LINEAR SOLVERS FOR DENSE AND SPARSE MATRICES

Yuna Nakamura, Ellya Kanimova, Niranjana Sundararajan

January 30, 2022

I. Introduction

Linear Systems of the type $A * X = B$ that arise from the discretisation of a number of partial differential equations are a very common type of problem that requires robustly designed solvers that account for optimisations in both time and memory management. This matrix library can be used for such implementations. Depending upon specific requirements and the nature of the linear system, different solvers each with their own set of properties are available to the user.
The solvers implemented in the project include both direct and iterative solvers. The direct solvers implemented include Gauss elimination with partial pivoting, LU decomposition and the Cholesky Method. The iterative methods include the Jacobi Method, the Gauss-Siedel Method and the Conjugate Gradient method.
Of the methods listed above, all are implemented for the dense matrix input and the conjugate gradient and Jacobi methods are implemented for the sparse matrix

II. Code Structure and Design Decisions

The figure below shows all the base classes in our library(in light blue) and the inherited classes (darker blue) with the dotted line indicating inheritance.

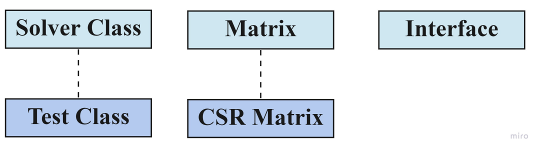

Figure 1: *All Classes and Hierarchies*

In the current versions of the Matrix and CSR Matrix class, the implemented library has build upon the existing code with added functionalities.
While our code builds on the Matrix and CSR Matrix classes, the library implements three new classes - the Solver, the Interface and the Test classes with extensive functionalities needed to solve our linear systems. These functionalities are briefly illustrated in the charts on the right:

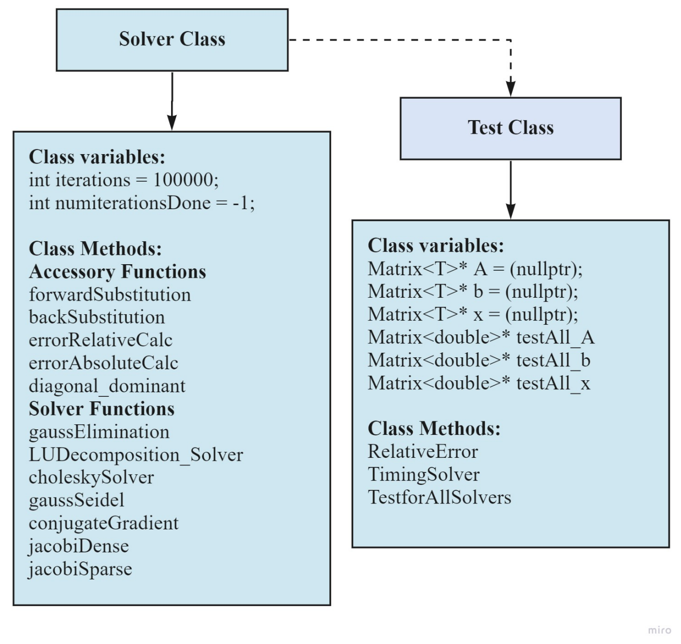

Figure 2: *Classes Implemented to Extend Backend Functionalities*

Further, an interface class is implemented as a frontend at the terminal to provide additional ease of testing and validation.

