ACSE5 Advanced Programming

Machine Breaking

https://github.com/acse-2020/group-project-machine-breaking

C++ Linear Solver Library

Authors:
Iñigo Basterretxea Jacob
Nina Kahr
Jakob Torben

Module coordinators:
Dr. A. Paluszny Rodríguez
Dr. S. Dargaville

February 7, 2021

Imperial College London

Contents

1 Code Design & Structure

The guiding principles behind the structuring of our linear solver library are maximising the variety of linear systems that the user can solve for the same matrix, as well as facilitating the sustainability of the library. With regards to linear solvers, one can distinguish 2 different domains: the matrix with its attributes and the operations that are frequently performed on them; and the algorithms that carry out the methods to solve the linear systems. Thus, we have created a class for dense matrices, and another for their corresponding solvers.

Likewise, when considering sparsely stored matrices, these naturally will have similar attributes to their dense counterparts, yet the operations performed on them will diverge largely from the former. Hence, CSR matrices will have its own separate class which inherits features from Matrix, whilst the two solver classes remain independent from each other.

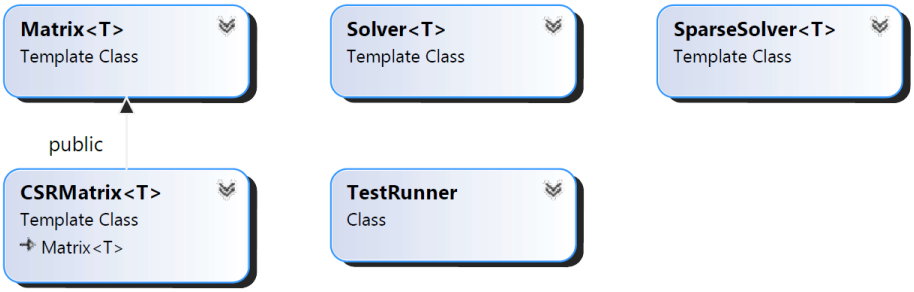

Figure 1: Structure of the classes for the code.

2 Generating $N \times N$ Definite Positive Matrices

Towards the end of a relentless week of coding and studying linear algebra, we found ourselves facing a new challenge: How could we generate symmetric positive definite matrices of arbitrary $N \times N$ dimensions? Further, such a task would become significantly onerous, if we had to accomplish that for sparse matrices. Luckily, our experience with linear solvers enabled us to realise that a matrix is symmetric positive definite only if it has a Cholesky factorisation (Grigori 2015). Therefore, to generate one can simply first create a lower triangular matrix R with positive diagonal entries, obtain its transpose, and multiply them as

$$A = RR^T \tag{1}$$

Thus, we may be sure that the resulting matrix will be positive definite. This will be very useful for establishing the time-performance of our linear solvers for matrices of increasing dimensions and determine whether they correspond to the theoretical orders of complexity.

3 Dense Solver Class

3.1 Jacobi & Gauss-Siedel Methods

The complexity of the Jacobi method is n^2 flops per iteration for dense matrices. Gauss-Seidel iterations converge always while Jacobi iteration don't. We had to make the matrix very diagonally dominant for Jacobi. Jacobi is a lot worse than Gauss. More unstable and slower. The latter can be explained by Jacobi having to store the x from the previous iteration.

3.2 LU Decomposition

The complexity of computing LU is $(2/3)n^3$ flops for dense matrices. The LU decomposition algorithm implemented was inspired by Press et al. (2007).

3.3 Comparison of performance

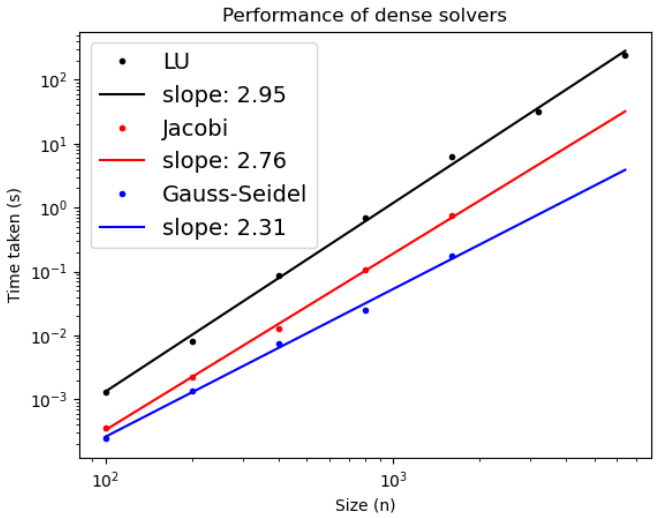

Figure 2: Time-performance of different linear solvers for dense matrices of increasing size.

As expected, iterative methods (e.g. Jacobi, Gauss-Seidel) are superior to direct methods (e.g. LU decomposition) in terms of time-performance. However, they result in only approximate solutions, which may converge only for specifically conditioned matrices (i.e. diagonally dominant). The order of complexity displayed in the figure above.

4 CSR Solver Class

4.1 Jacobi & Gauss-Seidel Methods

4.2 Conjugate Gradient Method

The complexity of the Jacobi method at n^3 flops for dense matrices. This method should be superior (e.g. require fewer iterations) than the Jacobi and Gauss-Seidel methods.

