Team Void report

Code structure

The library contains two classes, for a dense and a sparse matrix and their associated solvers. The dense **Matrix** class is the base class and the **CSRMatrix** class is a child class. Matrix values are stored in a flattened array with row major ordering.

The CSRMatrix is used to store sparse matrices in a CSR format which reduces the memory usage for low density matrices. The format only stores the non-zero values of the matrix, their associated column indices and the number of values in every row.
In the case of low density matrices, this can eliminate redundant operations and lead to significant improvements in performance.
A **dense2sparse** method can be used to transform a matrix which is stored in a dense row major format to a dense format.
The CSRmatrix class includes basic matrix operations for sparsely stored matrices and four solvers.

Implemented methods:

We have implemented six (almost seven!) dense matrix solver methods within this library.
The solvers require the linear system to have an exact solution and the matrix needs to be symmetric positive-definite with no zeroes on the main diagonal.
If these requirements are satisfied, the following solvers can be used to solve a linear system with no restriction on the size of the system:

jacobi_element: Iterative algorithm using an element wise operations
jacobi_matrix: Iterative algorithm using an matrix operations
gauss_seidel: Iterative solver which immediately uses previously calculated values
LUSolve: Direct solver using LU decomposition and back/forward substitution
conjugate_gradient: Iterative method which is finding the direction of steepest descent
Cholesky_solve: Direct solver using LL^T decomposition and back/forward substitution
multigrid: Iterative solver using jacobi method on a coarser grid (not fully working)

The 4 solver methods for sparse matrices which can be found in the CSRMatrix class are **Gauss_seidel_sparse**, **jacobi_solver_sparse**, **CholeskySolve** and **conjugate_gradient**.

Testing

We have tested the library using an extensive test suite available in the ***main_tests_solvers.cpp*** and ***main_tests_components.cpp*** files.
The former tests all available solvers on a random symmetric and very diagonally dominant matrix A (using SPD method) and random vector b of the same size.
This test is done on linear systems of different sizes, reaching up to 5000 rows and columns.

The solver error is calculated by computing Ax=b through matvec multiplication and comparing our “solved” b against the input.
The error is then compared against the tolerance, and a message is printed to the screen detailing the outcome for every method.

We have also included a few tests where a known linear system is solved and the outputs of the solvers are checked against the known solution.

The main_tests_componets.cpp file checks many of the Matrix and Vector methods of both classes with known inputs and outputs.

Precision

We can achieve varying degrees of precision with different methods.
Iterative methods, LU and Cholesky (dense and sparse), are very much in line with what we would expect from machine precision.
On 32 bit computers, double precision is approximately 10^{-16}, and our RMS increases linearly with the number of rows of the matrix. On the range of Ns tested (10-5000), it goes from $2 * 10^{-16}$ to $5 * 10^{-13}$.Unfortunately we cannot seem to achieve the same level of precision with iterative methods, and a precision higher than 10^{-12} *cannot be guaranteed*.
The time it takes for iterative methods to return a solution seems to depend linearly on the tolerance specified. For Jacobi To achieve a precision of 10^{-6} it takes 2 times as long than it takes for it to achieve a precision of 10^{-2}. For the conjugate gradient method the factor is closer to 4, with factors for other methods falling somewhere in between.

When precision is important, we advise using non-iterative methods.