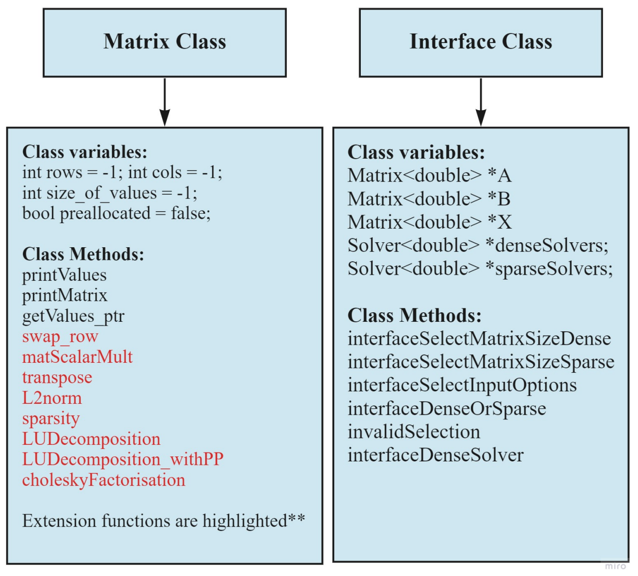

Figure 3: *Extended Functionalities of Matrix Class and Classes Implemented to Add Frontend Tests*

The library is implemented using various Object Oriented Programming design principles for efficient and sustainable code structure and maintainability. The following are few principles and their implementations in our library :

- **Inheritance**: The Test class inherits from the solver class. This was a design decision to further simplify the usage of solver functions for both validation and performance testing of the solvers implemented.
- **Templates**: All classes, except the interface class are templatised to allow for matrix inputs to be of preferred data types. This allows for both better memory management, flexibility for the user and performance.
- **Smart Pointers**: Smart pointers are used in the Matrix functions and classes within the library to ensure memory leakage is managed appropriately. However further considerations were made to reset the shared pointers and release the memory during the run-time as soon as the container under consideration was not required for further computations.

  Other factors that were considered during the design of this library that helped make the solvers robust include:
- **Row Major Ordering**: Since C++ stores container data in row major format, all the for loops in the solver code are written in row ordering in order to optimize cache memory usage during computations in the for loops. This drastically optimises the performance of our solver in terms of speed of execution.
- **Robust Testing**: This matrix library tests the solvers for randomly generated linear systems of different input sizes. This allows for validation tests that compute both absolute and relative errors.
- **User Interface**: Implemented for this solver is a terminal based GUI that allows the user to test whether all the solvers are working as expected. The user can test the solvers for a range of input sizes that will be read-in through text files. The output is displayed on the terminal for quick validation and analysis.

III. A Brief Discussion on Implemented Solvers

Each of the solvers implemented have specific properties and should only be used under specific circumstances to obtain results that are reliable. Of the solvers implemented, three are of the direct type (Gauss, Cholesky and LU Decomposition) and the other two (Jacobi and Gauss-Siedel) are of the iterative type.
The properties of the input matrix to the solver greatly influences the selection of the solver. Without considering pre-conditioning, the Jacobi solver assumes a diagonally dominant matrix, Cholesky requires the matrix to be symmetric-positive definite and the Gauss-Siedel needs the matrix to be either symmetric and positive-definite or strictly diagonally dominant. Thus matrix inputs is one of the most important consideration. Thus, in terms of robustness of the solvers for different kinds of input Gauss Siedel is the most appropriate choice.
Further, for sparse input data, the user can pick a a solver that is uses the compressed sparse row format to compute solutions for the defined linear system. Within this library, the Jacobi Solver is implemented that can be used to solve diagonally dominant sparse matrices.

IV. Performance Analysis of Solvers

Based on consideration of time and spatial complexities of the algorithms, the following sections explore the results of multiple analyses that were performed using each of our solvers. The table below illustrates the average time taken by each of our implemented solvers for increasing matrix sizes.

Table 1: *Absolute Error in Implemented Solvers*

Solver	50 x 50	100 x100	200 x 200	400 x400	600 x800	800 x800
Gauss- Elimination	8.43E-12	7.57E-11	6.13E-10	4.83E-09	9.46E-08	3.29E-11
LU Decomposition	1.36E-11	1.10E-10	8.83E-10	7.39E-09	1.53E-07	4.55E-11
Cholesky	1.54E-11	1.24E-10	8.72E-10	6.78E-09	1.76E-07	4.73E-11
Conjugate Gradient	2.43E-06	8.21E-06	7.54E-06	1.46E-05	1.84E-05	2.48E-05

