**

**ECE 4960: Computational and Software Engineering**

**Spring 2019**

**Programming Assignment 2: Modular Testing in Sparse Matrix Solvers**

**1. Goal**

1. Understand the operations and computational efficiency in sparse matrix operations
2. Implement matrix operations in sparse and full matrix formats for debugging purposes on known small matrices by different data structure
3. Implement matrix solvers with modules that have independent testings

**1.1 Before you start**

* You are allowed to use any platform and math libraries, but you **have to** implement the sparse matrix methods explicitly with row-compressed formats. This is the one time in this class that data structure is actually prescribed, due to the importance to computational efficiency and verification from different data structures. For examples, from the different full and sparse matrix data structures, you can learn the Wilkinson testing strategies.
* You do not need to use the extravagant numerical methods for the best solution or computational efficiency. The main purposes of this assignment are two: (1) How to write modular testing, specifically when the ground truth may not be known? (2) How to deal with problems with high sparsity with high computational and memory efficiency. This is NOT an assignment about linear algebra or extensive schemes in matrix conditioning for direct and iterative solvers.
* You can use the built-in exception handling by the supported IEEE floating-point standards (i.e., no explicit check for exceptions, as you know how exception is propagated from your previous assignment). If explicit checks for exception are used, they should be able to be turned off by a global DEFINE so that the efficiency comparison is more meaningful.
* You have learned the Wilkinson principle for debugging, and apply as much as possible to this program assignment for verification. Debugging can possibly be done by tracing your source code carefully (in debuggers) time after time (not ideal or sufficient), but debugging by writing a redundant or alternative code (such as the full matrix representation and tests in Homework 4) achieve something different (and probably better than spend long time in debuggers). Although the Wilkinson tests do not tell you exactly which line is wrong, the redundant tests can indicate the error source and can be executed and reported automatically! This will become VERY useful when you deal with large codes!
* Load the sparse matrix ‘mat1’ from the class blackboard site (rank: 5,000, number of non-zero entries: 253,677). It is stored in compressed row storage format with row pointer, column indices and values saved in rowPtr.csv, colInd.csv and value.csv respectively. Row/column pointers use 1 as the starting index, unlike the 0 indexing in C/C++.

**1.2 Tasks**

* You will choose to implement an iterative solver (either Jacobi or Gauss-Seidel). Notice that the Gauss Seidel method can be implemented by using the already available *x(k)* values for the lower elements, instead of constructing the inverse of the lower triangular matrix. The iterative solver, without implementing the minimum fill-in symbolic LU factorization is expected to have much shorter codes than the direct solver, but may have more convergence problems for some cases where the diagonal dominance or largest eigenvalues of the matrix cannot be guaranteed.
* We will construct a reliable solution for a sparse matrix with different levels of testing and verification, including the modular testing you developed in Homework 4. First, try to achieve solution for the matrix mat1 with the following *b* vectors (The first two *b* vectors have a 1.0 in 1st, and 5th element, and 0 otherwise. The third *b* vector has 1.0 in all elements):

|  |  |  |
| --- | --- | --- |
| *b* = (1.0, 0, 0, …,0)T | *b* = (0, 0, 0, 0, 1.0, 0,…,0)T | *b* = (1.0, 1.0 ,.…,1.0)T |

* For the final validation, report the normalized residual norm for each of your solution corresponding to the three matrices and the above *b* vectors:



**Additional Reminders:**

* Maintain proper testing and modular design with the above implementation.
* Report the normalized residual norm and comment if the additional implementation achieves better results, why or why not. Check the additional memory and computation requirement.

**2. Submission and grading**

* A short report (< 2 pages ideally) that documents your modular design and test strategies. Also discuss briefly the results achived, reasons and possible improvements for any convergernce issues, if observed.
* The grading will be evaluated by the following points:
  + Your design of the modular testing to verify your results, not just on the final results, but your test suites for all of your sparse matrix methods. The more comprehensive and efficient your test is, the higher the grade (40%).
  + Your understanding of the matrix operations and the residual norm you are able to achieve (notice that the residual norm ||*b – Ax*|| might not be small for ill-conditioned problems) (20%).
  + Computational time and memory usage checks (10%). You should use your estimate of number of operations to compare with your measured time and memory. We will get a sense of the compiler or interpreter efficiency.
  + How your design can be used as library functions to others and to your future projects (10%).
  + The in-code documentation and short report (10%).
  + Additional task including the algorithm with modular implementation, testing and reporting of results (10%).
* A lab session on 3/6 7:30pm (Wed.) will be used to check and demonstrate your design and programming.
* All submission should be on your Git hub before end of day of 3/10 (Sun.). Remember that you should only check in source codes, tests and reports.

**3. Further information for future exploration**

We have abundant matrix solvers in the free and licensed software. This assignment is a practice of modular programming/testing and the demonstration of handling sparse structures with memory and computational efficiency. You are welcome to implement more types of solvers, once all of the basic matrix methods are implemented. The best API can be designed when you intend to support all kinds of matrix solvers.

You may be tempted to try to solve an *Ax = b* problem you have written above for the Memplus sparse matrix in Homework 4. However, most likely your solver will fail as better conditioning needs to be applied for the direct and iterative solvers to work on Memplus, which is mostly due to the almost filled rows and columns derived from the data bus lines. The previous Mat1 large matrix has already gone through the diagonal dominance conditioning so that your solver at this level can work. As we have no time to treat the detailed matrix conditioning matrix, you will need to take another class to implement top-of-the-line matrix solvers. However, the techniques we have covered (sparse structures, modular testing, Wilkinson principles, etc.) should give insightful knowledge to use any ready matrix packages.