Project 3 of "Metodi Del Calcolo Scientifico"

Academic Year 2019/2020

Simone Paolo Mottadelli, 820786

Contents

1 Introduction

The main purpose of this project was to implement different solvers of systems of linear equations and organize them into a library. In particular, the following iterative solvers had to be implemented, limiting to the case of symmetric and positive definite matrices:

- The Jacobi method;
- The Gauß-Seidel method;
- the Gradient method;
- the Conjugate Gradient method.

This document consists of three sections: Section 2 describes how the library has been implemented; Section 3 describes the validation study that has been conducted with the developed library and, finally, Section 4 concludes this work.

2 Implementation

The required library has been implemented using the **Python** [1] programming language because it is open source, very powerful and it provides all the functionalities and libraries needed to achieve the goals of the project. In particular, the following libraries have been exploited:

- **NumPy** [2], which is the core library for scientific computing in Python [1], which mainly adds support for dense multidimensional arrays. In particular, it provides the *ones()* and the *zeros()* functions to generate a vector with all the entries equal to 1 or to 0, respectively;
- **SciPy** [3], which is a Python-based ecosystem of open-source software for mathematics, science and engineering, which also adds support for sparse matrices.

In order to improve the readability of the code and, thus, its maintainability, the sofware has been organized into the following modules and classes:

- **main.py**, which is the entry point of the program and which controls the execution flow of the program;
- **input_parser.py**, which implements the logic for validating the user input in the *InputParser* class;

- **mtx_file_reader.py**, which contains the *MTXFileReader* class, whose purpose is to implement the logic for loading the matrix in input from an .mtx file and store it in memory;
- **iterative_solver_comparator.py**, containing the *IterativeSolverComparator* class, which is responsible for comparing the four aforementioned solvers in terms of their execution time, relative error and number of iterations.
- **abstract_iterative_solver.py**, which contains the *AbstractIterativeSolver* class, which is the template for all the iterative solvers of the library, i.e., it implements the logic of a general iterative solver;
- **jacobi_solver.py**, containing the *JacobiSolver* class, which realizes the logic for updating the approximate solution based on the Jacobi updating strategy;
- **gauss_seidel_solver.py**, containing the *GaussSeidelSolver* class, which implements the logic for updating the approximate solution based on the Gauß-Seidel updating strategy;
- **gradient_solver.py**, containing the *GradientSolver* class, which realizes the logic for updating the approximate solution based on the Gradient updating strategy;
- **conjugate_gradient_solver.py**, containing the *ConjugateGradientSolver* class, which realizes the logic for updating the approximate solution based on the Conjugate Gradient updating strategy;

From an architectural point of view, Figure 1 shows the structure of the project through a **UML** [4] diagram. It is a simple diagram showing both the dependencies between the software components and the attributes and methods of each class at a high level of abstraction. In fact, every software class is modelled as a rectangle: on top there is the name of the class, in the middle there are the attributes of the class (if any), each specified by a data type, and on the bottom there are the signatures of the methods. The links between the rectangles model the relationships between the software components: a class can use another class in its implementation or a class can extend another class. Note that the *main.py* module is just a simple script that implements the entry point of the program and that controls the execution flow of the program. This is why it has not been modelled as a class rectangle.
More in detail, as it is shown in the UML diagram and in Figure 2, the *main.py*

