Modular Testing in Sparse Matrix Solvers

Modular design

File organization

In this assignment, we implemented the Jacobi iterative solver with a bottom up approach. The whole project is divided into small modular functions, main method, and tests.

In order to increase program efficiency, we utilized two data structures: array and list. For operations that return matrix with fixed sparsity such as multiplication of a matrix with a vector, we use array. For operations that involve removing and inserting elements such as row scale and extracting triangular matrices, we used list. The first set of helper function we wrote is list.cpp. List.cpp include functions such as conversion between list and array, compare elements in two arrays, etc. Then we created functions in crsOperation.cpp. This file include operations on compressed row sparse matrix, such as retrieve an element. The next level is jacobiOperation.cpp, where we implemented functions such as retrieve triangular matrices and matrix mulitplication. Then we implemented the actual Jacobi method in jacobi.cpp. Norms.cpp which can find various vector norms was written to support jacobbi.cpp. It also include other small helpful functions. In the end, we used only a fraction of the total helper functions that we wrote, but each helper function improved our understanding of the whole structure and give insight to the next step. For example, function such as print matrix greatly improved debugging speed.

Tests

Method test

We wrote modular tests as we wrote each helper functions. Small functions such as swap array were repeatedly tested with preknown conditions. There is a main test function for functions in every file. Then there is also modularTests.cpp which calls each of those test functions. One can easily call modularTest in main function and see if there is a problem with any of the helper functions.

Main function test

The main jacobi method was tested by comparing the result with a direct matrix solver, utilizing Wilkison Principle. Since we cannot use the direct matrix solver on the large matrix, we used a small 5x5 matrix to test result. The two methods returned results within small precision error. We also found the residual vector norm by backward substitution to ensure precision. The test outputs are given in tests_output.txt.

Result

The final result is given in main_output.txt. The first implementation of our jacobi was extremely slow, seems to take over hours. We realized that the problem is we were re-allocating arrays for too many times. Originally, we used a function called deleteValue, this function would delete a single array component and return a new array. So every time we modify a single element, a new array has to be created which is very time consuming. We fixed this problem by first converting the array to list, do all the insertion and deletion in list format, and then convert back to array format. In this way, a new array is only allocated once every major operation. Then in array format, we can perform the fixed sparsity operations. This change dramatically improved run time. After the change, one Jacobi method takes only 30-60 seconds.

Review and Further Improvement

One thing that could have increased code efficiency and reusability is implementing a class for the row compressed matrix. In our project, we had to call parameters such as value array pointer and value array size for most of the functions. A class would reduce a lot of those function call parameters.

Another improvement is that we could have written more tests. We can implement LU decomposition, and compare its result with the other methods that we have written. We could also have more tests on larger matrices, and check if there are still places where we can reduce array reallocations and loops, and improve speed.

Due to the non-diagonal domination issue, the matrix we solve may not be convergent all the time. When the matrix is divergent, the Jacobi iterative solver need to be improved to SOR solver as we implemented in hacker practice, the parameter, omega, has to be set appropriately to make sure the solution converge.