Project 1: Linear Systems

Philippine des Courtils, Stella Petronio

January 2021

1 Presentation

The project consists in the implementation of several direct and iterative methods for the solution of the linear systems $A\mathbf{x} = \mathbf{b}$, taking advantage of the performing tools offered by the C++ Object-Oriented Language. Precisely, regarding the direct methods we focused on:

- The **LU factorization** method, which consists in determining the *LU factorization* of the matrix A ($A = LU$), if it exists; solving the lower triangular system $L\mathbf{y} = \mathbf{b}$ with the *forward substitutions algorithm* and solving the upper triangular system $U\mathbf{x} = \mathbf{y}$ with the *backward substitutions algorithm*;
- The **Cholesky factorization** method, which consists determining the *Cholesky factorization* of the matrix A ($A = R^T R$); solving the lower triangular system $R^T\mathbf{y} = \mathbf{b}$ with the *forward substitutions algorithm*; solving the upper triangular system $R\mathbf{x} = \mathbf{y}$ with the *backward substitutions algorithm*.

For the iterative ones, we concentrated on the two following splitting methods:

- The **Jacobi** method, which can be used for a non singular matrix $A \in R^{nxn}$ with nonzero diagonal entries. It consists in setting the preconditioning matrix P as the diagonal matrix extracted from A. The *Jacobi algorithm* is: given $\mathbf{x}^{(0)} \in R^n$, compute $x_i^{k+1} = \frac{1}{a_{ii}}(b_i - \sum_{j=1, j\neq i}^{n} a_{ij}x_j^k)$, with i = 1,2, ... n and for k = 0, 1, ..;
- The **Gauss-Seidel** method, which can be used for a non singular matrix $A \in R^{nxn}$ with nonzero diagonal entries. It considers as preconditioning matrix P the lower triangular matrix extracted from A. The *Gauss-Seidel algorithm* is: given $\mathbf{x}^{(0)} \in R^n$, compute $x_i^{k+1} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i} -1a_{ij}x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij}x_j^k)$, with i = 1,2, ... n and for k = 0, 1, ..

And then on:

- The dynamic **Richardson** method which represents a generalization of the iterative method, reading: given $\mathbf{x}^{(0)} \in R^n$, solve $P\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$, with $P \in R^{nxn}$ a nonsingular preconditioning matrix, precisely in our method we set P equal to the identity matrix, and set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k\mathbf{z}^{(k)}$ for k = 0,1,.. and $\alpha_k = \frac{(\mathbf{z}^{(k)})^T\mathbf{r}^{(k)}}{(\mathbf{z}^{(k)})^T A\mathbf{r}^{(k)}}$;
- The **Conjugate Gradient** method which considers a symmetric and positive definite matrix $A \in R^{nxn}$ and compute the new iterate as $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k\mathbf{p}^{(k)}$ along a direction $\mathbf{p}^{(k)}$ which is A-conjugate (or A-orthogonal) to all the previous directions: $(\mathbf{p}^{(j)})^t A(\mathbf{p}^{(k)})$ for j = 0, ... , k-1.

Concerning the running time of each of the algorithms described above, we have that the number of operations required for the LU factorisation method is $O(\frac{2}{3}n^3)$, while the Cholesky algorithm requires $O(\frac{1}{3}n^3)$ operations to determine the matrix R, a number about the half of that of the LU factorisation. In addition, the memory storage is also inferior. With regard to iterative method,s we can say that the necessary and sufficient condition for the convergence is that the spectral radius of the iterative matrix has to be less than 1 and the smaller is it , the faster is the convergence.

The more common way to solve $A\mathbf{x} = \mathbf{b}$ with a direct method is characterized by the LU factorization method; the Cholesky factorization method is used for a symmetric and positive definite matrix A.
The choice to use a direct or an iterative method for the solution of the linear system $A\mathbf{x} = \mathbf{b}$ is strictly related to the properties of the matrix A itself and the computational resources available (CPU and memory). For example, if the size n of the matrix A is very large, direct methods can be conveniently chosen if A is sparse and banded while iterative methods are preferred if A is a full matrix.

2 Program design

2.1 Template classes

To begin with, we tried to generalise our code to any type of data. The numbers can be either complex or real, and this has been implemented through template classes. Splitting the code between header and source code files isn't trivial when dealing with such classes. This will lead to compilation or linking errors if not properly handled. Indeed, the compiler needs the template implementation code which cannot be accessible if written in a separate file, when instantiating a template method with a template argument. We overcome this issue by explicit class instantiations within source code files. Moreover, C++ forbids purely virtual template methods, so we carefully designed the mother classes.

