

Project 1 - bis

Iterative solvers of linear systems



Table of contents

Introduction and goal.....	3
Introduction.....	3
Linear systems and the problems of computers	4
Iterative methods and why	5
The library	7
Software architecture and the Object-oriented programming.....	7
Strategy Design Pattern.....	7
The architecture of the library	8
The languages and frameworks used	9
How the library works and some screenshots	11
The results	14
What we have done	14
Relative error.....	14
Time of execution	16
Number of iterations	18
Some considerations and conclusion	21

Introduction and goal

Introduction

The goal of this project is to implement a **library of iterative solvers for linear systems**. The solvers in question are:

- **Jacobi's** method,
- **Gaub-Seidel's** method,
- **Gradient** method,
- **Conjugate gradient** method.

To accomplish this, it was decided to create an **architecture structured** according to the **state-of-the-art paradigms** of software engineering, trying to ensure a high architectural standard. The algorithmic implementation of the methods, on the other hand, was studied at the "Methods of Scientific Computation" course at the Bicocca University of Milan. Several ready-to-use "collections of algorithms" for almost all programming languages are already available online, however, they have not been employed.

In making the library, we tried to find a compromise between **performance**, **readability** of the code and **extendibility** of the code to allow for future modifications. We will describe, next, the more technical aspects, however we start by highlighting some features, peculiarities, and functionality:

- The code takes the following parameters as **input**:
 - A **matrix A** constrained to be **symmetric** and **positive defined**.
 - A **symmetrical matrix** is a square matrix that coincides with its transpose; equivalently, a square matrix whose elements are symmetrical with respect to the main diagonal is defined as symmetrical,
 - A **positive definite matrix** is a symmetric matrix for which all its eigenvalues are strictly positive.

Important to point out is that, in our case, we set up a library to work with large **sparse matrices**. Sparse matrices are data structures that store only the nonzero elements of an array, saving memory compared to storing all elements. This aspect is considered when choosing libraries to support our implementation; in fact, some of them handle this aspect efficiently (e.g., Scipy).

- A **right member b**;
- An **exact solution vector x**,
- A **tol tolerance**, which is a measure of the maximum error or acceptable difference between a calculated or approximated value and the true value.
- The program **outputs**:
 - The **relative error** between the exact solution x and the approximate solution obtained from the code, i.e., a measure of the accuracy of an approximation with respect to the true value of a quantity or calculation,

$$e = \frac{||\bar{x} - x||}{||x||}$$

- The **execution time**, which is the period taken by a program to perform an operation,

- The **number of iterations**, or the number of times a process or algorithm is repeated to achieve a specific goal.
- Since, however, the library was created for **demonstration purposes**, some of the above inputs are **automatically generated** by the program. However, a simple extension (already prepared) of the library is possible and ensures that the user can manually enter all inputs. The **algorithm generate**:
 - The **exact solution** x , as a vector with all entries equal to 1,
 - The **vector** b is calculated as $b = A * x$.
- In addition, it should be pointed out that:
 - The **implemented methods stop** if the k -th iterate satisfies:

$$\frac{\|A * x^k - b\|}{\|b\|} < tol$$

Where “tol” is a tolerance chosen by the user and belonging to the following domain: $\{10^{-4}; 10^{-6}; 10^{-8}; 10^{-10}\}$

- The **implemented methods also stop** if they reach a number of iterations $k > 20000$, in which case they will not reach convergence,
- The implemented methods **start from the initial null vector** (all entries equal to zero), which under the appropriate assumptions of the A matrix does not affect convergence

For ease of use, moreover, the library is accompanied by a **graphical interface**, later described, that allows the insertion of the A matrix either by **.mtx file** or by **manual input**. This is **additional functionality** and does not, in any way, constrain future extensions or iteration via command line.

Linear systems and the problems of computers

A linear system is a fundamental concept in **linear algebra**. It refers to a set of linear equations involving a set of unknown variables. A **linear system** can be **represented in matrix form** as follows:

$$Ax = b$$

Where:

- **A** is a matrix of the coefficients,
- **x** is a vector of the unknown variables,
- **b** is a vector of the known terms.

Each equation in the linear system is a linear equation expressing a linear relationship between the unknown variables. The solution of a linear system is to find the values of the unknown variables x that simultaneously satisfy all the equations:

- If there is a unique solution, the linear system is called **determined**,
- If the system has no solution, it is called **impossible**,
- If the system has infinite solutions, it is considered **indeterminate**.

We will not go into the finality of a linear system, however, it is important to point out that today's **computers**, thanks to ad hoc algorithmic techniques, **are able to solve linear systems**, but they **must take into account**:

- Of the handling of **sparse matrices**,
- From the **approximation error** due to the computers' representation of numbers in "floating point" format,
- Of the **speed of execution**,
- ...

All these aspects were considered in implementing the library.

Iterative methods and why

Iterative methods for solving linear systems are algorithms that generate a **sequence of approximations** of the desired solution, **gradually converging to the exact solution**. Unlike direct methods, which provide an exact solution after a finite number of steps, iterative methods continue to improve the approximation of the solution until a desired convergence condition is reached.

In more detail, iterative methods start from an initial vector $x^{(0)} \in \mathbb{R}^n$ (of only entering with a value of 0 in our case), create a sequence of vectors $x^{(k)} \in \mathbb{R}^n$ and they get closer and closer to the exact solution x .

