ACSE-5 Assignment: A Linear Solver Library

Team Morpheus: Karim Bacchus, Andika Hakim, Wisit Promrak

February 2, 2020

1 Project Summary

Github repo: https://github.com/acse-2019/acse-5-assignment-morpheus
Please see the `README` for the list of solvers we've implemented and how to use them.

Team contributions:

- Karim Bacchus: Dense GMRES, Sparse Cholesky factorisation.
- Andika Hakim: Dense LU Factorisation, User interface, Tests for comparing linear solvers.
- Wisit Promrak: Dense and Sparse Jacobi, Gauss-Seidel & Conjugate Gradient solvers.

2 Comparison of Solvers

2.1 Comparing run time for each algorithm

The input for software testing is by using matrices with the size ranging from 10×10 to $5,000 \times 5,000$. In general, the larger the input, the more run time required to solve. Since our inputs are well-conditioned, all methods can converge to the right solution. Notably, Conjugate Gradient is the fastest method; followed by Jacobi and Gauss-Seidel solver. LU decomposition is the slowest method as expected since this method computes all the element and decompose it into three matrices. We test our sparse Cholesky later since it requires the CSR Matrix only store its lower triangular values.

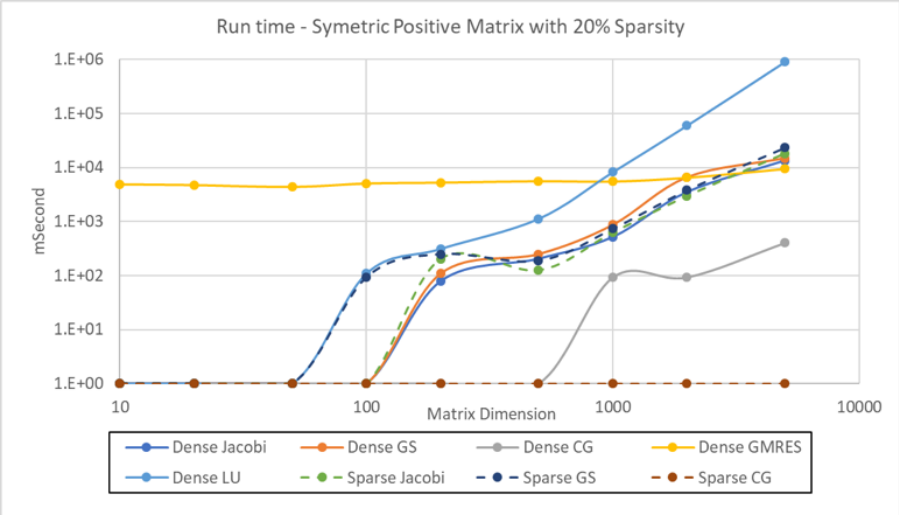

Figure 1: Run time (in ms) with Symmetric Positive Matrices of 20% sparsity

2.2 Comparing dense and sparse solver

By changing input matrix from 20% to 80% sparsity, we observed that sparse solver will outperform dense solver when the size of the matrix is larger than 1000×1000 dimension. The reason for this condition is in the CSR format, the information and the value need three arrays (although it ignores zero value) compared to dense matrix which only needs 1 array. Thus, in this case the trade off for accessing position and value is in size of 1000. As shown in the lower figures, the benefit of using CSR format is highlighted at the difference between dashed line and solid line. The solver in CSR is faster because the nature in CSR format is to avoid zero value and thus the solver will not calculate any operation that ends up another zero value.