2.2 Data stored type

We made the arbitrary decision to store the numbers into long doubles, even though our solvers, Matrices and Vectors are defined for other basic types. We firstly wanted to avoid integers since the results are rounded. Then, we chose double over float since they twice as precise, and the long specifier to store large entries. Regarding how to handle the complex entries, we adapted some solvers in order to solve the complex equations, making use of the <complex> library features and operations.

2.3 Classes

Our program presents the following classes with the relationship illustrated by the *Unified Modeling Language* (UML) diagram (fig. 1 and 2): the ***Matrix*** is a base template class which takes as members the number of rows and columns, both of type int, and a matrix of type STL vector of STL vectors. STL vectors are very convenient containers, because of their size flexibility and automatic memory deallocation. Moreover, inside the matrix class we have implemented several useful overloading operators to deal with operations between matrices and between a matrix and a scalar. We also added a read/write and read operators. Methods which return the transpose of a matrix, an extracted diagonal matrix, a lower triangular matrix and a upper triangular matrix are present.

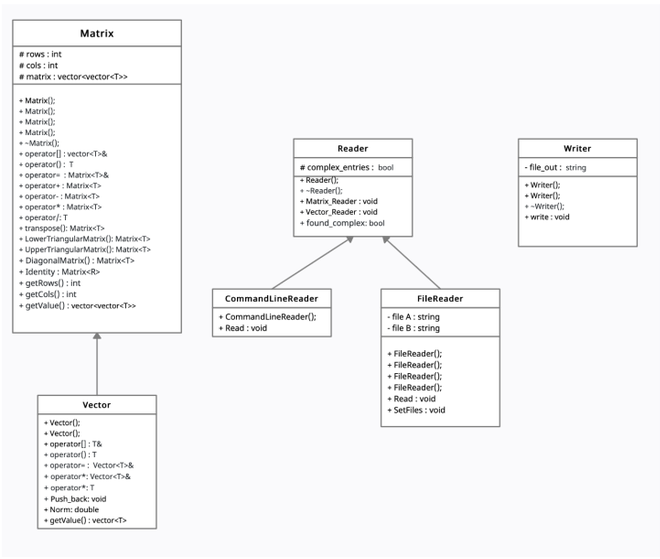

Figure 1: Matrix, Read and Write classes inheritance

As it can be seen from the UML diagram, the ***Vector*** class is a derived template class from the parent class Matrix. We made this choice because we have defined a Vector as a Matrix with one column, i.e. column vectors are Vectors but row vectors are Matrices. Along this line, the Vector class inherits functions of the base class. Additional functionalities specific to vectors such as the norm, the `Push_back` and inner-product operations have been implemented.

The ***Reader*** is a generic reader parent class from any istream. It has a boolean complex entry as a protected member of the class, which indicates the presence of any complex entries in the input. It also provides a generic function to read from a stream. The ***CommandLineReader*** and the ***FileReader*** are both derived classes from the Reader one, because both of them read from a input but in the first case from a line received by the keyboard and in the second case from a file. There is also a ***Writer*** class, which is separated from the Reader class. We made this choice because the two of them don't share anything, in fact the Writer is a generic class to write into a file, i.e. the solution of the linear system.

Regarding how to deal with the linear solvers, we have implemented a ***LinearSolver*** parent template class for any linear solver, which defines a convenient interface for the development of our linear solvers sub classes. This class could appear useless but, actually, this is not the case because it provides a very useful behaviour inside the main. Since any solver is a Linear solver, it is possible to take advantage of polymorphism and save the user's choice about the method he/she wants to solve his/her linear system in the LinearSolver$< T >*$ solver pointer, thus avoiding the use of several for loops. We benefit from the polymorphism with the virtual `Solve` method re-implemented in each of the daughter classes.
The ***NonIterative_Solver*** and the ***Iterative_Solver*** classes are both template derived classes from the parent LinearSolver class. As it is shown in the UML diagram (see fig. 2, both the `Lu` and ***Cholesky*** classes are derived classes from the ***NonIterative_Solver*** class because they both use the forward substitutions and the backward substitutions algorithms to solve the linear system. Concerning the structure of the iterative solvers, we have that ***Jacobi***, ***Gauss-Seidel***, ***Richardson*** and ***Conjugate-Gradient*** are all derived classes from the ***Iterative_Solver*** class for the fact that all of them share the number of iterations, the tolerance and a Vector as a initial guess.

