ACSE-5 Coursework: Linear Solvers in C++

Rory Johnston, Alex Campbell & Jorge Garcia

February 2020

Team name: Git Pullers (click here for the repository).

1 Code Design & Structure

The header file Matrix.h contains the prototype member functions and variables of the Matrix class and is used to operate with dense matrices. The file CSRMatrix.h inherits from the Matrix base class, and contains additional prototypes to operate with Compressed Sparse Row (CSR) matrices.
The member functions defined in the header files are found in Matrix.cpp and CSRMatrix.cpp. Instead of having a distinct solver class, we chose to include our solvers within the matrix base and sub classes as this provides easy access for the user and minimises the number of files required. The member functions within our .cpp files have been ordered logically as follows: (i) Constructors, (ii) Matrix operations & overloaded operators, (iii) Linear Algebra operations, (iv) Decomposition methods, (v) Direct & iterative solvers, (vi) destructors.

For full installation and usage instructions please see the README on our github repository (click here).

2 Linear Solvers Implemented

The solvers we implemented are shown in Table 1.

2.1 (a) Direct methods

LU decomposition builds on Gaussian Elimination, where we form upper triangular matrices to solve for each value individually. This is expanded into factorizing the input matrices into both upper and lower triangular matrices, as well as a permutation matrix when conducting partial pivoting (A = P L U). This can be rearranged to form the inverse matrix to solve the system for any generic RHS vector using P (L (U$\mathbf{x}$)) = $\mathbf{b}$. This information was obtained from ACSE Lecture 3 Linear Solvers Lecture. Now Cholesky decompositon: the Hermitian matrix A can be expressed as the product of a lower diagonal matrix L and its transpose L^T. This is used to solve A$\mathbf{x}$ = $\mathbf{b}$ by: (1) Expressing the system as LL$\mathbf{t}$$\mathbf{x}$= $\mathbf{b}$; (2) Solving the system L$\mathbf{y}$ = $\mathbf{b}$ using forward substitution; (3) Solving the system L$\mathbf{t}$$\mathbf{x}$ = $\mathbf{b}$ by using back substitution. For the method to converge, the matrix must be Hermitian. Information obtained from: (click here).

2.2 (b) Iterative methods

Both Conjugate Gradient, Gauss-Seidel and Jacobi are iterative methods. Gauss-Seidel uses previously computed results as soon as they become available, whereas Jacobi does so after a whole iteration. For convergence,

Solvers List		
Solver	Dense	CSR
LU decomposition	Yes	No
Cholesky Decomposition	Yes	No
Gaussian Elimination	Yes	No
Gauss Seidel	Yes	Yes
Jacobi	Yes	No
Conjugate Gradient	Yes	Yes

Table 1: Indicates the solvers implemented successfully for both our dense matrix and CSR matrix classes.

matrix A must be diagonally dominant and have no zeros on its main diagonal. These methods will iterate until either the tolerance (2-norm) or maximum number of iterations is reached.

3 Testing

3.1 (a) Code Verification

To test our solvers we used the method of manufactured solutions. We (1) generated a known linear system $A\mathbf{x} = \mathbf{b}$; (2) Input A and $\mathbf{b}$ into each solver; and (3) Output our solver's estimate, $\mathbf{x}_{est}$, and calculated the RMS error between our $\mathbf{x}_{est}$ and $\mathbf{x}$ to check they match sufficiently closely. We provide options to produce both a fixed matrix system and a pseudo-random system, where a random number generator fills the matrix A and vector $\mathbf{x}$ within a specified range of numbers. It was ensured the matrix produced was always SPD. This was implemented for both sparse and dense systems.

3.2 (b) Comparison to BLAS/LAPACK

To assess performance, we compared the runtime of our matrix-matrix (DGEMM) and matrix-vector (DGEMV) multiplication functions in our Matrix class versus the BLAS and LAPACK packages, the results are shown in Figure 1. We see that our matrix-vector function performs comparably to BLAS but our matrix-matrix function performs significantly worse at larger array sizes. See note below on nested-loop optimisation.