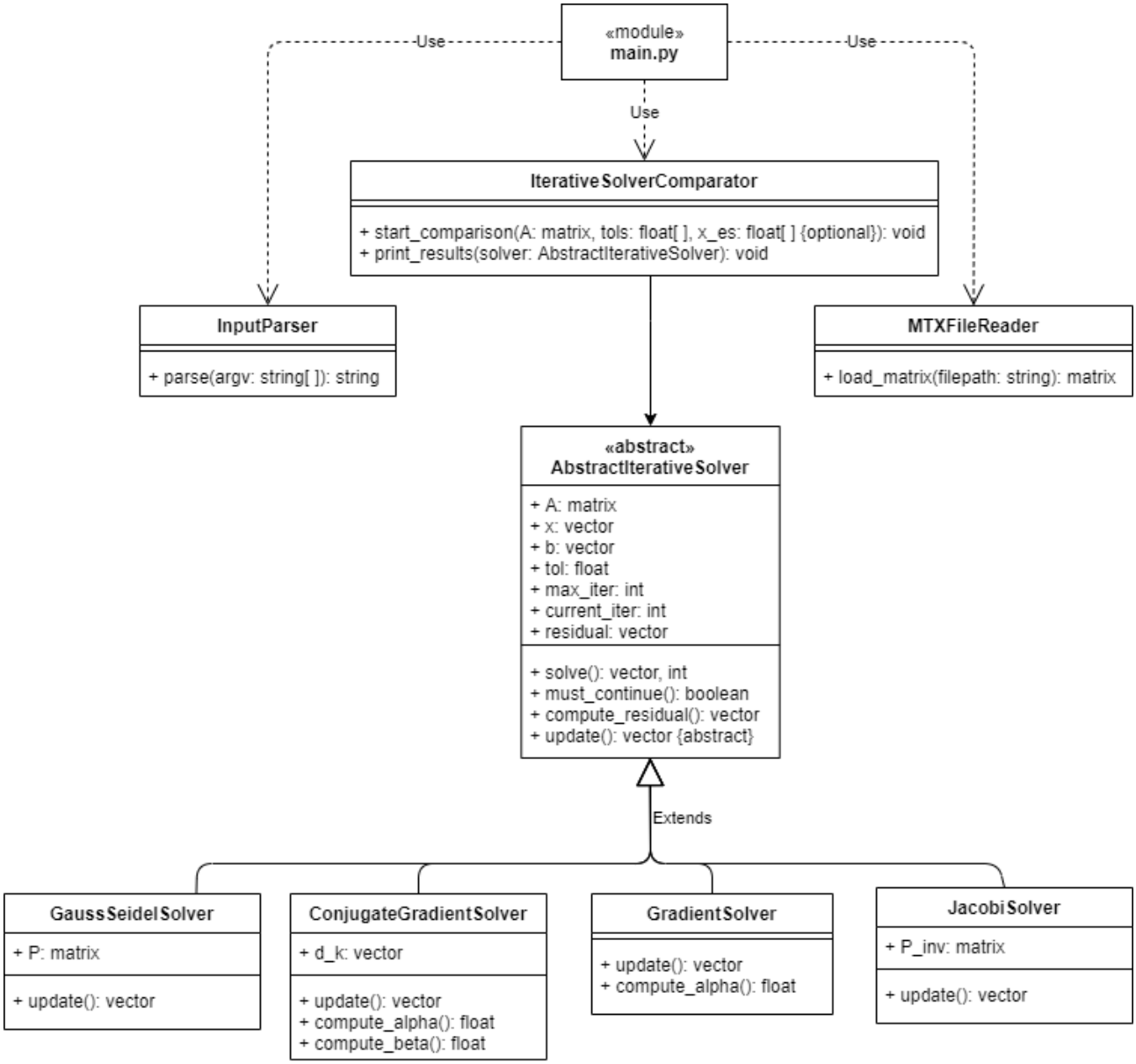


Figure 1: A UML [4] diagram that shows the structure of the software

module uses the *InputParser* class to parse and validate the user input, that is, it checks whether the .mtx file exists in the file system. Then, as soon as the user input has been validated, the *main.py* module uses the *MTXFileReader* class to load the matrix from the file system and store it in memory as sparse. Finally, the *main.py* module uses the *IterativeSolverComparator* class to start the comparison between the different solver implementations and to print the results obtained at the end of the comparison.

```
def main(argv):
    # input validation
    mtx_file = InputParser().parse(argv)

    # matrix extraction from the .mtx file
    A = MTXFileReader().load_matrix(mtx_file)

    # launch comparison
    tols = [1e-4, 1e-6, 1e-8, 1e-10]
    IterativeSolverComparator().start_comparison(A, tols)
```

Figure 2: Code inside the *main.py* module

```
class IterativeSolverComparator:

    # This is a helper method for the start_comparison() method:
    # it simply prints the results achieved by a solver to the console
    def print_results(self, solver, x_es):
        time_start = time()
        x_app, num_iter = solver.solve()
        time_end = time()
        exec_time = "{:.32e}".format(time_end - time_start)
        rel_error = "{:.32e}".format(norm(x_es - x_app) / norm(x_es))
        formatted_tol = "{:.0e}".format(solver.tol)
        solver_name = type(solver).__name__
        print("\n%s results with tolerance = %s:" % (solver_name,
                                                     formatted_tol))
        print("Relative error = %s" % rel_error)
        print("Number of iterations = %d" % num_iter)
        print("Execution time = %s\n" % exec_time)

    # This method simply solves the linear systems with the four different
    # solver to the varying of the tolerance and shows the achieved results
    # on the console
    def start_comparison(self, A, tols, x_es=None):
        if x_es is None:
            x_es = ones(A.shape[0])
        b = A.dot(x_es)
        for tol in tols:
            self.print_results(JacobiSolver(A, b, tol), x_es)
            self.print_results(GaussSeidelSolver(A, b, tol), x_es)
            self.print_results(GradientSolver(A, b, tol), x_es)
            self.print_results(ConjugateGradientSolver(A, b, tol), x_es)
```

Figure 3: Code inside the *IterativeSolverComparator* class

As shown in Figure 3, when the *start_comparison()* method of the *IterativeSolverComparator* class is called, first the *x_es* vector and the b vector are generated: the former has all its entries equal to 1 and it represents the exact solution of the system of equations, while the latter represents the right-hand side coefficients of the system and it is generated by using the A matrix in input

