Project Report

Team: CTZL
Report submitted: Feb.7th, 2021
Member: Zhiyong Liu, Ran Tao, Yichao Zhou

General Structure:

1. Matrix and CSRMatrix

In our entire program procedure, we have two matrix classes, which are Matrix Class and CSRMatrix class. The Matrix is a base class used for dense matrices in general form. The CSRMatrix is a child of Matrix, which is designed to operate on sparse matrices in compressed sparse row form.

2. Solver

We implemented different type of solvers in Solver class. They were programmed in the form of member functions in the Solver class. To use these solvers, all you need to do is to construct a Solver instance and then use the member methods of that instance.

3. Test

The test class is used to verify if the result of each solver is correct and to return the result to the main program. Its core method test result input A, b, and x, and check if Ax == b. It multiplies A and x to generate a check_b. If the error of check_b compared to b is less then tolerance, the test is passed successfully.

4. Main

This is the entry and main program. It has interface which outputs the example A, x and b for Dense and Sparse situation. Also, the procedure of using different solvers and corresponding test results are shown to user. The main program is divided into two parts. Part one invoked different dense solvers and part two invoked different sparse solvers. In part one, we created two diagonally dominant matrices with size 11x11. One is a symmetric matrix used to test the dense Cholesky solver, and the other to test the rest of dense solvers. In part two, we created an 11x11 sparse matrix to test all our sparse solvers. In the end of main program, we also test the matMatMult method we implemented for two sparse matrices. The input and output were show to user.

Jacobi and Gauss-Seidel:
dense_jacobi_solver, dense_gauss_seidel_solver,
sparse_jacobi_solver, sparse_gauss_seidel_solver

The Jacobi and Gauss-Seidel are implemented in quite similar algorithms. The first one updates the initial guess after completing a full iteration, while the latter one updates the guess after solving each row. This makes the Gauss-Seidel method converge faster than Jacobi. Since both methods are iteration methods, both methods in dense and sparse versions all have an optional input of user tolerance. The default value is set to be 1e-6, and user can change it by adding an input (of type double) at the end when calling these methods. Also, there is a set maximum number of iterations to prevent an unnecessarily low number of iterations and the iteration ends when that maximum is reached. However, the weakness of both solvers is that they can only deal with positive definite matrix. For example, if the matrix has diagonal value of 0, solving that row cannot achieve a new guess for the corresponding term and both solvers will report an error.

Multigrid:
dense_multigrid_solver, sparse_multigrid_solver
The multigrid method [1] for dense matrices is roughly complete but can be further modified in various aspects. First of all, the current version only contains one level of coarse grid and can only deal with input matrices with odd number of rows/cols. This is caused by the methods of implementing restriction and interpolation matrices. The current interpolation matrix transfers a vector in coarse grid to fine grid by making even rows in fine grid having the same elements with those in coarse grid and odd rows having one half of the element in previous row plus one half of the element in next row [2]. (Both matrices either in dense or sparse form can be viewed by calling the corresponding multigrid solver in main.cpp). An improved version should deal with odd or even rows/cols so that more levels of coarse grid can be created if the input matrix is quite large in size. Furthermore, current implementation does the pre-smoothing and pro-smoothing processes in a fixed number of Gauss-Seidel cycles which might not perform as expected in some cases while the high frequency errors are large. The big advantage of this solver is that after getting rid of high frequency errors and change the system to a coarse grid, it will need fewer iterations than traditional iteration methods such as Jacobi and Gauss-Seidel as we are working on a smaller system instead.

GMRES:
DenseGMRES
The GMRES implemented in this program is not complete due to time and it can be further completed. This algorithm is based on subspace iteration method. It first finds one proper subspace and find the best approximation in this subspace. If this best approximation satisfies the precision requirement, stop calculation, or it updates the subspace and find the best approximation again until precision requirement is satisfied. GMRES [3] uses the Krylov subspace, $K_m = K_m(A, r_0) = \text{span}\,\{r_0, Ar_0, A^2r_0, \dots, A^{m-1}r_0\}$. Arnoldi algorithm is used to orthogonalized the Krylov subspace to orthonormal basis, $V_m = [v_1, v_2, \dots, v_m]$. The method to find the best approximate solution is to minimize the value the formula: $\| r_m \|_2 = \| b - Ax^m \|_2$. Then, to solve the least square problem, use the QR decomposition based on Givens transformation to get y_m , then iterative solution is obtained by $x_m = x_0 + V_m y_m$.
Current implementation achieves the construction of Krylov subspace and Arnoldi algorithm, but least square problem solve has not been achieved. GMRES has the advantage of fast convergence rate and good stability. Besides, there are other two improved versions, SGMRES (Simpler GMRES) and PGMRES (Preconditional GMRES), which can be further implemented in future if do this assignment again.

