# C++ group project report-30/01/2022

By Yifan Zhang, Donghu guo, Sanjeet Chhokar

## Project Aims

The objective of this assignment was to produce an efficient C++ library for solving linear systems of the form Ax=b where A is a matrix and x and b are vectors respectively. We use multiple algorithms for dense and sparse matrices to achieve this objective.

## Software Design Choices

Our linear solver library is segmented into different components, with a Matrix class, CSR matrix class, a solver class, and an interface class for user interaction with our code.

The matrix class is responsible for both defining matrices and handling common operations that they involve such as matrix vector multiplication and changing the value of a particular matrix element.

The CSR matrix class is responsible for handling sparse matrices within our linear solver library. It has methods that include initialising CSR matrices, doing sparse matrix vector multiplication and being able to set additional values within the sparse matrix that do not rely on converting the matrix to a dense format first.

The solver class contains the functions corresponding to our implemented linear solvers, namely: Gaussian elimination [1], Gauss-Jordan elimination, LU decomposition (with and without partial pivoting), the Jacobi method and the Gauss-Seidel method.

The interface to our linear solver library has been designed to be sequential, with functions that called that depend on the choice of user input and therefore the type of linear solver the user requires to be implemented.

The choice to separate our code in this way was taken after the consideration of several factors. Firstly, code readability and the ability to debug our code was helped by the separation of the code into different classes (as opposed to having one class with all of our functions).

## Using the Software

The software, once downloaded can be run via the command line via a sequence of commands such as

```
g++-11 -o output Matrix.cpp Solver.cpp Interface.cpp main.cpp

./output
```

This code generates an executable which is then run and allows the user to interact with our code.

Once done, an interface is shown that allows for the selection of a linear system to solve and the method used to solve it. The user is given multiple choices when it comes to selecting what linear system they would like to solve:

1. Randomly generated data
2. Data inputted via a text file
3. Manually inputted data via the terminal

This allows for different use cases to be met; for example, the text file feature would be useful for large matrix systems arising out of research problems and this method therefore provides an easy way for real problems to be solved easily.

The user is expected to ensure that the inputs given to the jacobi method and the gauss seidel method are diagonally dominant to ensure convergence.

An example of use of the interface in solving a linear system is shown below using randomly generated data.

```
 ------------------------------------------------
|           Fill up the matrix data             |:
 ------------------------------------------------
| 1: Randomly generated data                    |
| 2: Enter the data                             |
| 3: Text file                                  |
| b: Back                                       |
| x: Exit                                       |
 ------------------------------------------------
>> 1

 ---------------------------------------------------------------------------
| Generate randomly generated data between left_boundary and right_boundary |
 ---------------------------------------------------------------------------
  Please enter the left_boundary first
>> 1
  Then please enter the right_boundary first
>> 10
```

The left and right boundaries specified by the user above correspond to the minimum and maximum values present within the randomly generated matrix.

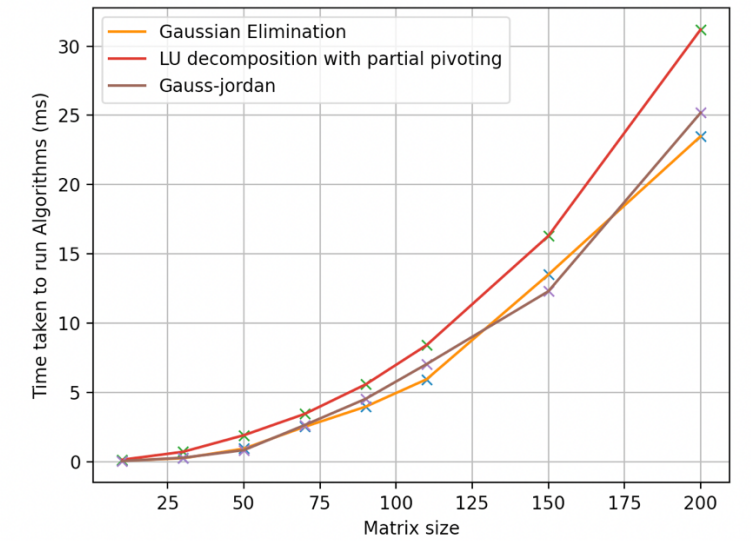
```
------------------------------------------------
Print randomly generated matrix APrinting matrix
4.23854 8.21649 4.53575 8.30442 6.38221 9.87547 4.97325 8.42636 3.77523 6.21036
1.53801 7.36884 5.14708 4.98776 9.25868 6.55544 5.30136 5.97032 8.12135 7.51983
4.73209 5.29386 5.95886 4.58438 6.63299 3.62006 8.33806 4.71802 2.72293 5.25386
8.66736 4.37958 2.6381 1.48798 3.43905 8.1607 2.84874 4.80013 5.77943 3.84355
2.62752 3.803 4.97124 1.69012 7.82718 4.48698 7.70044 7.374 1.74392 3.12373
9.51425 8.97859 6.18982 7.28412 3.17974 5.93239 9.67646 8.3183 5.72127 7.44547
5.93928 8.48373 5.1248 8.52132 6.88384 7.78284 6.1393 4.22769 5.82644 1.91739
2.51503 3.15705 2.489 6.57271 7.58033 4.64341 8.75924 9.53645 4.11137 3.73632
9.37619 1.60742 3.94116 5.01227 7.23587 2.31739 2.39895 5.13152 4.44038 7.51116
3.01721 1.1676 9.78102 1.65972 9.86329 7.30049 8.36216 4.89365 2.51779 4.50135
------------------------------------------------

------------------------------------------------
Print randomly generated vector b:
[4.23854,8.21649,4.53575,8.30442,6.38221,9.87547,4.97325,8.42636,3.77523,6.21036]
------------------------------------------------

 ------------------------------------------------
|  Select the dense matrix solver               |:
 ------------------------------------------------
| 1: Gaussian Elimination                       |
| 2: Gaussian Jordan                            |
| 3: Gaussian Seidel                            |
| 4: LU factorisation                           |
| 5: LU factorisation with partial pivoting     |
| 6: Jacobi                                     |
| b: Back                                       |
| x: Exit                                       |
 ------------------------------------------------
>> 1
Time spent: 0.203

------------------------------------------------
The output solved by Gaussian Elimination:
[ 0.0175292,-0.180511,0.151638,-0.464553,-0.331574,0.0124639,0.432607,0.664858,1.14937,-0.20649]
------------------------------------------------
```