and the just created *x_es* vector. Note that it is still possible to use a different *x_es* vector, as also specified in the signature of the method. Soon afterwards, the *IterativeSolverComparator* class uses all the different solvers to solve the system of equations $Ax = b$ to the varying of the tolerance, which is a value among $[10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}]$, and prints on the screen the relative error, the number of iterations and the execution time needed to generate a solution.

```
class AbstractIterativeSolver:

    # Constructor
    def __init__(self, A, b, tol):
        self.A = A
        self.b = b
        self.tol = tol
        self.x = zeros(A.shape[0]) # A.shape[0] gets the first dimension
        self.max_iter = 20000
        self.current_iter = 1
        self.residual = self.compute_residual()

    # Helper method that computes the residual
    def compute_residual(self):
        return self.b - self.A.dot(self.x)

    # Helper method that returns TRUE if the solver must continue to
    # iterate, FALSE otherwise.
    def must_continue(self):
        if self.current_iter > self.max_iter:
            print("[WARNING] max number of iterations reached: solver failed
                                                    to achieve convergence!")
            return False
        return norm(self.residual) / norm(self.b) >= self.tol

    # Template method for a general iterative method
    def solve(self):
        while self.must_continue():
            self.x = self.update()
            self.residual = self.compute_residual()
            self.current_iter += 1
        return self.x, self.current_iter

    # Method that must be overridden by the subclasses.
    def update(self):
        raise NotImplementedError("[ERROR] This method has not been
                                        implemented yet")
```

Figure 4: Implementation of the general iterative solver

All the solvers extend the *AbstractIterativeSolver* class, thus inheriting all its methods and attributes. The code of such class can be inspected in Figure 4 and, as it can be seen, the *solve()* method implements the main loop, which stops if either the maximum number of iterations is reached or the scaled residual is less than the fixed tolerance. At each iteration of the main loop, the *update()* method updates the approximate solution, but, since the update strategy strictly depends on the specific solving method (e.g., the Jacobi method), the *update()* method is abstract. More in detail, an abstract method is such that all the iterative solvers that extend the *AbstractIterativeSolver* class must override it, providing an implementation of such method (abstract methods are analogous to the concept of virtual methods in the *C++* [5] programming language). With this design pattern, implementing the solvers is a trivial task, since they only have to extend the *AbstractIterativeSolver* class and implement the *update()* method.

```
class JacobiSolver(AbstractIterativeSolver):

    # Constructor
    def __init__(self, A, b, tol):
        super().__init__(A, b, tol)
        self.P_inv = csr_matrix(identity(A.shape[0]) / A.diagonal())

    # This function implements the update strategy of the Jacobi method
    def update(self):
        return self.x + self.P_inv.dot(self.residual)
```

Figure 5: Implementation of the Jacobi method

For instance, Figure 5 shows the implementation of the Jacobi method, which consists of just the *update()* method. It is worthwhile to notice that the inverse of the P matrix is computed only once inside the constructor, where the P matrix is the diagonal matrix resulting from the factorization of the A matrix.
Figure 6 shows the implementation of the Gauß-Seidel method. Note that also in this case, for matters of efficiency, the P lower triangular matrix is computed just once using the *tril()* function provided by the SciPy [3] library. In addition, the *spsolve_triangular()* method, also provided by the SciPy [3] library, has been used to efficiently solve the system of equations where P is the coefficient matrix and the residual vector is the right-hand side of the system.
Lastly, Figure 7 and Figure 8 show the code used to implement the Gradient solver and the Conjugate Gradient solver, respectively. In order to improve the maintainability of the code, the *compute_alpha()* and the *compute_beta()* helper methods have been implemented.

```
class GaussSeidelSolver(AbstractIterativeSolver):

    # Constructor
    def __init__(self, A, b, tol):
        super().__init__(A, b, tol)
        self.P = tril(self.A, format="csr")

    # This method simply computes the next vector solution x
    def update(self):
        return self.x + spsolve_triangular(self.P, self.residual)
```

Figure 6: Implementation of the Gauß-Seidel method

```
class ConjugateGradientSolver(AbstractIterativeSolver):

    # Constructor
    def __init__(self, A, b, tol):
        super().__init__(A, b, tol)
        self.d_k = self.residual

    # This method computes the next vector solution x based on the Conjugate
    # Gradient update strategy
    def update(self):
        y_k = self.A.dot(self.d_k)
        alpha = self.compute_alpha(y_k)
        self.x = self.x + alpha * self.d_k
        self.residual = self.compute_residual()
        beta = self.compute_beta(y_k)
        self.d_k = self.residual - beta * self.d_k
        return self.x

    # This is a helper function for the update() method and simply computes
    # the value of "alpha"
    def compute_alpha(self, y_k):
        alpha_numerator = self.d_k.dot(self.residual)
        alpha_denominator = self.d_k.dot(y_k)
        return alpha_numerator / alpha_denominator

    # This is a helper function for the update() method and simply computes
    # the value of "beta"
    def compute_beta(self, y_k):
        beta_numerator = self.d_k.dot(self.A.dot(self.residual))
        beta_denominator = self.d_k.dot(y_k)
        return beta_numerator / beta_denominator
```

Figure 7: Implementation of the Conjugate Gradient method

```python
class GradientSolver(AbstractIterativeSolver):

    # This method computes the next x vector solution using the Gradient
    # update strategy
    def update(self):
        alpha = self.compute_alpha()
        return self.x + alpha * self.residual

    # This is a helper function for the update() method and simply computes
    # the value of "alpha"
    def compute_alpha(self):
        alpha_numerator = self.residual.dot(self.residual)
        alpha_denominator = self.residual.dot(self.A.dot(self.residual))
        return alpha_numerator / alpha_denominator
```