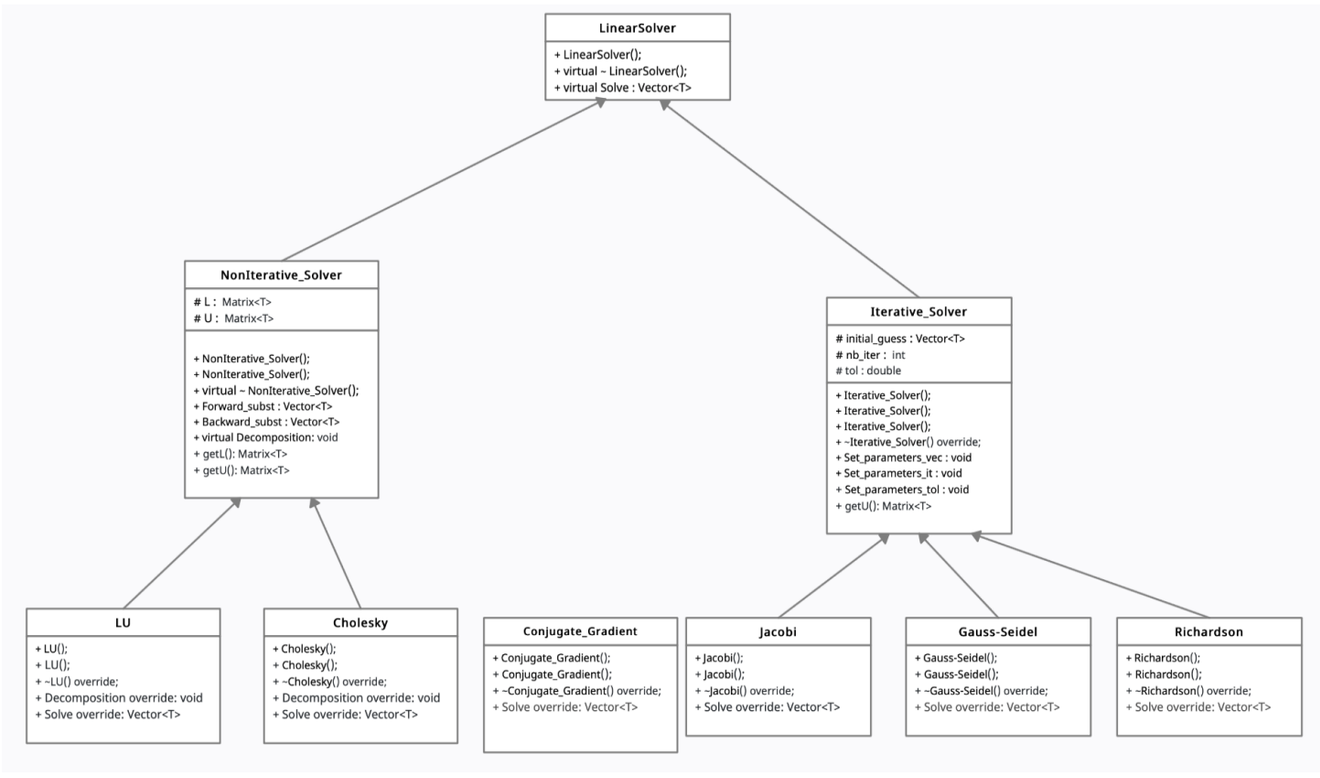

Figure 2: Linear solvers inheritance

3 Program execution and usage

3.1 Input format

3.1.1 Command line execution

Our program runs as described in fig. 4 can handle two types of inputs: command line and files inputs. If the program is expected to read from the command line, this has to be specified first when launching the program through the command `-C`. Then the user can enter row-wise each matrix and then each vector entries. Otherwise, relative paths to both matrix and vector files must be entered. Those files must be located under `data/`, and their relative name should begin with `..data/`. We provided on the right the TCLAP requirements. Moreover, if the files are likely to include any complex entry, the user shall specify it with the command `-I`.

```
USAGE:

   ./Linear_solver  [-I] -D <int> [-P <int>] -S <0|1|2|3|4|5> [-O <string>]
                    [-B <string>] [-A <string>] [-C] [--] [--version] [-h]


Where:

   -I,  --complex
     Specifies if there is any complex entry in the files

   -D <int>,  --dimension <int>
     (required)  Dimension of the square matrix

   -P <int>,  --precision <int>
     significant digits of solution

   -S <0|1|2|3|4|5>,  --solver <0|1|2|3|4|5>
     (required)  Method chosen to solve the linear system

     : 0:lu, 1:cholesky, 2: conjugate gradient, 3: jacobi, 4: gauss seidel,
     5: richardson


   -O <string>,  --out <string>
     Name of the output file storing the solution

   -B <string>,  --vector <string>
     Relative path of the file storing vector B

   -A <string>,  --matrix <string>
     Relative path of the file storing matrix A

   -C,  --terminal
     Specifies if matrix and vector are read from command line

   --,  --ignore_rest
     Ignores the rest of the labeled arguments following this flag.

   --version
     Displays version information and exits.

   -h,  --help
     Displays usage information and exits.
```