## Code Performance

Many of the algorithms used by our linear solvers rely on the invariance of the solutions to a set on linear equations (in matrix form) when subject to operations. For example, adding or subtracting multiples of different rows whilst changing the b vector correspondingly leaves the solution to the equations unchanged. For example, Gaussian elimination applies these

operations to transfer the augmented matrix system into one that is in echelon form (eg in upper triangular form). Due to the form of each of these algorithmic operations, each of the algorithms we have used also has a theoretically determined time complexity.

Gaussian elimination, gauss-Jordan elimination and the LU decomposition [2] method are expected to scale as $O(n^3)$[3]. The performance of our code based on the time taken for the algorithms to run was computed for various sized matrices in order to compare the theoretical result to our codes performance.




The graph shows that our algorithmic implementations follow the predicted scaling relationship, thus adding validity to our coding methods and showing that we have not introduced code that is more computationally expensive than necessary.

### Sparse solver vs dense solver time comparison

The jacobi method was performed via the dense solver and the sparse solver on the 4x4 matrix shown below with the b vector chosen as [1,2,3,4].

```
10 0 0 0
0 15 5 1
0 0 25 6
0 0 0 30
```

Time taken for Jacobi solver:

Dense solver: 0.067ms
Sparse Solver: 0.032ms

Storing and manipulating the linear system in a sparse format provides significant performance gains as compared to the dense version. The above analysis was replicated on a larger 7x7 matrix system and the sparse solver was shown to be faster again than its dense counterpart (as only non-zero values are used in the sparse algorithms the time taken for the operations is reduced). Similar performance gains are present for the other sparse matrix solvers implemented.

## Strengths of our code

We use pointers to arrays in our solver class functions as opposed to passing in vectors and this makes use of the speed of C++ arrays and leads to improved code optimisation.

The development of the CSR matrix class methods that allow for sparse matrices to be handled and manipulated for use in our algorithms without the need for converting back into a dense format.

Our code makes us of the concept of row major ordering that respects the cache hierarchy therefore leading to speed improvements for our code.

Our code can consistently provide performance gains for sparse matrices as the computational cost of working with these matrices is reduced as a result of our CSR matrix class methods.

## Areas for Improvement

We could have tried to implement methods for matrices with real physical significance for example, those resulting out the discretisation of partial differential equations (e.g., looking at the discretisation of Poisson's equation for steady state diffusion).

We would improve the consistency of the structure of our code and standardise our coding style.

Making use of more advanced techniques that aid memory management such as smart pointers.

We could improve the efficiency of our code by using BLAS/LAPAC routines for the level of complexity below which we are working as these are highly optimised.

## References:

[1] Lecture_5_Linear_solvers.ipynb by Steven Dargaville

[2] https://courses.engr.illinois.edu/cs357/sp2020/assets/lectures/Lecture13-March3-annotated.pdf

[3]] https://www.geeksforgeeks.org/gaussian-elimination/

## Division of Groupwork

Yifan Zhang：Developing the user interface for our code and designing multiple dense solvers.

Donghu guo：Writing the Jacobi method, developing the CSR matrix class and implementing sparse solvers with the class.

Sanjeet Chhokar：Writing the Gauss Seidel algorithm code and the report.

**Repo link**: https://github.com/acse-2020/group-assignment-lsg.git