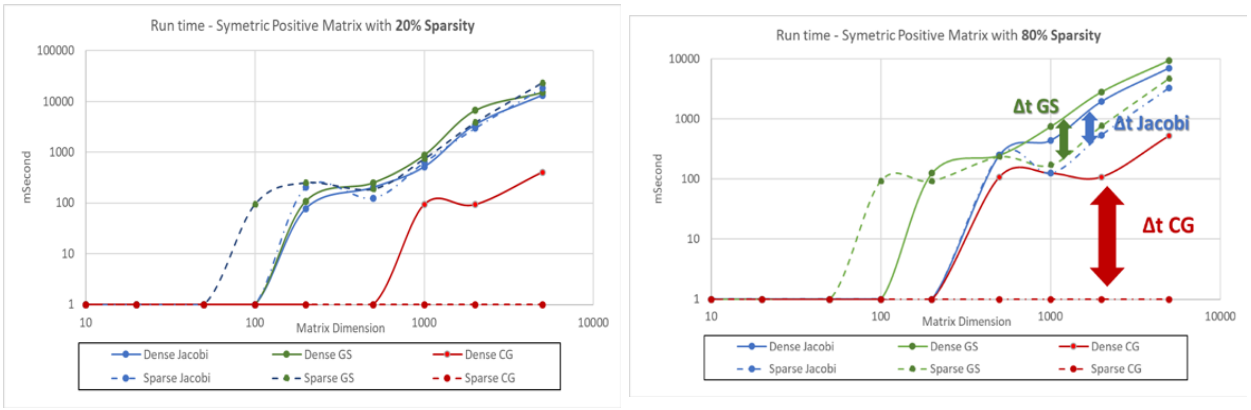

Figure 2: Run time (in ms) with Symmetric Positive Matrices of 20% sparsity

3 Evaluation of Methods

3.1 LU Decomposition

- Solve linear system directly thus it is independent to the input.
- More robust compared to other method even the input was not diagonally dominant, however it's more time consuming method for a large matrix.

3.2 Jacobi

- The simplest algorithm compared to any other iterative method and fasted compared to complex algorithm (Such as GMRES)
- Require strictly diagonal dominant, well-conditioned system (it was tested on random non-diagonal dominance matrix, but the result was not converging)

3.3 Gauss-Seidel

- More stable compared to Jacobi although it still requires diagonally dominant matrix.
- Algorithm is considerably simple compared to GMRES.
- Require strictly diagonal dominant, well-conditioned system (it was tested on random non-diagonal dominance matrix, but the result was not converging, although there were case that GS could give convergence result while Jacobi was failing.

3.4 Conjugate Gradient

- The most powerful method to solve linear problem in this case.
- Algorithm, taken from [3], is considerably simple and easy to implement
- Require input to be positive definite and symmetric matrix.

3.5 Dense GMRES with Preconditioning

The algorithm described by [1]. The maximum number of iterations determines the size of our iteration matrices, thus if a larger `max_iter` than necessary, much more memory will be used than needed. A solution in [1] is described, the *restarted* GMRES method, where we restart the algorithm after a much smaller fixed set iterations to reduce the memory needed, however it can converge much slower.

3.6 Sparse Cholesky Factorisation

We make use of elimination tree [2], which enables us to do the symbolic cholesky factorisation; we can infer the fill in entries (the new non-zero entries in the cholesky factor L) from this, and then we can proceed to calculate the numerical cholesky decomposition on only these non-zero entries.

Size of Matrix	10×10	25×25	50×50	100×100	200×200
Dense LU	0.048	0.245	1.441	9.951	54.389
Sparse Cholesky	0.046	0.053	0.120	0.368	25.196

Table 1: Time in ms to solve $Ax = b$ using Dense LU versus Sparse Cholesky

We note that the advantage of Sparse Cholesky versus the standard dense LU factorisation gap closes as n increases above 200×200. Probably two fold:

- Because the matrix is randomly chosen, and there are more fill-in's as the matrix size increases. We would also in future implement a bandwidth reducing algorithm reverse Cuthill-McKee.
- We did not do any parallelisation, whereas we've made use of blas routines in our dense LU method. We could take advantage of the elimination tree to compute columns that are not dependent on each other in parallel.

On the other hand, if we are using banded matrix to begin with, the speed is vastly superior.

4 Lessons learnt

Developers of this software learn about:

1. Solving linear system depends heavily on the input matrix, thus pre-conditioning and screening of the input matrix is critical in order to select the appropriate solver in a particular problem.
2. Using dynamic memory allocation grant faster performance yet developers has to be aware about memory leaks or memory corruption (This can be minimized by using smart pointers).
3. In case of solving relatively small matrix, dense format is more preferable compared to sparse format because the benefit of using sparse format is negligible for small matrix.
4. Simplifying structure of the software help promote the performance as simple loop ordering can be easily vectorized by the computer.
5. The architecture of sparse-format software needs to be totally re-designed in a new way; using the structure based on dense format can lead to complicated loops and inefficient algorithm.

References

[1] Trefethen, Lloyd N. and Bau, David *Numerical Linear Algebra*. SIAM, pp. 266-270, 1997.

[2] Joseph W. H. Liu, *A Compact Row Storage Scheme for Cholesky Factors Using Elimination Trees*. ACM Trans. on Math. Software 12(2), pp. 127-148, 1986.

[3] Ryan G. McClarren, *Computational Nuclear Engineering and Radiological Science Using Python*. Academic Press, pp. 163-166, 2018

[4] Matthew Piggott, *ACSE-3 Numerical Methods Lecture Notes*. Unpublished, 2019.