Figure 8: Implementation of the Gradient method

In conclusion, from a practical point of view, to launch this program it is sufficient to execute the following command on a command prompt window (e.g., Windows PowerShell or Linux Bash):

```
> python main.py <filepath>
```

For example, to launch it on the matrix saved in *my_file.mtx*, it is sufficient to execute the following command:

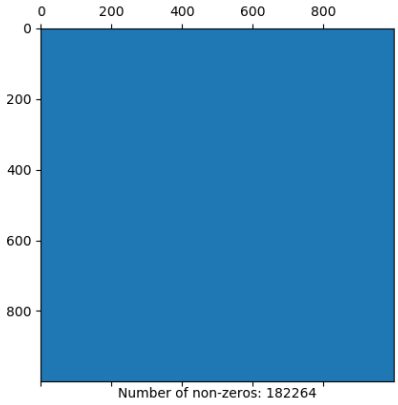
```
> python main.py my_file.mtx
```

For further details on the implementation of this library, all the code is available at:

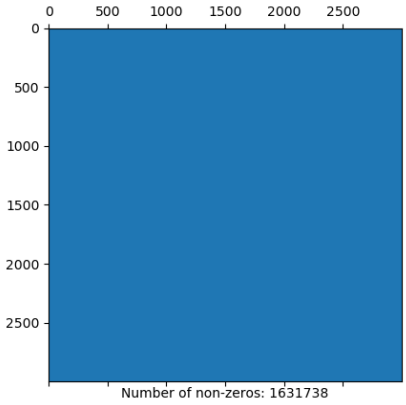
`https://bit.ly/3dvpX4y`.

3 Validation study

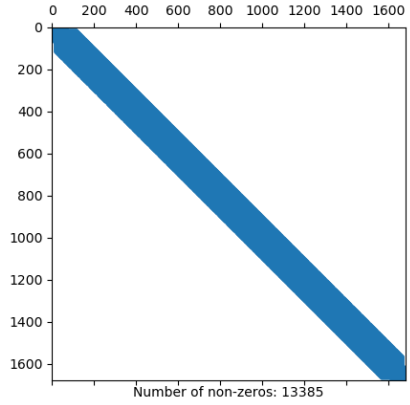
Once the library has been implemented, a validation study has been conducted with the aim of assessing the effectiveness and the efficiency of the different solvers of the library. The experiments have been conducted with the same machine equipped with an Intel Core i-7-4720HQ processor and 8 GB of RAM. In particular, the 4 sparse matrices (*spa1.mtx*, *spa2.mtx*, *vem1.mtx*, *vem2.mtx*) visualized in Figure 9 have been used. It is important to notice that *spa1.mtx* and *spa2.mtx* do not present a particular pattern, instead *vem1.mtx* and *vem2.mtx* have the non-zero coefficients concentrated on the main diagonal. In addition, Table 10 shows some more information about the matrices. For example, it is possible to notice that *vem1.mtx* and *vem2.mtx* are more sparse than the other two matrices.



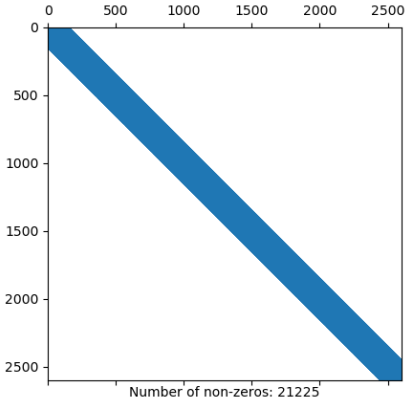
(a) spa1.mtx



(b) spa2.mtx



(c) vem1.mtx



(d) vem2.mtx

Figure 9: Spy plots of the matrices used during the experiments

Matrix	**Condition number**	**Dimension**	**Density**
spa1.mtx	2048.15	1000×1000	0.18
spa2.mtx	1411.97	3000×3000	0.18
vem1.mtx	324.64	1681×1681	0.0047
vem2.mtx	507.02	2601×2601	0.0031

Figure 10: Condition number of each matrix used for the experiments

Every solver of the library has been executed on the aforementioned 4 matrices using different values of the tolerance (i.e., $[10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}]$). For each execution, the following quantities have been registered:

- Number of iterations;
- Relative error, computed as $\frac{||x_{app} - x_{es}||}{||x_{es}||}$, where x_{app} is the approximate solution returned by a solver and x_{es} is the exact solution;

- Execution time in seconds.