Some of the **most common iterative methods** for solving linear systems as well as used in this library are:

- **Stationary iterative methods.** These methods construct a succession of approximations that converge to the exact solution. During each iteration, the new approximation is computed using a formula involving the previous approximation and a series of calculations that do not depend on the number of the current iteration.
 - **Jacobi's method**
In this method, the linear system $Ax = b$ is rewritten as a series of equations involving only a single unknown at a time. Then, for each equation, the value of the corresponding variable is approximated using the current values of the other variables.
 - **Gauss-Seidel's method**
This method is a variation of Jacobi's method, in which the newly computed approximations during the current iteration are used. In other words, while computing the approximation of one variable, the already updated values of the other variables are used.
- **Non-stationary iterative methods.** These methods construct a succession of approximations that may vary in each iteration. During each iteration, the new approximation is calculated using a formula that may depend on the number of the current iteration. Nonstationary iterative methods are often more complex than stationary methods, but they can be more efficient in terms of convergence and speed.
 - **Gradient method**
This method starts from an initial solution and tries to reduce the residual error between the matrix-vector product Ax and the vector of known terms b . The main idea is to follow the direction opposite to the residual error gradient to approach the optimal solution.
 - **Conjugate Gradient method**

This method is widely used for solving scattered and symmetric positive definite linear systems. It is based on the idea of finding the solution by following an optimal direction in the subspace of solutions. The algorithm converges in a finite number of iterations if the matrix of coefficients is symmetric positive definite.

Iterative methods offer several advantages, such as the ability to handle large linear systems and the ability to stop iteration at any time to obtain an acceptable approximation of the solution. However, they can require a significant number of iterations to converge and can be sensitive to the choice of parameters or the structure of the linear system.

The **main reason we use iterative methods** is that in the reality the **matrices are usually large and sparse**. We know that sparse matrices are handled by today's calculators in a space-saving way (via triples). Here, **iterative methods tend to preserve the sparse nature of matrices** by exploiting basic sum and product operations among them, without any decomposition, **avoiding the "fill-in" phenomenon**.

The library

Software architecture and the Object-oriented programming

When we talk about "**architecture**" in the software industry, we are referring to an ill-defined notion that covers the **most important aspects of the internal design** of a software system: the fundamental structures, their relationships, and how they interact. Software architecture programming knowledge to plan the project at a high level so that details can be added later, allowing software teams to lay out the big picture and begin preparing a prototype.

One cannot, however, talk about contemporary software architecture without making an introduction to **Object-Oriented Architecture** (OOA), a design paradigm based on dividing the responsibility for an application or system into individual reusable, self-sufficient objects. The popular object-oriented design approach is to view a software system as a collection of entities known as objects, components containing encapsulated data, and procedures grouped to represent those entities. This way of designing architecture is close to the real world and its applications because the state and behavior of objects are almost identical to those of the world around us.

The use of OOA is **used in our library** to ensure a **well-structured architecture** (described below), guaranteeing **modularity, complexity reduction, efficiency, maintainability, reusability**, ... all of which is then reinforced using a particular design pattern explained below.

This paragraph is also repeated in the "Project 2" report to make it complete.

Strategy Design Pattern

Designing object-oriented software is not an easy job, and trying to make it reusable and flexible is even more difficult; attention must be paid to the level of granularity when creating objects and their relationships to each other.

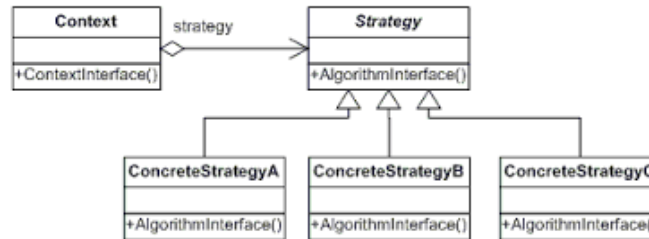
A way to keep the quality of the software architecture high is to take advantage of **design patterns**, which originated from the discussion in a 1994 Gang of Four (GoF) book entitled "Design Patterns - Elements of Reusable Object-Oriented Software". They are a general, reusable solution to common problems within certain scenarios in software design: not a solution that can be used directly in code, but adaptable to use cases. Design patterns can **speed up the development process** by providing tested and proven paradigms: effective design requires consideration of problems that may not be visible until later in the implementation; using design patterns helps in this regard. Last but not least, making use of these patterns promotes **code readability** for programmers and architects familiar with them.

In the architecture of our library, we used design patterns, in particular the design pattern Strategy, we will soon understand why.

Strategy pattern is a design pattern that allows the isolation of an algorithm within an object, in a way that is useful in those situations where it is necessary to **dynamically change the algorithms used by an application**. This pattern requires that **algorithms be interchangeable** with each other, based on a specified condition, in a manner that is transparent to the client using them. In other words, given a family of algorithms that implements a certain functionality, they must always export the same interface, so the "client"

of the algorithm does not have to make any assumptions about which strategy is instantiated at a particular instant. By extension it is, therefore, possible to implement variable program and object behavior even while the software is running.

A **generic implementation** of the design pattern strategy is as follows:



But why was design pattern strategy employed? The **iterative algorithms** we implemented all have a common code part described by a **generic procedure** structured as follows:

Input: $A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n, \text{tol};$
 Output: $x \in \mathbb{R}^n;$

```

1: inizializzo  $x^{(0)} \in \mathbb{R}^n;$ 
2:  $k = 0;$ 
3: while CRITERIO DI ARRESTO( $x^{(k)}$ ) do
4:    $x^{(k+1)} = \text{UPDATE}(x^{(k)});$ 
5:    $k = k + 1;$ 
6:   if  $k > \text{maxIter}$  then
7:     errore non converge
8:     break;
9:   end if
10: end while
11:  $x = x^{(k)};$ 
  