3.3 (c) Time Complexity

We generated a symmetric positive definite (SPD) penta-diagonal matrix of increasing size and measured the time taken for each solver to converge. We ensured that we used the same tolerance for each iterative method, 10^{-10}. We then fitted polynomials to each dataset, with the results shown in Figure 2.
The ideal situation is a linear relationship $O(n)$ between computation time and matrix size. We see that both our Conjugate Gradient and Gauss-Seidel solvers, when implemented with CSR matrices, achieve this. LU-decomposition with dense matrices also appears to achieve a linear relationship but is clearly orders of magnitude slower than the other two methods. Other dense solver methods give $O(n^2)$ - $O(n^3)$. To explain these observations we take the conjugate gradient method as an example. The most expensive operation is a matrix-vector multiplication of order $O(n^2)$. In CSR format this operation reduces to $O(n)$, now it is the number of non-zero elements that is relevant. We confirm that this is the case from figure 2 by examining the order of the polynomials fitted. Overall, for larger matrices Gauss-Seidel appears to be the most computationally-efficient of our solvers.

4 Issues, Strengths, Weaknesses & Future Improvements

We encountered numerous issues during our project. For example, forgetting to delete memory from new constructors such as the SPD constructor. This memory leak issue only appeared later when stress-testing the solvers for far larger array sizes. Secondly, integration of our collective work proved a problem as we all specified slightly different inputs into our code. In retrospect, we should have agreed guidelines as to the inputs/outputs from functions.

Overall, our code provides a good range of solvers for the user, has both sparse and dense options, is well-tested, documented and robust. However, given more time there are several things we would have changed. Firstly, we would have implemented more sparse options - we came close to implementing sparse Cholesky. Other potential methods include Chebyshev, which would provide an efficient option if using a distributed memory architecture.

We also initially used raw pointers. By using smart pointers from the start we would have avoided many of the memory leak issues encountered. Nested loop optimization (tiling) was attempted by deconstructing matrices into smaller grids to perform matrix-matrix multiplication in parallel and make use of cache hierarchies. This achieved millisecond-level speedup for matrices of dimension >1500, but would only work under certain conditions, and more advanced tiling methods optimized machine-specific cache sizes. For this reason, it would have been beneficial to have devoted more time to implementing cache-oblivious algorithms.

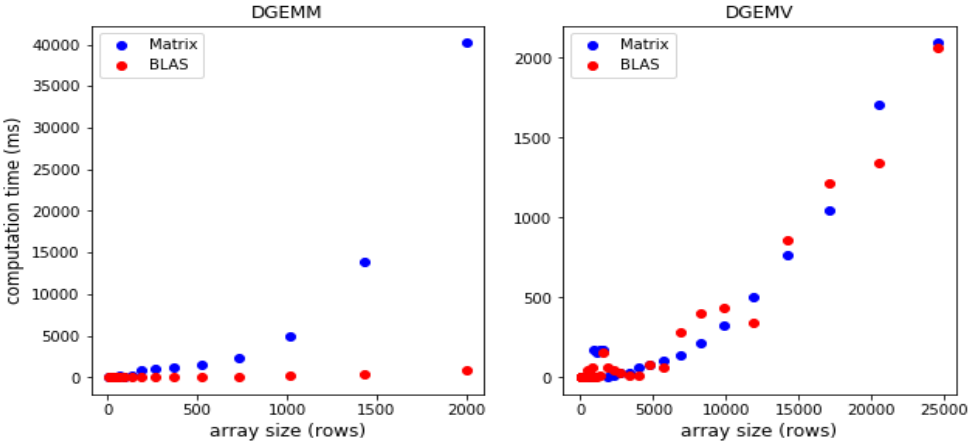

Figure 1: Plots of computational time vs array size (rows) for our matrix-vector (DGEMV) and matrix-matrix (DGEMM) functions vs the BLAS equivalent. Plotted in Python 3.7.