While the number of iterations and the relative error do not depend on the specific execution, because the methods are deterministic algorithms, the execution time strictly depends on the scheduler of the operating system, which decides how much CPU time to allocate to each process. For this reason, at each run of a method, a different execution time could be registered. To mitigate this problem, the experiment has been repeated 10 times and the execution time has finally been computed by averaging the values observed in such experiments. The results obtained are shown in the following tables (the results in the table have been rounded to the second decimal place):

spa1.mtx with Tolerance: 10^{-4}			
Method	**Iterations**	**Rel. error**	**Time (in seconds)**
Jacobi	115	1.77×10^{-3}	2.30×10^{-2}
Gauß-Seidel	9	1.82×10^{-2}	7.30×10^{-2}
Gradient	143	3.46×10^{-2}	5.20×10^{-2}
Conjugate Gradient	49	2.08×10^{-2}	3.50×10^{-2}

spa1.mtx with Tolerance: 10^{-6}			
Method	**Iterations**	**Rel. error**	**Time (in seconds)**
Jacobi	181	1.80×10^{-5}	3.50×10^{-2}
Gauß-Seidel	17	1.3×10^{-4}	1.20×10^{-1}
Gradient	3577	9.68×10^{-4}	1.28
Conjugate Gradient	134	2.55×10^{-5}	9.40×10^{-2}

spa1.mtx with Tolerance: 10^{-8}			
Method	**Iterations**	**Rel. error**	**Time (in seconds)**
Jacobi	247	1.82×10^{-7}	5×10^{-2}
Gauß-Seidel	24	1.71×10^{-6}	1.71×10^{-1}
Gradient	8233	9.82×10^{-6}	2.89
Conjugate Gradient	177	1.32×10^{-7}	1.30×10^{-1}

spa1.mtx with Tolerance: 10^{-10}			
Method	**Iterations**	**Rel. error**	**Time (in seconds)**
Jacobi	313	1.85×10^{-9}	6.30×10^{-2}
Gauß-Seidel	31	2.25×10^{-8}	2.20×10^{-1}
Gradient	12919	9.82×10^{-8}	4.64
Conjugate Gradient	200	1.20×10^{-9}	1.39×10^{-1}

Table 1: Results with *spa1.mtx* matrix to the varying of the tolerance

spa2.mtx with Tolerance: 10^{-4}			
Method	**Iterations**	**Rel. error**	**Time (in seconds)**
Jacobi	36	1.77×10^{-3}	7.80×10^{-2}
Gauß-Seidel	5	2.60×10^{-3}	1.47×10^{-1}
Gradient	161	1.81×10^{-2}	$5,78 \times 10^{-1}$
Conjugate Gradient	42	9.82×10^{-3}	3.23×10^{-1}

spa2.mtx with Tolerance: 10^{-6}			
Method	**Iterations**	**Rel. error**	**Time (in seconds)**
Jacobi	57	1.67×10^{-5}	1.10×10^{-1}
Gauß-Seidel	8	5.14×10^{-5}	2.39×10^{-1}
Gradient	1949	6.69×10^{-4}	7.27
Conjugate Gradient	122	1.20×10^{-4}	9.28×10^{-1}

spa2.mtx with Tolerance: 10^{-8}			
Method	**Iterations**	**Rel. error**	**Time (in seconds)**
Jacobi	78	1.57×10^{-7}	1.49×10^{-1}
Gauß-Seidel	12	2.79×10^{-7}	3.59×10^{-1}
Gradient	5087	6.87×10^{-6}	$1.87 \times 10^{+1}$
Conjugate Gradient	196	5.59×10^{-7}	1.47

spa2.mtx with Tolerance: 10^{-10}			
Method	**Iterations**	**Rel. error**	**Time (in seconds)**
Jacobi	99	1.48×10^{-9}	2.11×10^{-1}
Gauß-Seidel	15	5.57×10^{-9}	5.64×10^{-1}
Gradient	8285	6.94×10^{-8}	$3.42 \times 10^{+1}$
Conjugate Gradient	240	5.32×10^{-9}	2.00

Table 2: Results with *spa2.mtx* matrix to the varying of the tolerance

vem1.mtx with Tolerance: 10^{-4}			
Method	**Iterations**	**Rel. error**	**Time (in seconds)**
Jacobi	1314	3.54×10^{-3}	7.10×10^{-2}
Gauß-Seidel	659	3.51×10^{-3}	6.67
Gradient	890	2.70×10^{-3}	6.70×10^{-2}
Conjugate Gradient	38	4.08×10^{-5}	5.01×10^{-3}

vem1.mtx with Tolerance: 10^{-6}			
Method	**Iterations**	**Rel. error**	**Time (in seconds)**
Jacobi	2433	3.54×10^{-5}	1.28×10^{-1}
Gauß-Seidel	1218	3.53×10^{-5}	$1.23 \times 10^{+1}$
Gradient	1612	2.71×10^{-5}	1.23×10^{-1}
Conjugate Gradient	45	3.73×10^{-7}	7×10^{-3}

vem1.mtx with Tolerance: 10^{-8}			
Method	**Iterations**	**Rel. error**	**Time (in seconds)**
Jacobi	3552	3.54×10^{-7}	1.95×10^{-1}
Gauß-Seidel	1778	3.52×10^{-7}	$2.05 \times 10^{+1}$
Gradient	2336	2.70×10^{-7}	2.25×10^{-1}
Conjugate Gradient	53	2.83×10^{-9}	9.01×10^{-3}

vem1.mtx with Tolerance: 10^{-10}			
Method	**Iterations**	**Rel. error**	**Time (in seconds)**
Jacobi	4671	3.54×10^{-9}	3.46×10^{-1}
Gauß-Seidel	2338	3.51×10^{-9}	$2.81 \times 10^{+1}$
Gradient	3058	2.71×10^{-9}	2.90×10^{-1}
Conjugate Gradient	59	2.19×10^{-11}	9.99×10^{-3}