```

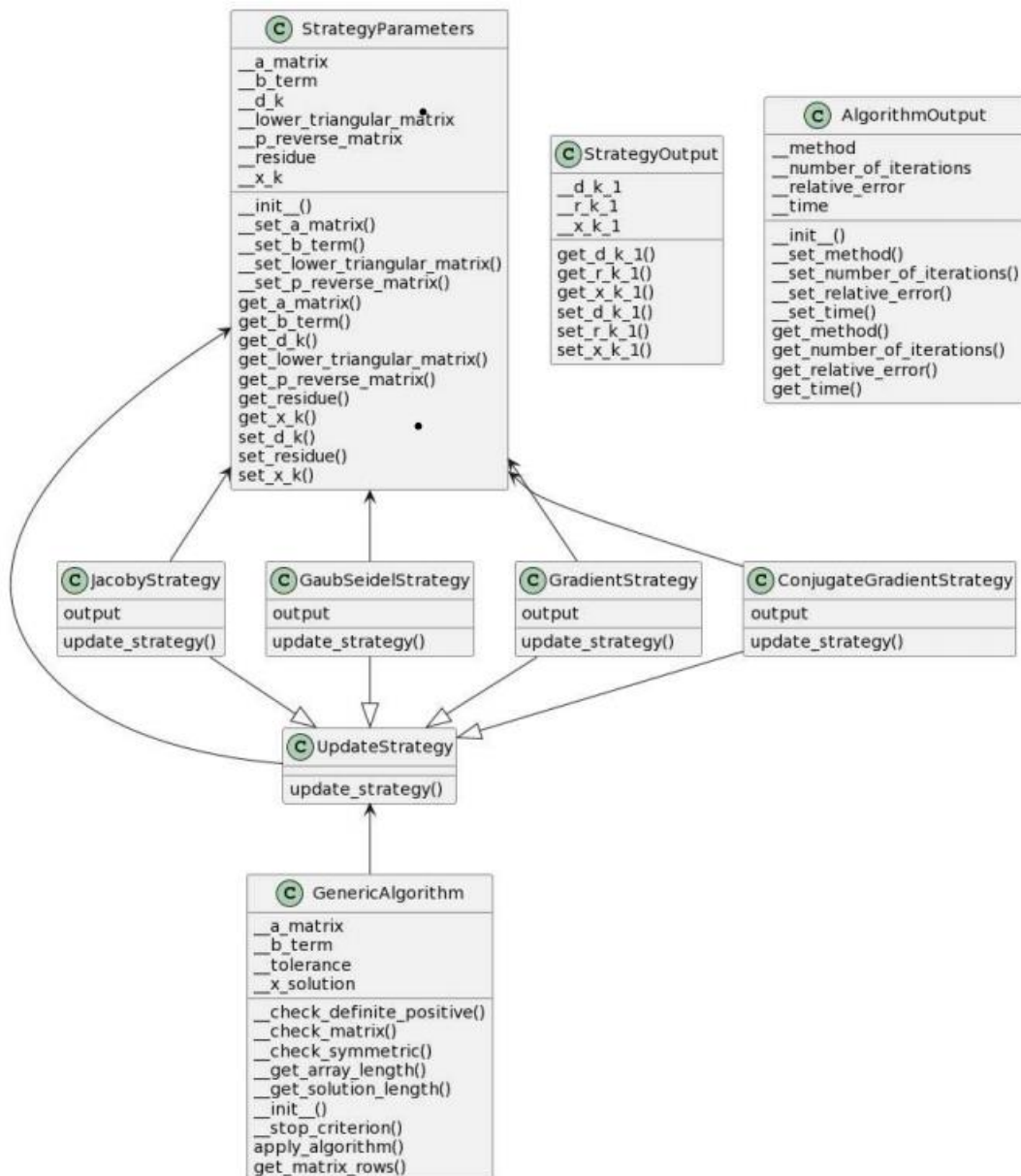
What **they differ** in is the single iterated, particularly the "**upDate**" method. This is where the Design Pattern Strategy comes in, **allowing dynamic modification of the "strategy."** Thus, our library will have one abstract strategy "**UpdateStrategy**" and **four concrete implementations**, one for each iterative method ("JacobyStrategy," "GaubSeidelStrategy," "GradientStrategy," "ConjugateGradientStrategy"), each with its own peculiarities. All will be **transparent** and changes to one of them will not compromise the others. This aspect allows, in addition to the advantages described above, to easily **extend our code in the future** to consider new iterative methods.

We could add much more than the pattern strategy, however, let us move on to the visualization of our architecture.

The architecture of the library

After describing more purely theoretical notions and motivating why some implementation choices were made, we go on to visualize in **diagram form our architecture**.

To **describe it** we use the **Class Diagram**, which is a structural UML diagram that describes **entities** (classes and interfaces), with **their characteristics and any relationships**. The conceptual tools used are the class concept of the object-oriented paradigm. It is important to highlight that this diagram shows classes at a "high precision" and is, therefore, suitable for the presentation of detailed software architectures.



We will not dwell, any more than we already have on describing the architecture; however, it can be seen from the diagram how **object-oriented programming is exploited**. Furthermore, it is easily evident how the pattern strategy for the "**UpdateStrategy**" was implemented with the four concrete implementations: "JacobyStrategy," "GaubSeidelStrategy," "GradientStrategy," "ConjugateGradientStrategy." In addition, just to take full advantage of OOP and the chosen pattern, **even the output data/method parameters themselves are objects**, which makes everything easily extensible, clear and efficient.

No code, other than explanatory pseudocode, will be reported in this specific report because the algorithms are standard, while the core is the architecture and analysis. This is to give more consistency to this report.

The languages and frameworks used

Having reached this point, let us go on to analyze the **languages and libraries/frameworks used** to build our library.

The **main one is Python**, a high-level, interpreted, **object-oriented**, and easy-to-learn programming language. Python is used in several areas, including web application development, data analysis, artificial intelligence, task automation, machine learning, and more.