Figure 3: TCLAP output

3.1.2 Files

As long as it is a text file, any file extension is supported. However, the entries must follow a specific convention. Indeed, we use the `<complex>` STL library from C++ to handle complex operations, and complex numbers syntax is `(a,b)` where a stands for the real part, and b for the imaginary one. The entries must be separated by tabs.

Additional care must be taken regarding the input matrix and the chosen solver. First of all, the non iterative methods require some specific structure. While LU decomposition with pivot is guaranteed to exist for any matrix, this is not the case for LU decomposition. Moreover, Cholesky decomposition only applies to positive definite matrices. Then, the iterative solvers convergence depends on particular matrix properties. A general condition is that the spectral radius of the matrix is less than one, as stated above. The Jacobi method will converge under the sufficient condition that the matrix is strictly diagonally dominant. Conjugate gradient, Gauss Seidel and Richardson methods are guaranteed to converge if the matrix is symmetric and positive definite. To finish with, Richardson method also converges under some conditions on the matrix eigenvalues and conditioning number. Therefore, the user should be aware of those requirements.

3.2 Output format

As stated in the command line options, the user can also specify the name of the file storing the solution, along with the scientific precision. The solution file will eventually be stored under the `Solution/` folder.

3.3 Usage

We provide here two execution examples.
`./Linear_solver -D 4 -S 3 -A "../data/ItSolver/Mat4x4.txt" -B "../data/ItSolver/VecB4x1.txt" -O "sol.txt"`
This command should output: `Solution saved to file sol.txt`.

`./Linear_solver -I -C -S 0 -O Lu.txt -D 3` should print

```
Enter matrix entries, row by row (separated by linespaces) :
(5,2)    (0,0)    (3,5)
(2,7)    (1,-3)   (0,7)
(0,1)    (12,-5)  (-7,0)
Enter vector entries element-wise (separated by linespaces):
1
1
1
Matrix and Vector correctly stored !
Solution saved to file Lu.txt
```

4 Available tests

The available tests are described below and can be called under the syntax `.\test_name` in the `build/` folder, once everything has been compiled.

4.1 Interface tests:

We have assessed the robustness of our read and write functions by providing them different input formats. First of all, we analysed whether they could handle correct inputs. We then tested if they could detect unexpected complex values in command line or file inputs. Eventually, we tried out inputs with incorrect dimensions. This test is used under the following name: `test_interface`.

4.2 Matrix tests:

We examined the different constructors present in the Matrix class, the copy constructor and all the operations between two matrices and between a matrix and a scalar. We tested also the methods which return respectively the transpose, the diagonal, the lower triangular and the upper triangular of a matrix. All those methods are tested on a matrix of dimension 3. Finally we studied the case to cast a 1x1 matrix into a scalar. This test is used under the following name: `test_Matrix`.

4.3 Vector tests:

We investigated the different constructors, copy constructors, and operators we have derived. We enforced the fact that only a Matrix with one unique column can be assigned to a Vector, and that it was convenient to equalise a Vector with a STL vector. We have also studied the case where two Vectors could be multiplied together, i.e. as a dot product, and therefore their length must be the same. Additionally, two Vectors of size 1 can be seen as scalars and therefore it makes sense to define a division operator for two objects of size 1. Finally, we tested some linear operations on Vectors storing complex numbers. This test is used under the following name: `test_Vector`.

4.4 Non Iterative solver tests:

We have implemented our tests for both complex and real valued Matrices and Vectors. First of all, we have tested correctness of LU and Cholesky decomposition, and of the solutions computed by backward and forward substitutions. We used sparse matrices of size 20 for real-valued matrices tests. This test is used under the following name: `test_nonItSol`.