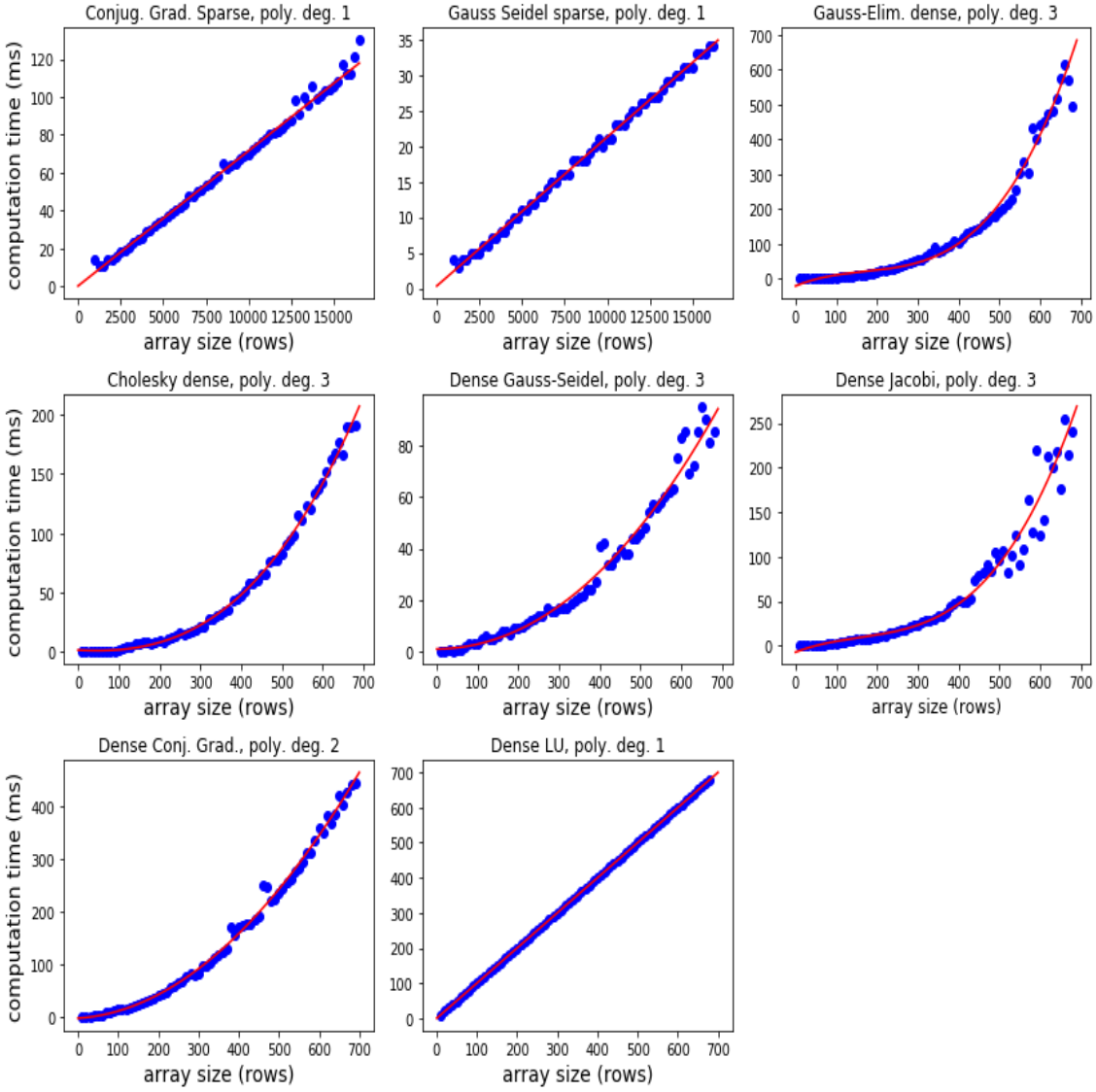

Figure 2: Plots of computational time vs array size (rows) for each of our solvers. Note the sparse solvers were tested up to significantly larger array sizes. Plotted in Python 3.7.