This language stands out in the **scientific domain** for, as highlighted earlier, the **vast number of scientific libraries that accompany it**, which is why it was selected for this project. In our case we used the following:

- **NumPy**
It is a very popular Python library for scientific and numerical computation, as it efficiently handles the processing of numerical data and multidimensional arrays, in addition to the fact that it is supported by a large community of developers and a consequent wide compatibility.
- **Scipy**
It is an open-source library for Python that provides advanced functionality for scientific and engineering computation. It is an extension of NumPy and offers a wide range of specialized functions for math, science, and engineering problems. This library was selected because, unlike Numpy, it allows us to work with sparse matrices (which we had as input).
- **Time**
It is a standard Python library that provides functions for working with time and timing. This library was chosen to compute the timing that our algorithm needs to output.
- **Matplotlib**
It is a very popular and widely used data visualization library in Python. It provides a wide range of features for creating graphs and data visualizations easily and effectively. Matplotlib was used to create the graphs for analysis purposes for this report. The ability to generate plots has been "disabled" on the code side for the reasons after explained, but it is easily reactivated by following the steps.



In addition, as highlighted earlier, our library provides a **graphical user interface** to support the user (additional functionality that does not preclude the ability to use the command line). It has been developed around **three main languages**:

- **HTML**
It is the standard markup language for documents viewable by a web browser.
- **CSS**
It is a style sheet language used to describe the presentation of a document based on a markup language such as HTML.
- **Javascript**
It is a high-level, object-oriented, interpreted programming language commonly used for client-side web application development in users' browsers.



The following frameworks were used as **support** for these three languages:

- **Bootstrap**

It is an open-source front-end framework widely used for the development of Web sites and Web applications. It provides a collection of tools, components and predefined styles that enable developers to create modern, responsive and well-designed user interfaces efficiently.

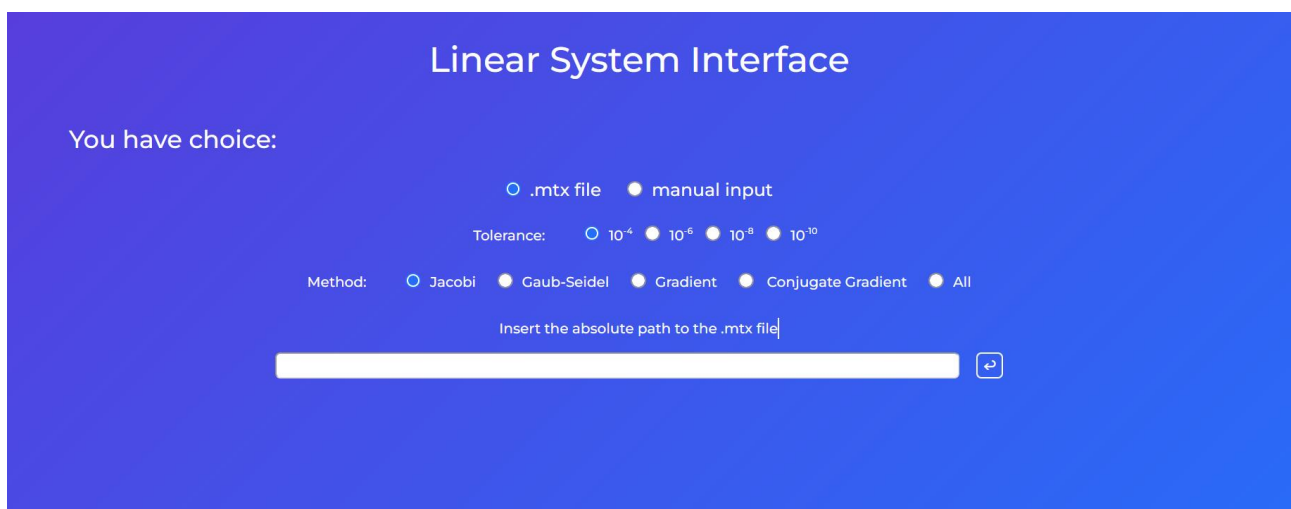


Finally, a library called **eel** was used to **connect front-end (the graphical part) and back-end (the actual library)**, with the goal of creating a simple, browser-executable desktop application. **Eel** is a Python library that enables the creation of desktop user interfaces using Web technologies such as HTML, CSS and JavaScript. It is designed to simplify the development of desktop applications using Python as the backend language and web technologies for the frontend part of the application.

This paragraph is also repeated in the "Project 2" report to make it complete.

How the library works and some screenshots

Having reached this point, we move on to **briefly analyze the functionality of the library** (not all, many have already been discussed: see previous paragraphs), as well as the **GUI** made available to the user, a GUI that abstracts and simplifies internal functions. Written in the languages explained earlier, the library, also through the interface allows the following operations.



Linear System Interface

You have choice:

☐ .mtx file ☒ manual input

Tolerance: ☒ 10^{-4} ☐ 10^{-6} ☐ 10^{-8} ☐ 10^{-10}

Method: ☒ Jacobi ☐ Gaub-Seidel ☐ Gradient ☐ Conjugate Gradient ☐ All

Insert the matrix dimension


x

The **main interface** looks like this. The user can choose:

- Whether to use a **".mtx" file** to load the matrix or **manual input**.
 - If the choice falls on **input from file**, the user must provide the absolute path to the file,
 - **Otherwise**, the user is free to choose the size of the $n \times n$ matrix and manually enter values into the cells (n/d format is not supported by the controls),
- Select a desired **tolerance level** from the available options (10^{-4} , 10^{-6} , 10^{-8} , 10^{-10}),
- Decide which **resolution method** to apply (Jacobi, Gaub-Seidel, Gradient, Conjugate Gradient) (1 at a time or all at once).