Table 3: Results with *vem1.mtx* matrix to the varying of the tolerance

vem2.mtx with Tolerance: 10^{-4}			
Method	**Iterations**	**Rel. error**	**Time (in seconds)**
Jacobi	1927	4.97×10^{-3}	1.37×10^{-1}
Gauß-Seidel	965	4.95×10^{-3}	$1.85 \times 10^{+1}$
Gradient	1308	3.81×10^{-3}	1.75×10^{-1}
Conjugate Gradient	47	5.73×10^{-5}	1.10×10^{-2}

vem2.mtx with Tolerance: 10^{-6}			
Method	**Iterations**	**Rel. error**	**Time (in seconds)**
Jacobi	3676	4.97×10^{-5}	3.35×10^{-1}
Gauß-Seidel	1840	4.94×10^{-5}	$3.34 \times 10^{+1}$
Gradient	2438	3.79×10^{-5}	2.27×10^{-1}
Conjugate Gradient	56	4.74×10^{-7}	1.00×10^{-2}

vem2.mtx with Tolerance: 10^{-8}			
Method	**Iterations**	**Rel. error**	**Time (in seconds)**
Jacobi	5425	4.97×10^{-7}	3.66×10^{-1}
Gauß-Seidel	2714	4.96×10^{-7}	$5.63 \times 10^{+1}$
Gradient	3566	3.81×10^{-7}	3.30×10^{-1}
Conjugate Gradient	66	4.30×10^{-9}	1.30×10^{-2}

vem2.mtx with Tolerance: 10^{-10}			
Method	**Iterations**	**Rel. error**	**Time (in seconds)**
Jacobi	7174	4.96×10^{-9}	4.88×10^{-1}
Gauß-Seidel	3589	4.95×10^{-9}	$6.77 \times 10^{+1}$
Gradient	4696	3.80×10^{-9}	5.86×10^{-1}
Conjugate Gradient	74	2.25×10^{-11}	1.70×10^{-2}

Table 4: Results with *vem2.mtx* matrix to the varying of the tolerance

As shown in the Tables above, the best results are highlighted in gray. The following subsections describe the results obtained in detail.

3.1 Results with spa1.mtx and spa2.mtx

This subsection summarizes the results obtained with the *spa1.mtx* and *spa2.mtx* matrices, which do not present a particular pattern in their structure and they both have higher condition numbers and are less sparse than the other two matrices *vem1.mtx* and *vem2.mtx*.
As it is reported in Tables 1 and 2, the Jacobi method is the fastest method in terms of time complexity and it is very accurate considering the achieved relative error. Instead, considering the Gauß-Seidel method, although it takes the least number of iterations to achieve convergence, it is not the fastest method in terms of actual time required. These results highlight an important difference between the Jacobi method and the Gauß-Seidel method. In fact, the Gauß-Seidel method is a variant of the Jacobi method that it is based on the idea of exploiting the entries of the x vector just computed in the current iteration to compute the next entries of the vector. More formally, at the current k-th iteration, the generic i-th entry of the x vector, representing the current approximate solution, is computed as follows:

$$x_i^{(k+1)} = \frac{1}{a_{i,i}}(b_i - a_{i,1}x_1^{(k+1)} - ... - a_{i,i-1}x_{i-1}^{(k+1)} - a_{i,i+1}x_{i+1}^{(k)} - ... - a_{i,n}x_n^{(k)})$$

where b_i is the i-th entry of the RHS of the system and $a_{i,i}$ is the entry of the coefficient matrix in position (i, i).

With this improvement over the Jacobi method, the Gauß-Seidel method takes very few iterations to achieve convergence, but it is slower than the Jacobi method. This is due to the fact that a generic iteration in the Gauß-Seidel method is expensive in terms of time complexity, because at each iteration a system of linear equations must be solved. However, both the methods perform good with these metrices.
The Gradient method turned out to be inefficient with these matrices, because it required too many iterations and too much time to achieve convergence. Its results might improve if a different initial solution vector $x^{(0)}$ is chosen.
Finally, although the Conjugate Gradient method adds improvements to the simple Gradient method, considering the results obtained with the *spa2.mtx* matrix, it still takes too long to converge.
It is possible to conclude that if the coefficient matrix of the system of linear equations to be solved does not present a particular pattern in its structure, the Gauß-Seidel method and the Jacobi method are to be preferred to the other methods.