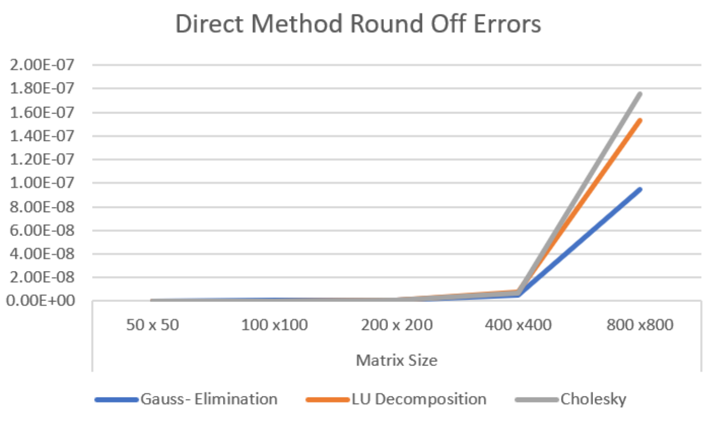

Figure 4: *Arithmetic Floating Point Error*

While we would expect direct methods(discussed below) such as Gauss, LU and Cholesky to generate an exact solution, we can see through our analyses that, due to nature of computer based computations that cause issues when floating point arithmetics are under consideration, the absolute errors in these methods, that we would theoretically expect to be zero, are not observed to be so. The errors tend to grow with increase in matrix size as more computations are required.

i. Dense Vs Sparse Solvers

While the sparsity of matrices does not affect the performance of the solvers, when a sparse matrix is stored and iterated upon using the compressed sparse row format implemented in the CSR class, we can see the efficiency (both in terms of speed but especially memory used) improve drastically. This was especially notable when we analysed and compared the a diagonally dominant sparse matrix solved using the Jacobi dense versus the Jacobi Sparse method as shown in the table below.

Table 2: *Solving a Sparse Diagonally Dominant Matrix with a Dense Vs Sparse Jacobi Solver*

	Jacobi Dense		Jacobi Sparse	
Matrix Size	100x100	200x200	100x100	200x200
Time	0.004	0.02	0.002	0.007
Absolute Error	3.20747×10^{-9}	1.67387×10^{-8}	5.9555×10^{-7}	6.54×10^{-7}
Relative Error	1.5059×10^{-9}	2.9288×10^{-9}	1.3305×10^{-7}	9.0728×10^{-8}
Iterations	12	15	15	13

From the table above, we observe that the Jacobi Sparse solver is indeed faster for both matrix sizes tested, however it can be noted that it is considerably faster for the 200x200 matrix. This aligns with our theoretical predication that for a sparse matrix as the matrix size increases, a sparse solve will be much more efficient in time and memory. One additional point to be noted is the fact that for a sparse matrix, we would need to compute the symbolic matmatmult to find the number of non-zero elements. This is an additional function that needs to be accounted for in sparse calculations.

ii. Time and Matrix Size Analysis

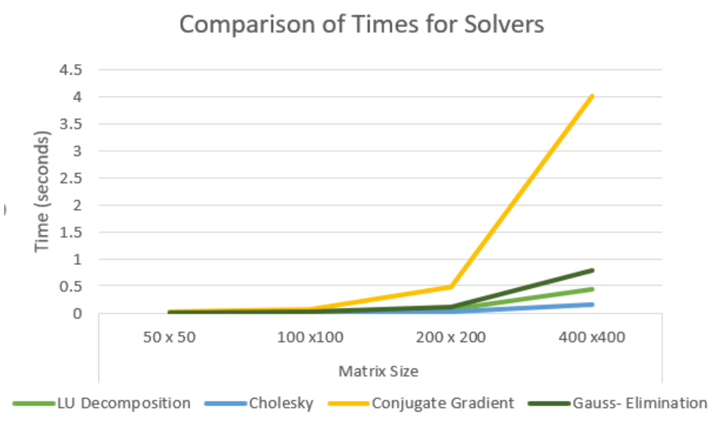

Figure 5: *Times for Solvers*

From the graph above we can see that from amongst the direct solvers, Gauss elimination takes greater time to generate out compared to Cholesky and LU decomposition while conjugate gradient, an iterative solver, converges the slowest. Since iterative methods such as conjugate gradient tend to be faster for larger matrix sizes, we can speculate based on the data generated that a) the matrices we generated were not large enough to see the payoffs of an iterative method. b) Our compiler in Visual Studio optimized very well the direct methods but found the iterative method more challenging to optimize during runtime or c) conjugate gradient was sensitive to the generated inputs.