Once the data submission is made, a **check is made on the constraints imposed** by the problem (positive definite matrix, etc.), and if all is successful, the system processes the matrix and provides the results as follows.

Linear System Interface

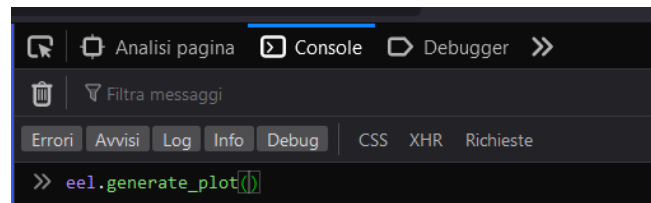
These are the results: 

Method	Relative error	Time	Number of iterations
Jacoby	0.004968461406195917	0.09418153762817383	1927
GaubSeidel	0.004951189291546085	11.367596864700317	965
Gradient	0.0038119295293938373	0.09427571296691895	1308
ConjugateGradient	5.729017222237361e-05	0.0	47

As we can see, the system shows a **new interface all the output** data required by the project specifications, in accordance with the choices made on the main screen. Clicking on the "home" button allows you to return to the main screen.

It is good to remember that the **interface is an additional function** and does not bind its use.

In addition, the library has an **additional feature**, which is by default **disabled for efficiency issues** as well as because it is a debugging tool. This feature is the **generation of automatic plots** based on input matrices-tolerances. However, for the reason just stated, it is necessary to re-enable the function from code, and once done by executing the instructions given in the code itself, simply interact from the browser console and type "eel.generate_plot()" to start the process.



The results

What we have done

Having reached this point, we can devote ourselves to **verifying** the operation of our library, as well as to **analyzing some of its results** (on tests) and drawing information about the implementations of the algorithms themselves.

Four sparse matrices were used to perform the **tests**:

- *Spa1.mtx*, 1000*1000 matrix with 182434 values (18% of the matrix),
- *Spa2.mtx*, 3000*3000 matrix with 1633298 values (18% of the matrix),
- *Vem1.mtx*, 1681*1681 matrix with 13385 values (0.5% of the matrix),
- *Vem2.mtx*, matrix 2061*2061 with 21225 values (0.5% of the matrix).

The matrices of the "Vem*" type are the most scattered, with 99.5% of the entries having no value.

We will **analyze the results**, as noted above, in terms of **three metrics**:

- **Relative error** between the exact solution x and the approximate solution,
- **Number of iterations** to reach convergence,
- **Computational time**.

To simplify interpretation, moreover, we have **grouped the results into plots** according to the metric and tolerance chosen. Each plot shows, for each matrix, the performance of each iterative method.

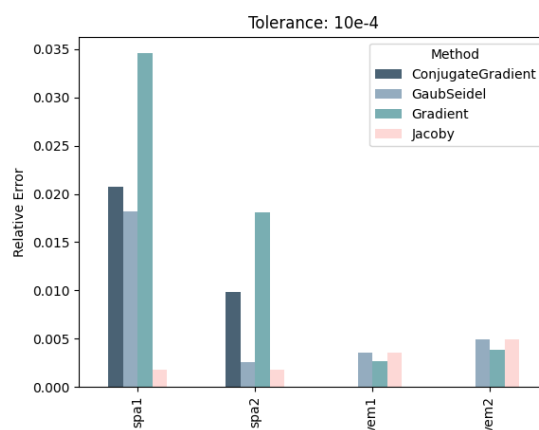
We proceed, then, to analyze, metric by metric, the results.

Relative error

Let us start with the metric of relative error. **Relative error** is a **measure of the accuracy** of an approximation to the true value x of a quantity or calculation.

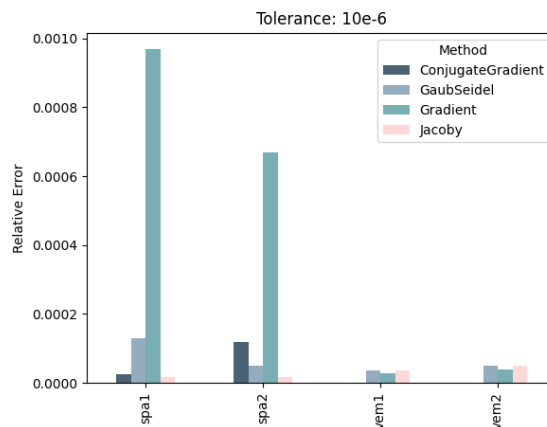
$$e = \frac{||\bar{x} - x||}{||x||}$$

By analyzing the relative error we can draw some conclusions about our models.



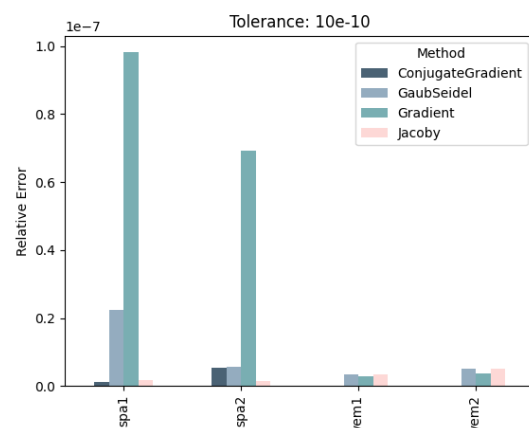
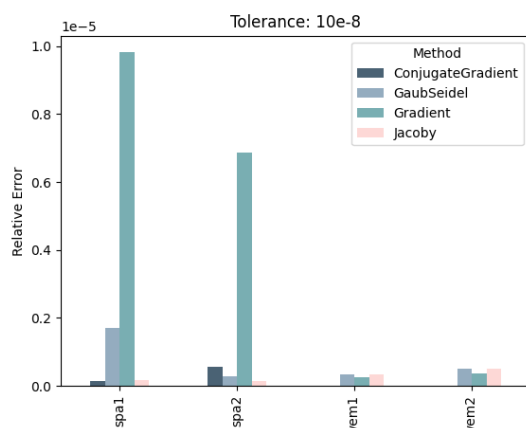
Method	ConjugateGradient	GaubSeidel	Gradient	Jacoby
Matrix				
spa1	0.020790	0.018206	0.034575	0.001771
spa2	0.009821	0.002599	0.018130	0.001766
vem1	0.000041	0.003507	0.002705	0.003540
vem2	0.000057	0.004951	0.003812	0.004968