Gaussian Elimination and Gaussian Elimination with partial pivoting:
DenseGaussESolve, DenseGaussEPPSolve
These two are simple direct solvers. Gaussian Elimination uses triangulation to convert A to an upper triangle and use back substitution to solve x. Gaussian Elimination with partial

pivoting is the same algorithm with Gaussian Elimination but adds partial pivoting. This is an upgraded version compared with Gaussian Elimination. The advantage is that Gaussian Elimination with partial pivoting could solve matrix with 0 entry in the diagonal. The complexity of Gaussian Elimination and Gaussian Elimination with the partial pivoting algorithm is N^3. According to the performance test, when the size of A enlarges the calculation time will become increase obviously.

LU Factorization:
DenseLUFactorisationSolve, SparseLUFactorisationSolve
LU factorization could efficiently deal with problems where we have multiple RHS vectors. DenseLUFactorisationSolve uses the LU factorization first decomposes matrix A to upper and lower triangular matrices U and L and then do forward and backward substitution to solve x. However, LU factorization decomposes A into L and U only once for all RHS vectors. Then, when using gauss elimination to solve, the Upper triangle and lower triangle matrices are faster to compute compared with general matrices, which efficiently reduces the amount of computation as the number of the RHS vector increases. As for the sparse solver is using the same algorithm with dense LU decomposition. But the difference is that the matrix input is saved as a CSR matrix. The general running method is to do symbolic decomposition and get the sparsity for L and U. and then do the LU decomposition. The symbolic decomposition for general is elimination trees. [4] The advantage of this solver is that the memory space could be saved when calculating a large sparse matrix compared with a dense LU decomposition solver. The disadvantage is the same with dense LU decomposition.

Cholesky decomposition:
DenseCholeskySolve, SparseCholeskySolve
This is a special decomposition for LU decomposition. When we input a symmetric positive definite matrix A. It could be decomposed to L and L^T and then do forward and backward substitution to solve x. In Cholesky decomposition. We can get the entries of L in equations as below[5].

$$L_{kk} = \sqrt{a_{kk} - \sum_{i=0}^{k-1} L_{ki}^2} \qquad L_{ik} = \frac{a_{ik} - \sum_{j=0}^{k-1} L_{ij} L_{kj}}{L_{kk}} \quad (i = k+1, \dots, n)$$

Same as all the direct solver as before. The complexity of the Cholesky decomposition algorithm is N^3. When the size of A enlarges the calculation time will become increase obviously. The disadvantage of this method is that the square root calculation is used in this method to get entry L_{kk}. This will decrease accuracy and increase the amount of calculation. In next steps improvement, LDL decomposition [5] can be used to avoid square root calculation and smart pointer could be used to store the matrix memory in the heap. In addition, sparse Cholesky decomposition is used the sparse matrix as input. And use 3 vectors to record the values row position and column array. Compared with the dense Cholesky decomposition, the upper sparse matrix could be written at first and use to write the lower sparse matrix. Finally, do the same forward and backward substitution and get the result.

Sparse multiplication:

We implement the multiplication of sparse CSR matrices in CSR::matMatMult. It obeys the computation rule [6]:

$$C[i][j] = \sum_{k=0}^{P} A[i][k] * B[k][j]$$

Iterate all values and do operations for each value in left matrix A: Check if exist a value in right matrix whose row index is equal to column index of current value in left matrix B. If exist, generate the result candidate value in result C, $[i][j]$. The row index is equal to row index of value in left matrix A and the column index is equal to column index of value in right matrix B. Value is the product of these two values.
In the process of iteration, the algorithm also checks if current result value's position has generated in previous iteration, because these values are the same position in C. And these values (same row index and same column index) have to be merged to generate the final C.

Performance test:
In our test, with the increase of size in the random positive definite matrix. The calculation time of different methods is shown below.

Solver name	calculation time(101*101)	calculation time(201*201)	calculation time(401*401)
Dense Jacobi	0.069s	0.199s	0.35s
Dense Gauss-Seidel	0.091s	0.29s	0.363s
Dense Gaussian Elimination	0.067s	0.372s	0.315s
Dense Gaussian Elimination PP	0.086s	0.223s	0.282s
Dense LU Factorisation	0.099s	0.179s	0.45s

GitHub repo Link:
https://github.com/acse-2020/group-project-team-ctzl

Breakdown of team CTZL:
Zhiyong Liu: Dense LU solver, dense multigrid solver, CSR matMatmult, code structure.
Ran Tao: Dense Gaussian Elimination (with PP), dense and sparse Cholesky, sparse LU.
Yichao Zhou: Sparse and dense Jacobi, Gauss-Seidel, multigrid solvers.

Reference:

[1] https://en.wikipedia.org/wiki/Multigrid_method
[2] https://math.mit.edu/classes/18.086/2006/am63.pdf
[3] https://en.wikipedia.org/wiki/Generalized_minimal_residual_method
[4] https://vismor.com/documents/network_analysis/matrix_algorithms/S8.SS4.php#LST22
[5] https://en.wikipedia.org/wiki/Cholesky_decomposition
[6] https://blog.csdn.net/meiyubaihe/article/details/29170915