iii. Memory and CPU Considerations

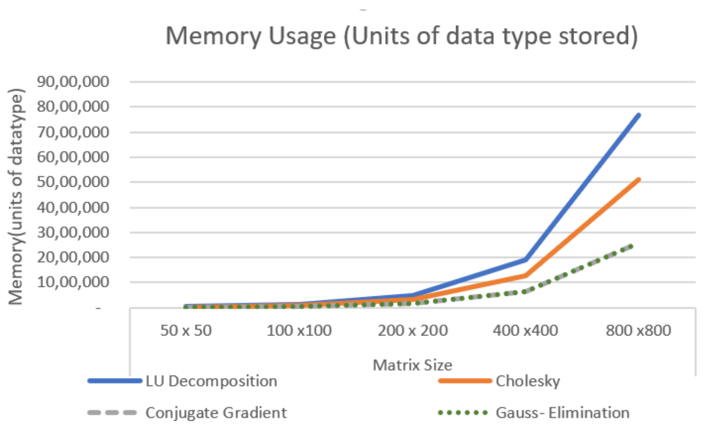

Figure 6: *Caption*

Table 3: *Average Memory(in units of memory) needed to Implement Solvers*

Matrix Size	50 x 50	100 x100	200 x 200	400 x400	800 x800
Gauss- Elimination	2,600	10,200	40,400	1,60,800	6,41,600
LU Decomposition	7,700	30,400	1,20,800	4,81,600	19,23,200
Cholesky	5,150	20,300	80,600	3,21,200	12,82,400
Conjugate Gradient	2,800	10,600	41,200	1,62,400	6,44,800

One of major issues upon which we see a clear contrast in our analyses is the theoretical memory usage of direct and iterative methods. In the analyses above, we observe that for the same randomly generated linear system, direct and iterative methods behave differently - especially across a range of matrix sizes.For the matrix sizes analysed, we observe that direct methods are considerably faster. Since unlike iterative methods they arrive at an exact solution, we do not use our computational power to reach convergence.Further, in direct methods, once the decomposition is complete in the first iteration, for subsequent iterations we need only to compute backward and/or forward substitutions to reach the solution. However because they decompose the matrix, we are required to store two(Cholesky) or three (LU Decomposition) matrices. Therefore as the size of input matrices increase, the required memory for storing the data grows at O(n^2) or O(n^3) as compared to O(n) for iterative methods. This eats up memory rapidly.

iv. Ill-Conditioned Matrices

In the usage of all our systems, we assume our matrices to be non-singular as a precondition to using our linear solvers. However for systems that are almost, but not quite singular in nature, i.e ill conditioned systems, linear solvers can show unexpected behaviour. We considered three ill conditioned system - the Hilbert Matrix with a condition number(calculated in python using the scipy library) of 15613, the Rutishauser System with a condition number of 44490 and the Wilkinson System with a condition number of 2.16×10^{16}. We can see from the calculation of this condition number that any small change in the inputs (i.e. the linear system) we are intending to solve will result in drastically different outputs. The tables below show the analyses for the different systems tested - both in python (for comparison purposes) and using our dense solvers.