Looking at this calculated error on the resolution of linear systems, giving the test matrices as input and with a tolerance of 10^{-4} we can see that, generally, with "vem*" matrices the behavior of the various methods is comparable. In contrast, for "spa*" type matrices we can observe that the highest relative error is given by the resolution by "Gradient method." Let us see if this behavior is confirmed by decreasing the tolerance.



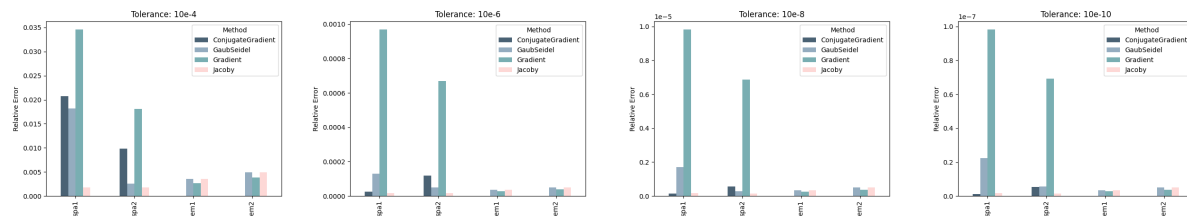
Method	ConjugateGradient	GaubSeidel	Gradient	Jacoby
Matrix				
spa1	2.552909e-05	0.000130	0.000968	0.000018
spa2	1.197985e-04	0.000051	0.000669	0.000017
vem1	3.732340e-07	0.000035	0.000027	0.000035
vem2	4.742996e-07	0.000049	0.000038	0.000050

By decreasing the tolerance ($\text{tol} = 10^{-6}$) we can see how the aspect just described is accentuated: while we have those matrices of the "spa*" type with approximately equal relative error, with "vem*" matrices **the relative error is evidently greater in solving by "Gradient method."** In any case, as we might have expected, as the **tolerance decreases it decreases**: the approximate solution comes closer in relative terms to the optimal solution.



Method	ConjugateGradient	GaubSeidel	Gradient	Jacoby	Method	ConjugateGradient	GaubSeidel	Gradient	Jacoby
Matrix					Matrix				
spa1	1.319841e-07	1.789733e-06	9.816364e-06	1.824979e-07	spa1	1.284618e-09	2.248088e-08	9.820388e-08	1.852437e-09
spa2	5.586661e-07	2.794322e-07	6.865240e-06	1.572870e-07	spa2	5.324230e-09	5.570741e-09	6.937815e-08	1.484272e-09
vem1	2.831873e-09	3.517457e-07	2.695337e-07	3.539766e-07	vem1	2.191751e-11	3.508242e-09	2.713167e-09	3.539459e-09
vem2	4.299984e-09	4.958370e-07	3.809851e-07	4.965608e-07	vem2	2.247627e-11	4.948913e-09	3.798768e-09	4.964185e-09

By decreasing the tolerance more and more ($\text{tol} = \{10^{-8}, 10^{-10}\}$), the underlined behaviors are confirmed, and the methods tend to have a **smaller error with the "vem*" matrices**. We can, at this point, decrease the size of the graphs to capture some "global" aspects.



With a global view of the plots, we can confirm how the general behavior of the methods, on the matrices, remains unchanged with respect to the different tolerances, or, at most, the various differences between the methods are, at most, accentuated even more as they decrease.

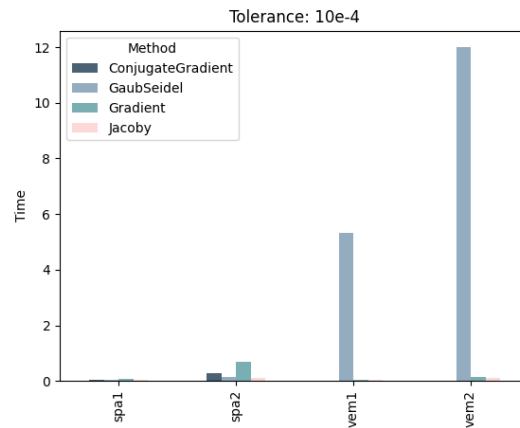
In general, **greater tolerance leads to greater accuracy of our resolution**, and if we were to decree a "winner" among the methods, we could see that, in general, **"Jacobi's method"** is the one that has given us results with **less relative error**, and, gradually, the **"Conjugate gradient method"** and **"Gaub Siedel"** have also aligned with it.

Let us see, at this point, if we can draw similar considerations for the other metrics as well.

Time of execution

All tests were conducted on a DELL XPS 15 7950H with an Intel Core i7-9750H, 16GB RAM and 512GB SSD Nvme. This is to have a reference for the times.