Runtime vs n

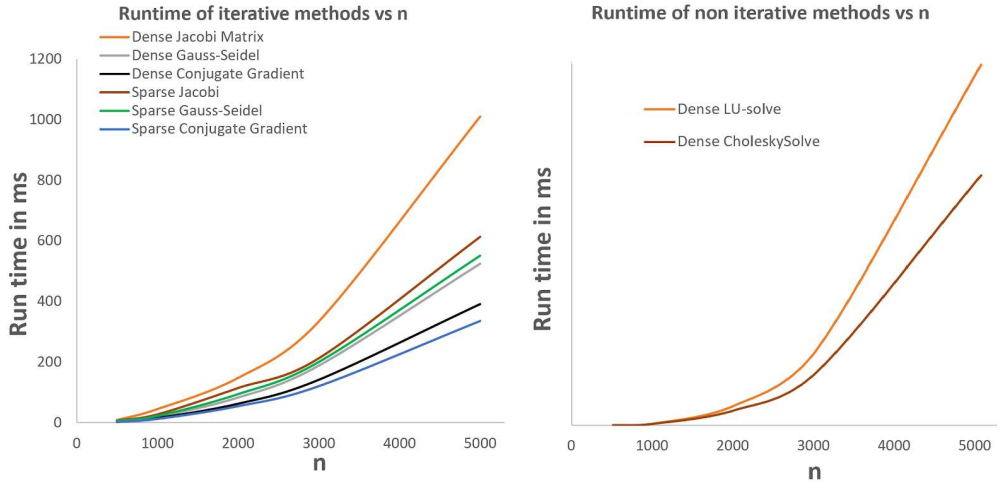

Figure 1, Runtime for methods for different sizes of linear systems

Probably due to our terrible memory management, the sparse Cholesky solver breaks for some nxn matrices with n>500, and so is not shown.
Non iterative solvers, both dense and sparse, clearly run in O(n^3) time while iterative solvers seem to run in O(n^2) time.

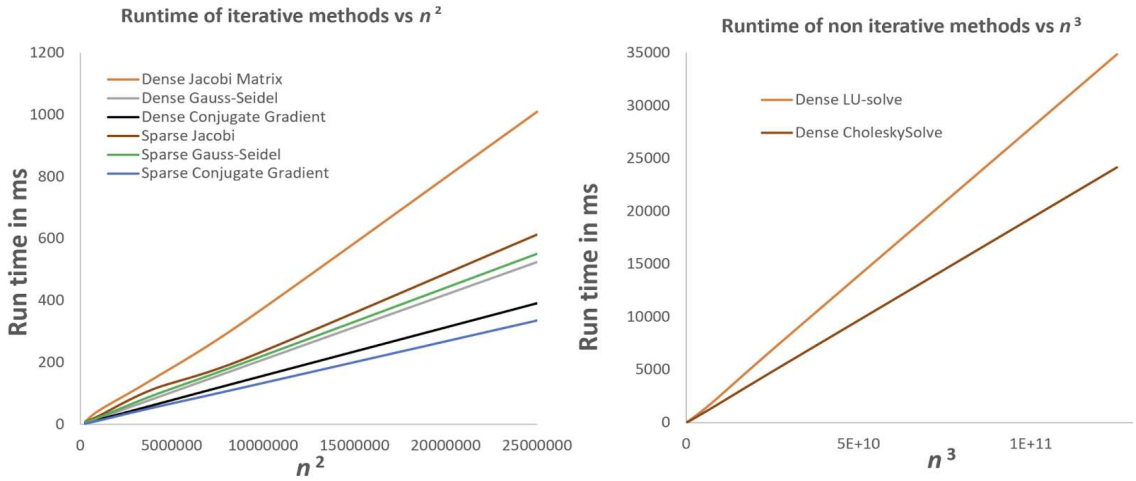

Figure 2, Runtime for methods against squared and cubed n

As this should not be possible (Jacobi should be O(n^3)) we guess that the compiler optimises our code so that several operations are run in parallel. Cholesky is slightly faster than LU, as expected.
We have run more tests on other methods, and we can confirm that our matMatMult runs in O(n^3) time. Strassen runs in O($n^{2.7}$) time, but we didn't have time to implement it.

Time vs density:

The behavior of the solvers with increasing matrix density is analysed below.
We used a 1000 x 1000 matrix which was filled with varying amounts of non-zero entries. The iterative methods were run until they reached a tolerance of 10^{-5} and -O3 optimization was used for every method. The matrix A was filled with 5000 on the diagonal and a varying amount of 1s on other spots, depending on density.
The time to run for each method was recorded as the density increased from 0.01 to 1.