4.3 LU Decomposition

To find the position of the fill-ins for the LU decomposition of a sparse matrix, we used a method outlined in Mittal & Al-Kurdi (2002). We multiply the sparse matrix by itself until the positions of non-zero values no longer changes.

4.4 LU Solver

This method performs forward substitution to solve $Ly = b$ and then backward substitution to obtain the solution from $Ux = y$.

4.5 Cholesky Decomposition

We have now reached the final stop in our search for the holy grail of linear solvers for positive definite systems: the Cholesky decomposition. The complexity of computing R is $(1/3)n^3$flops for dense matrices, which means that it is faster than any of the methods previously covered in this report. An advantage of the Cholesky method over the LU decomposition is that it does not require partial pivotting for accuracy during the decomposition.

This method has 2 distinct parts. The first involves obtaining the infill pattern (e.g. the locations of the non-zero entries) of the resulting sparse lower triangular matrix. We call this the "symbolic matrix". Given our limited

experience with sparsity patterns, instead of making use of sophisticated techniques such as elimination trees and graphs, we resolved to utilise the universal tool known as "finding a pattern". The countless hours spent on this task resulted in the following realisations:

- An entry that is non-zero in a matrix, will also be such in its Cholesky matrix.
- An entry that is 0 in a matrix, will be non-zero if there are any non-zero entries in the same column above the diagonal and in the preceding columns of the same row.

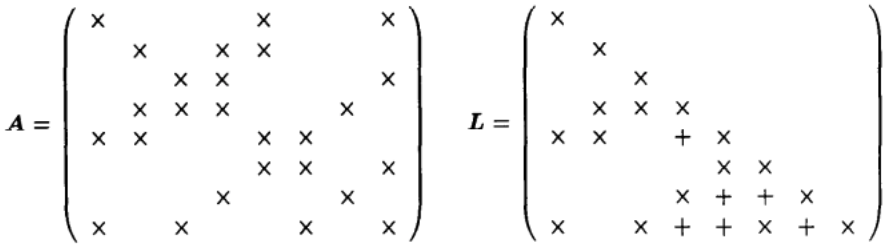

Figure 3: Sparsity of a symmetric matrix and its Cholesky factor (George et al. 1990).

Once this first step has been overcome, the values of the non-zero entries can be calculated by iterating only through the non-zero entries in the symbolic matrix. Now, it is worth noting that in this process, some of the non-zero entries might be cancelled and become 0. A more advanced algorithm would have taken this into account when finding the infill pattern, but this was unfortunately difficult to achieve for a humble taught postgraduate student in the given time-frame. An additional and reasonably attainable improvement could have been made by structuring the code in such a way that it could reuse the "symbolic matrix" for matrices with the same sparsity pattern but different values.

4.6 Cholesky Solver

This method performs forward substitution to solve $Ry = b$ and then backward substitution to obtain the solution from $R^T x = y$.

4.7 Comparison of performance

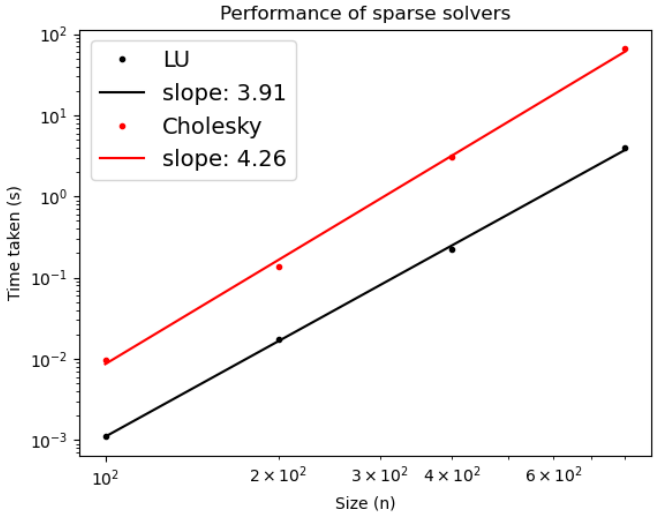

Figure 4: Time-performance of different linear solvers for sparse matrices of increasing size.

Unfortunately, we have been unable to reproduce the theoretical orders of complexity of the 2 direct solvers for sparse linear systems.

A Division of Work

- Nina: LU decomposition, Jacobi, Gauss-Seidel, random CSRMatrix generation, test framework
- Jakob: LU decomposition, Jacobi, random dense generator, solver performance script and plots
- Iñigo: Cholesky decomposition, Jacobi, Gauss-Seidel, Conjugate Gradient, random CSRMatrix generation, report

References

George, A., Heath, M., Joseph, L. & Esmond, N. (1990), Solution of sparse positive definite systems on a hypercube, *in* 'Advances in Parallel Computing', Vol. 1, Elsevier, pp. 129–156.

Grigori, L. (2015), 'Sparse linear solvers'.

Mittal, R. & Al-Kurdi, A. (2002), 'Lu-decomposition and numerical structure for solving large sparse nonsymmetric linear systems', *Computers & Mathematics with Applications* **43**(1-2), 131–155.

Press, W. H., Teukolsky, S. A., Vetterling, W. T. & Flannery, B. P. (2007), *Numerical recipes 3rd edition: The art of scientific computing*, Cambridge university press.