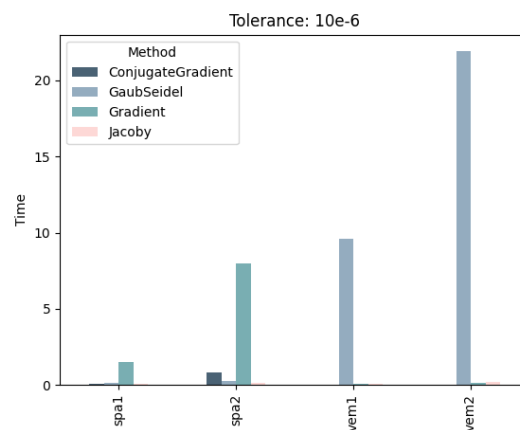
Execution time refers to the amount of time it takes a program to perform a task. It is a measure of **how quickly a task is completed**. The execution time depends on the complexity of the task itself, as well as on the implementation of the algorithms, system resources, and other factors such as processor speed, the amount of memory available, and system workload. Since the iterative algorithms we are studying were all executed on the **same computer sequentially** (negligible time lapse), at a time when, therefore, the computer was under the same workload, we will be able to draw conclusions about the **complexity of the operation**, the **input** (the matrix) and the **implementation**.



Method	ConjugateGradient	GaubSeidel	Gradient	Jacoby
Matrix				
spa1	0.036032	0.065059	0.068816	0.032007
spa2	0.296875	0.168098	0.728054	0.112061
vem1	0.004136	6.139200	0.053967	0.056347
vem2	0.005543	13.249240	0.095890	0.104491

Looking at the algorithmic time computed on the resolution of linear systems, giving the test matrices as input and with a tolerance of 10^{-4} we can see that, generally, in the presence of the "spa*" matrices the behavior of the various methods is comparable. In contrast, for "vem*" type metrics we can see there is a clear **difference** between the resolution by **"Gaub Seidel's method"** and the others.

Let us see if this result is confirmed by decreasing the tolerance.

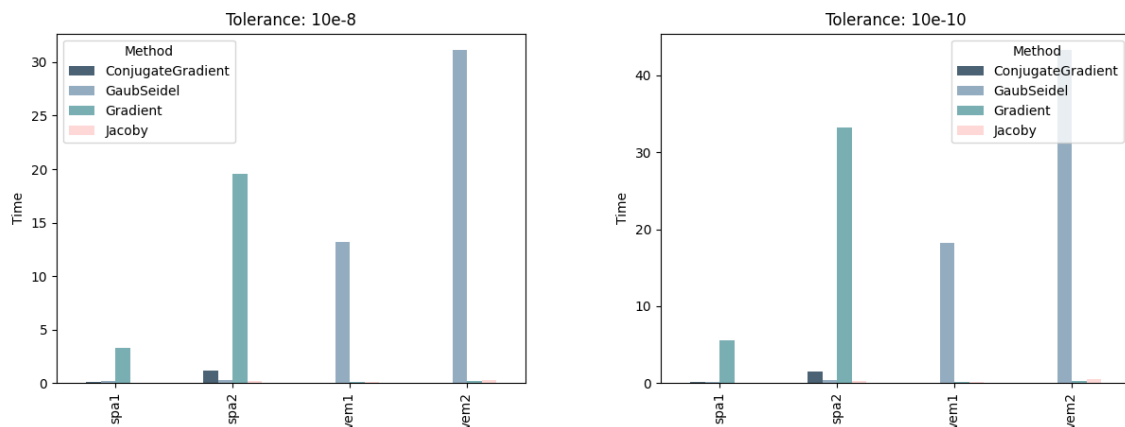


Method	ConjugateGradient	GaubSeidel	Gradient	Jacoby
Matrix				
spa1	0.091743	0.120603	1.633113	0.046016
spa2	0.833242	0.246384	8.707072	0.157604
vem1	0.004278	10.349263	0.094315	0.102537
vem2	0.006271	23.784882	0.166561	0.180515

By decreasing the tolerance ($\text{tol} = 10^{-6}$) we can see that the described aspect remains valid for the "vem*" matrices, however on the less sparse "spa*" matrices, even the **"Gradient method"**, at this point, begins to stand out **negatively from the others**.

In any case, as we might have expected, as the **tolerance decreases**, the **running time increases**: the **approximate solution** approaches, in relative terms, **more closely to the**

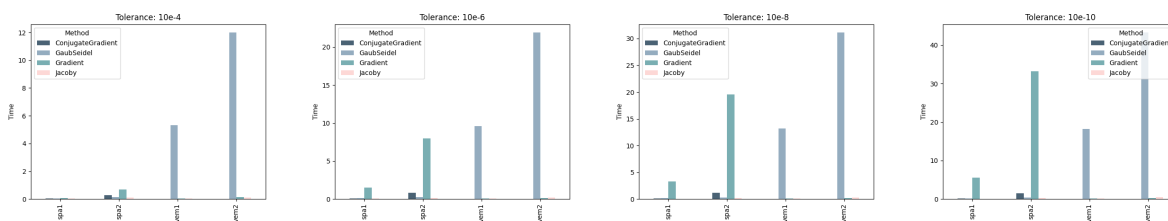
optimal solution (from what we had concluded earlier), but the computation time required increases.



Method	ConjugateGradient	GaubSeidel	Gradient	Jacoby
Matrix				
spa1	0.113159	0.157145	3.677490	0.055749
spa2	1.209870	0.321575	20.883539	0.189067
vem1	0.005010	15.046038	0.133831	0.153770
vem2	0.007911	35.049228	0.249537	0.280703

Method	ConjugateGradient	GaubSeidel	Gradient	Jacoby
Matrix				
spa1	0.125141	0.191786	5.297836	0.071060
spa2	1.487824	0.400931	33.857008	0.237067
vem1	0.006502	19.742322	0.172713	0.192901
vem2	0.008453	46.329669	0.323882	0.360631

By decreasing the tolerance more and more ($\text{tol} = \{10^{-8}, 10^{-10}\}$), the underlined behaviors are confirmed. We can, at this point, decrease the size of the graphs to capture some "global" aspects.