3.2 Results with vem1.mtx and vem2.mtx

This subsection summarizes the results obtained with the *vem1.mtx* and *vem2.mtx* matrices, which present the non-zero coefficients concentrated on their main diagonals and they both have lower condition numbers and are more sparse than the other two matrices *spa1.mtx* and *spa2.mtx*.
As it is reported in Tables 3 and 4, contrary to what happened in the previous matrices *spa1.mtx* and *spa2.mtx*, the Gauß-Seidel method is inefficient and slow, because it takes many iterations and a significant amount of time to achieve convergence, even with extremely low tolerance values. As already pointed out in Subsection 3.1, updating the current solution with the Gauß-Seidel method is expensive.
Interestingly, the Jacobi method is still a very fast method, even if it takes the greatest number of iterations to converge. This can be explained by the fact that updating the current approximate solution is less expensive in terms of time complexity than the other methods.
The Gradient and the Conjugate gradient method take less iterations to converge, because the condition numbers of the considered matrices are smaller than the ones of the previous matrices and, thus, the chosen initial solution vector $x^{(0)}$ has less influence on the convergence. In particular, the Conjugate Gradient method outperforms the other methods with respect to all the pa-

rameters considered in the evaluation. In fact, its values are always coloured in gray in Tables 3 and 4.
It is possible to conclude that if the coefficient matrix of the system of linear equations to be solved presents the non-zeros coefficients concentrated along the main diagonal and is very sparse, the Conjugate Gradient method is to be preferred to the other methods.

3.3 Further considerations on the Gradient and Conjugate Gradient methods

This section highlights a particularity that has been found by observing the graphs related to the curves of the scaled residuals to the varying of the iterations with respect to the Gradient and the Conjugate Gradient methods, which are illustrated in Figures 11 and 12.
In particular, it can be observed that the Gradient and Conjugate Gradient methods have a more irregular convergence, as better shown in Figure 13. The figure shows a zoomed view of both the Gradient graph and the Conjugate Gradient graph with respect to Figure 12. It is possible to see that both the methods do not present a monotonous decrease in the scaled residual and this can be explained by the fact that the Gradient and the Conjugate gradient methods are not based on splitting and, thus, Proposition 1 does not hold.

Proposition 1. *Suppose that A can be factored as $A = P - N$, where A and P are symmetric and definite positive matrices. If $2P - A$ is also a definite positive matrix, then the iterative method defined as $x^{(k+1)} = P^{-1}Nx^{(k)} + P^{-1}b$ converges for each initialization of $x^{(0)}$ and*

$$|\lambda_{max}| < 1,$$

where λ_{max} is the maximum-modulus eigenvalue of the $P^{-1}N$ matrix. In addition, the convergence is monotonous with respect to the norms $||\,.\,||_p$.

However, the Conjugate Gradient method was introduced in the literature also to solve the "zig-zag" problem of the simple Gradient method and this is why it converges with less iterations and the curve of the residuals in Figure 13 is more regular. In particular, the number of iterations observed during the experiments for the Conjugate Gradient method are conform to what it is asserted in Theorem 1.

Theorem 1. *Let $A \in R^{n\times n}$ be a symmetric positive definite matrix, then the Conjugate Gradient method converges in at most n iterations.*

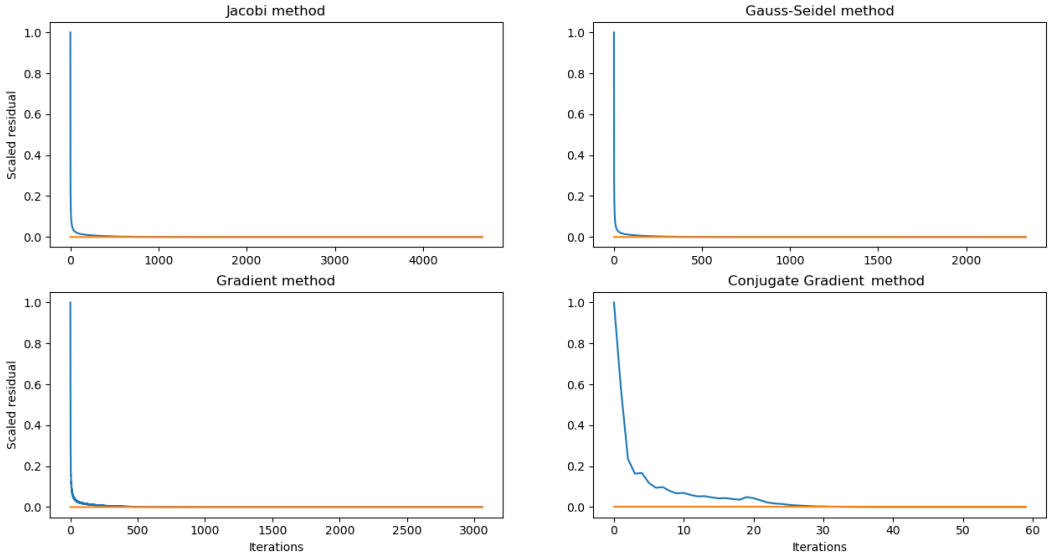


Figure 11: Comparison between the different methods in terms of the scaled residual and the number of iterations in *vem1.mtx* with tolerance equals to 10^{-10}