4.5 Iterative solver tests:

Also for the iterative solvers, we have implemented our tests for both complex and real valued Matrices and Vectors. Concerning real-valued entries we started to verify the correctness of the solution of our Jacobi, Gauss-Seidel, Richardson and Conjugate-Gradient methods for a simple case, precisely, for 2x2 symmetric and positive definite matrix. Then we tested them on a 4x4 a symmetric and positive definite matrix, and finally on a sparse matrix of dimension 20x20. Regarding the test for complex entries we used a symmetric and positive definite matrix of dimension 2. This test is used under the following name: `test_ItSol`.

5 Conclusion

5.1 Limitations

We have implemented the operations between matrices and vectors such that the default return type of any operation is a Matrix. The user should therefore know in advance the dimensions of the result.
Let's take a basic example: $A \cdot b = c$, where b, c are Vectors. If the user needs to exploit the features of the result c as a Vector, this result should be recast into a Vector. Let's take another example: $b \cdot A = c$, where those vectors are row vectors. The output format is a Matrix here and not a Vector, and this aspect can be improved in future work.
Another challenge was to deal with size-1 Matrices and Vectors. We should mention here that the dot product between two Vectors of same dimensions outputs a scalar, while the product of a row vector and a column Vector is seen as a Matrix of dimension 1. However, we derived a method which converts any Matrix of dimension 1 into a scalar. Here again, the user needs to anticipate the output dimensions.

5.2 Perspectives

First of all, LU decomposition can be improved by introducing a permutation matrix, and therefore computing a LUP decomposition, which exists for any matrix. Then, we could try to generalise our code to any input. It would require however some computationally expensive tests, such as computing the determinant or the eigenvalues, to determine matrix properties. Regarding the requirement for a faster convergence for the iterative methods in general, a smaller spectral radius is asked; but focusing on the Richardson method a good perspective could be a different choice of the preconditioning matrix in order to achieve a smaller conditioning number. Finally, more work can be invested in handling dimension 1 Matrices and Vectors, for further linear algebra operations.

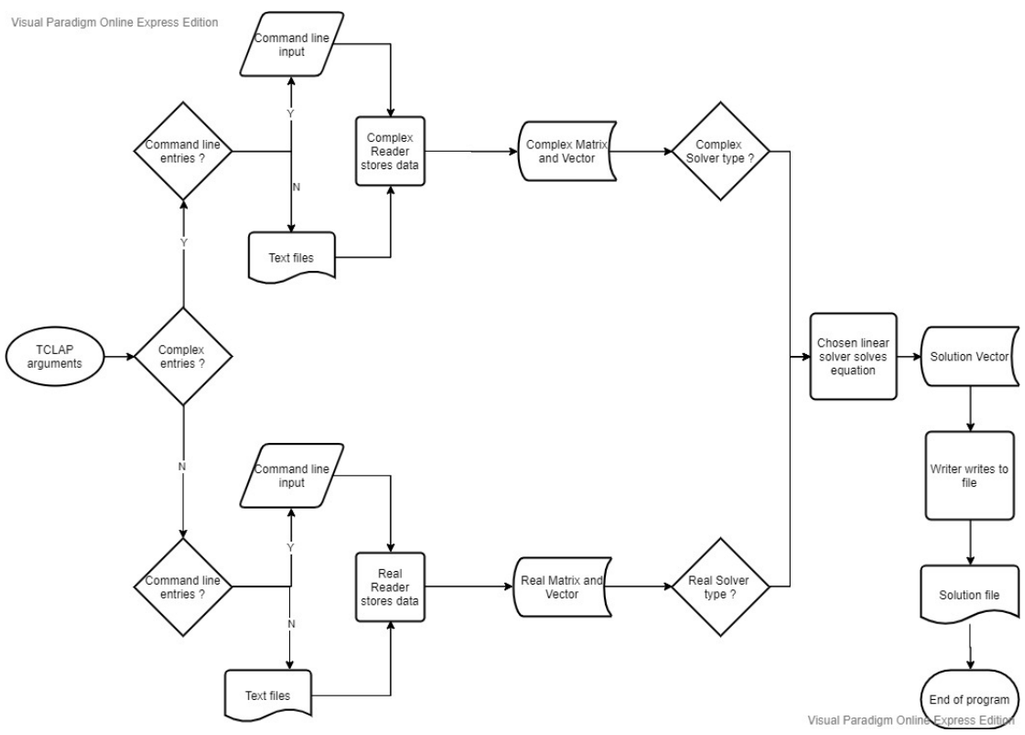

Figure 4: Flow diagram of the program