Table 4: *Ill Condition Matrices*

Output Matrix	Hilbert (4x4) matrix	Rutishauser
Condition Number (Scipy)	15613.79	100 x100
Direct Methods	[-4, 60, -180, 140]	[223, 355, -646, 542]
Gauss Siedel	[-3.99, 59.99, -179.99, 139.99]	[223, 355, -646, 542]
Python	[-4 ,60, -180, 140]	[223, 355, -646, 542]

The table above shows the results output by our solvers against the output obtained in python for Hilbert and Rutishauser. We see that for these systems our solvers are quite reliable. Thus we can see that even though the system maybe ill-conditioned to a certain extent, our solvers give us a reliable output.
However when the condition number is very large(as calculated in scipy), such as for Wilkinson's Matrix,we fail to get a reliable output. This is driven by the fact that the diagonal number in the last row of matrix is the smallest (non zero) value in matrix and tends to zero as we divide it further during factorisation. This is the reason that direct methods do not give a valid outputs and when the Cholesky Solver was used, we get invalid results. Thus we can conclude that for higher levels of ill-conditioning precautions (such as preconditioning the matrix inputs) must be taken to avoid these failures.

Table 5: *Wilkinson's Matrix where our Solvers Fail*

Output Matrix	Wilkinson (before perturbation)	Wilkinson (after perturbations)
Condition Number (Scipy)	$2.16x10^{16}$	$2.16x10^{16}$
Direct Methods and Gauss Siedel	[1.10937, 59.2326, -500900, 5.04E+09]	[1, 1, 1, 1]
Python results	[1.10937, 59.2326, -500900, 5.04E+09]	[1, 1, 1.00000001, 9.9994E-01]

V. Comparative Analysis and Results

Thus in terms of solver choice, the following three considerations are to be taken into account by the user : a) Robustness/ the ability to solve for changing properties of a linear system; b) Performance/ time taken by the solver to converge/produce an output and finally; c) Computational cost in terms of memory management and consumption by the solver Keeping under consideration these three parameters the following table enumerates our recommendations for solver selection based on the user's priority area of optimization.

Table 6: *Solver Selection Table*

Solver	Solver Order (Best to Worst)
Robustness	Cholesky > LU Decomposition > Gauss Elimination > Conjugate Gradient > Gauss-Siedel > Jacobi
Speed	Conjugate Gradient > Gauss-Siedel > Jacobi > Cholesky > LU Decomposition > Gauss Elimination
Memory Usage	Gauss -Siedel > Jacobi > Conjugate Gradient > Cholesky, LU Decomposition, Gauss Elimination)

VI. Discussion on Further Improvements in Code

- In terms of code structure, we could further improve our code by splitting the testing file into tests for developer and tests for users. This will allow the Test class to be used as a development and validation tool for the next versions while still allowing the users of the library to perform validation and error testing. The test class can be further improved by encapsulating our code for validations and performance into callable functions. This will avoid code repetitions which is in line with good design principles.
- We can extend our functionalities to support other different types of input including banded matrices and other types of sparse solvers. This library could also be extended with more functionalities for matrix and CSR matrix operations such as finding getting and setting values, finding the diagonals etc.
- There are certain errors in Read-Files function when handling matrices of very large sizes (such as 1000x1000). Better methods to generate and store inputs can be considered for more accurate and efficient computations.
- For iterative solvers, other than the two stopping conditions we have considered (a tolerance and a maximum number of iterations input by the user), to better understand the nature and usability of the results, there might be a strong reason to add a function that allows us to check for divergence.

References

[1] Lecture 5 : MSc ACSE - Advanced Programming Coursework. *Gauss Elimination and LU Decomposition*, .

[2] The Jacobi and Gauss-Seidel Iterative Methods and Imperial College London Library. *Science Direct*.

[3] An Introduction to the Conjugate Gradient Method Without the Agonizing Pain by Jonathan Richard Shewchuk. *Edition 114*.

[5] Notes on Cholesky Factorization Robert A. van de Geijn Department of Computer Science The University of Texas Austin

[6] How to Generate Random Matrices from the Classical Compact Groups by Francesco Mezzadri. *Volume 54, Number 5*.

[7] What is Wilson Matrix, Nick Higham. *https://nhigham.com/2021/06/01/what-is-the-wilson-matrix/*.

Yuna - Testing, Analysis, Gauss Siedel Ellya - Cholesky, Conjugate Gradient, LU Decomposition,Jacobi, Analysis Niranjana - Gauss, Conjugate Gradient, Jacobi, Interface, Report