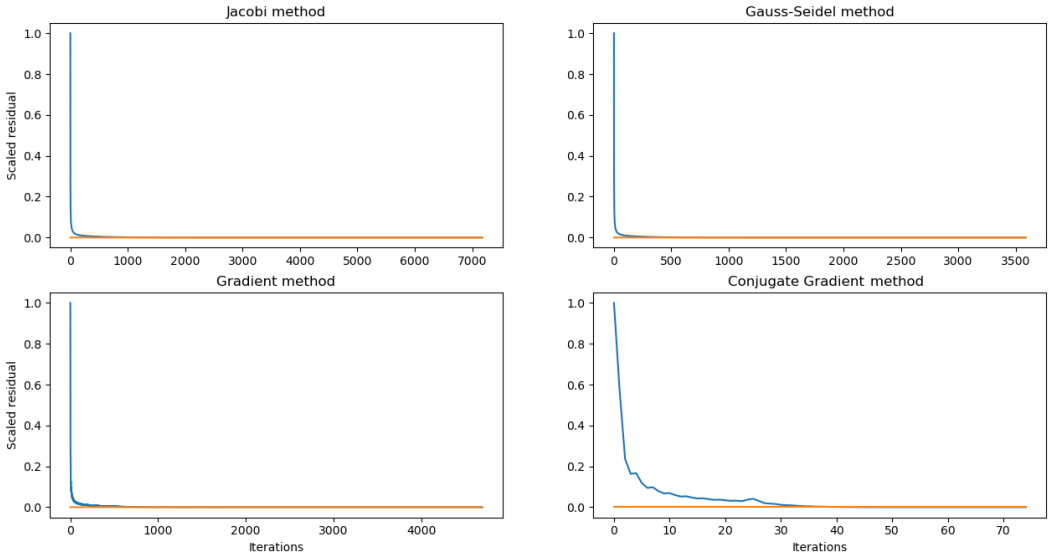


Figure 12: Comparison between the different methods in terms of the scaled residual and the number of iterations in *vem2.mtx* with tolerance equals to 10^{-10}

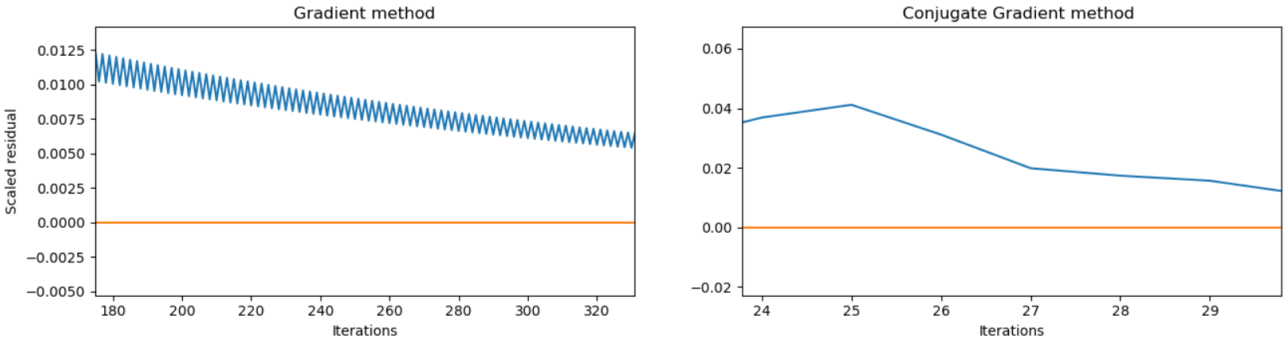


Figure 13: Zoomed view of the graphs in Figure 12 related to the Gradient and the Conjugate Gradient methods

4 Conclusion

This work has first presented in Section 2 the implementation of a library for solving systems of linear equations $Ax = b$, where A is a sparse matrix. In particular, the library contains the implementations of the following iterative methods: Jacobi method, Gauß-Seidel method, Gradient method and Conjugate Gradient method.
Then, this work has presented in Section 3 the validation study that has been conducted on the implemented library. This study revealed the strengths and weaknesses of the different methods and, in particular, this study has empirically demonstrated that there is no more efficient solver suitable for all possible systems of equations. In fact, from the results of the study, it has emerged that if the matrix does not have a particular pattern in its structure, the Jacobi and the Gauß-Seidel methods are faster in convergence than the Gradient and the Conjugate Gradient methods; instead, if the matrix presents the majority of the coefficients concentrated along its main diagonal and is very sparse, then the Gradient and the Conjugate Gradient methods perform well, while the Gauß-Seidel method is very inefficient. However, in both the cases, it is possible to conclude that the Jacobi method performs better than the Gauß-Seidel method and the Conjugate Gradient method performs better than the Gradient method.

References

[1] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[2] Travis Oliphant. *Guide to NumPy*. 01 2006.

[3] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[4] Steve Cook, Conrad Bock, Pete Rivett, Tom Rutt, Ed Seidewitz, Bran Selic, and Doug Tolbert. Unified modeling language (UML) version 2.5.1. Standard, Object Management Group (OMG), December 2017.

[5] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.