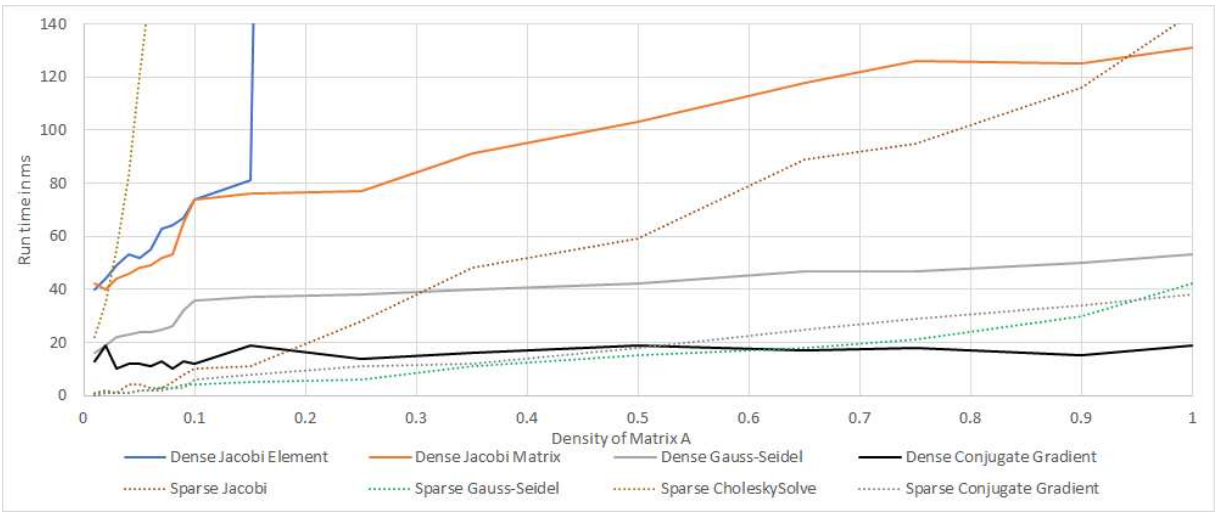

Figure 3, Timing for every method for varying density, not including LUSolve and dense Cholesky due to long runtime (constantly 600 ms and 350 ms respectively)

The two dense Jacobi solvers show very different behavior, the Jacobi matrix solver shows a steady increase in runtime. This behaviour is to be expected since more values on the non-diagonal mean that the diagonal values will be less damping which leads to slower

convergence. This might also be the reason why the Gauss-seidel solver shows an increase in runtime. The Jacobi element solver shows a different behaviour, and for densities of more than 0.25 %, the runtime is more than 2000ms because the convergence is very slow and requires more than 500 iterations.
All dense direct solvers (LU and Cholesky) have a constant runtime of about 350ms and 600ms respectively. They are doing the same amount of operations regardless of the density of the matrix and are not affected by changes to the eigenvalues of A, therefore both have a constant runtime.

The runtime of sparse methods is positively correlated with density matrix, due to the increase in operations needed. The sparse Jacobi and sparse Gauss-Seidel solvers are faster than their dense counterparts for most matrix densities. This is due to the fewer operations needed per iteration. As the matrix density approaches 1, the sparse Jacobi takes longer than the dense Jacobi solver which may indicate a deficient implementation for optimization.
The sparse Cholesky solver is very fast for sparse matrices but its runtime increases in order of $O(density^3)$ due more operations being required. For dense matrices it is significantly slower than the dense implementation which may indicate that the implementation is not optimal for the compiler optimization.

Possible improvements

One aspect which may prove valuable for a linear solver library, and which could be implemented given more time, is a method functionality to assist the user in choosing the appropriate linear solver for their specified inputs/system. With regards to the solvers demonstrated in the library, there are various advantages and limitations to each. For example although Gauss-seidel is advantageous for large/sparse systems and with a convergence of $O(n^3)$ - Cholesky also being $O(n^3)$ only requires half the storage.

Throughout the project, we were riddled with unintuitive memory leaks - proving spurious results in both timing and failed tests. This could be massively prevented and with further implementation of dynamic allocation of objects smart pointers. Specifically, one use of smart pointers enables nullifying possibility for memory leaks across all solver calls could be to implement when creating a pointer to the CSR or Matrix object.
As a diagnostic resource we could have made greater use of valgrind for memory leak detection and debugging.
Possibility for faster solvers may have been achieved through reducing redundant loops whilst maintaining stricter use of abstraction. Achieving optimised methods within the class and calling when required for each solver would be beneficial. Whilst this was partially achieved in some places (daxpy), there's room for improvement. Implementing calls to BLAS within matVecmult and other abstracted methods would provide a significant speedup.
Creating a Vector class would have made it easier to make our methods robust and provided a class for the vector methods, even though dynamically allocated memory might have ran slower.

At the start of this project, the team should have specified all inputs and outputs for the solver functions and more care will be taken the next time to ensure a uniform code base.
Lu pivoting could have also been implemented with more time.