usage message if there are no command line arguments:

```
$ ./main
Usage:
  ./main <input matrix file name> <output solution file name>
```

- with files specified:

```
$ ./main matrix2.txt solution2.txt
SUCCESS: CG solver converged in 25 iterations.
```

Submission

Please be very careful with directory and file names. To be clear you should have a minimum of these files in the `project` directory in your GitHub repository:

- `CGSolver.cpp`
- `CGSolver.hpp`
- `COO2CSR.cpp`
- `COO2CSR.hpp`
- `main.cpp`
- `matvecops.cpp`
- `matvecops.hpp`
- `writeup.tex`
- `writeup.pdf`
- `makefile`

Do not commit:

- temporary files produced by your text editor
- object files produced by compiler (`*.o` files)
- the binary executable (`main`)

We will use a Git tag to mark the commit that you want graded. A Git tag is essentially a bookmark to a particular commit. After you have committed the version of the code you would like graded, create a project1 tag with the following command:

```
$ git tag -am "Project-part-1 Submission" project1
```

or

```
git tag -m "Project-part-1 Submission" -a project1
```

The quotes contain a tag message. In the above command project1 is the actual tag. You can push commits along with tags to the remote repo with:

```
$ git push --tags origin main
```

It is a good idea to check if the tags show up on GitHub. These appear in the "release" tab on the web interface.

It is possible to change your tag later, too, if you discover a bug after pushing:

```
$ git tag -d project1 # delete the project1 tag pointing to old commit

$ git push --delete origin project1 # delete the remote project1 tag

$ git tag -am "My Project-part-1" project1 # create project1 tag pointing to current commit

$ git push --tags origin main # push changes to code and tags
```

We will use Git time-stamps on tags and the associated commits to check deadlines. Be careful with overwriting tags – you don't want to lose the submission tag.