With a global view of the plots, we can confirm how **the general behavior** of the methods, on the matrices, **remains unchanged** with respect to the different tolerances.

In general, **more tolerance leads to more computation time**, and if we were to decree some "winners," we could see that, in general, the **"Jacobi method"** and the **"Conjugate Gradient method"** are the ones that lead us to the approximate solution in less time. By drawing a parallel with the result obtained in the paragraph regarding relative error, we can further confirm the **validity of these methods**, as the solution they provide us with is, moreover, "close" in relative terms to the optimal one.

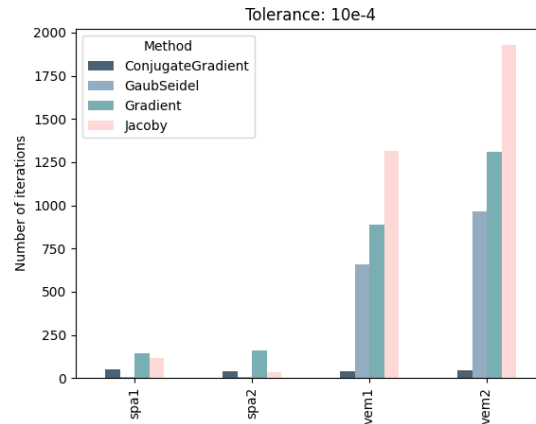
Finally, let us see if we can draw other useful considerations by analyzing the last metric: the number of iterations.

Number of iterations

The **number of iterations** refers to the **number of times** a process or algorithm is repeated to achieve a specific goal. Iterations are commonly used in iterative algorithms, in which a given **operation is performed repeatedly** until a termination condition is met. The number of

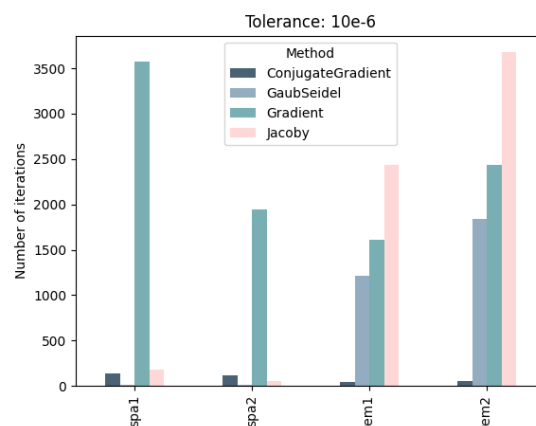
iterations required **depends** on the nature of the problem, the algorithm used and the input data.

The number of iterations can vary widely depending on the problem. Some algorithms may require only a few iterations to converge to an acceptable solution, while others may require a large number of iterations to achieve the desired accuracy.



Method	ConjugateGradient	GaubSeidel	Gradient	Jacoby
Matrix				
spa1	49	9	143	115
spa2	42	5	161	36
vem1	38	659	890	1314
vem2	47	965	1308	1927

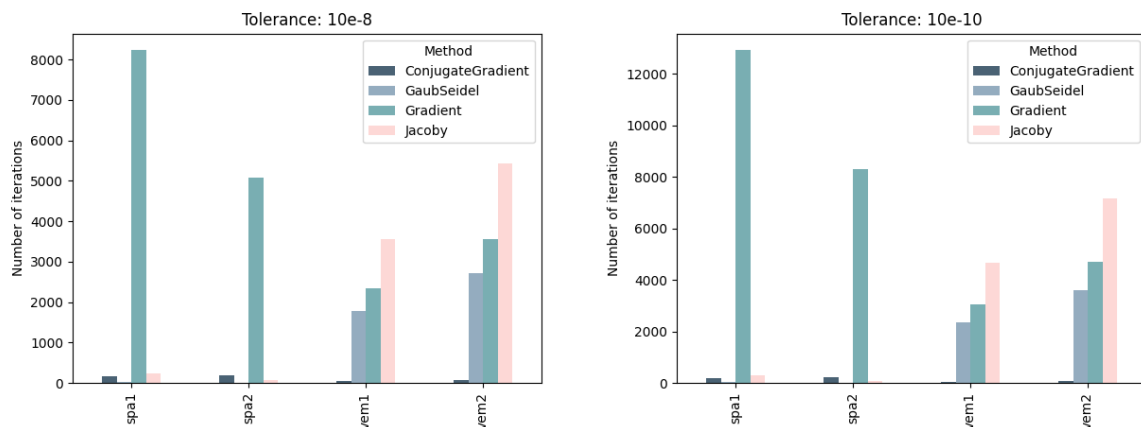
Looking at the number of iterations computed on the resolution of linear systems, giving the test matrices as input and with a tolerance of 10^{-4} we can see that, generally, in the presence of the "spa*" matrices the behavior of the various methods is comparable. In contrast, for "vem*" type metrics we can see that methods such as **"Jacoby" take many more iterations to converge**. Let us see if this turns out to be confirmed by decreasing the tolerance.



Method	ConjugateGradient	GaubSeidel	Gradient	Jacoby
Matrix				
spa1	134	17	3577	181
spa2	122	8	1949	57
vem1	45	1218	1612	2433
vem2	56	1840	2438	3676

By decreasing the tolerance ($\text{tol} = 10^{-6}$) we can see that the **"Gradient method,"** at this point, requires a **large number of iterations for any type of matrix**, beginning to stand out, in the negative, from the others. Nevertheless, for the less sparse "spa*" matrices, all solvers, apart from the one just mentioned, achieve very good results. Conversely, for "vem*" matrices, only the **"Conjugate Gradient Method" stands out positively.**

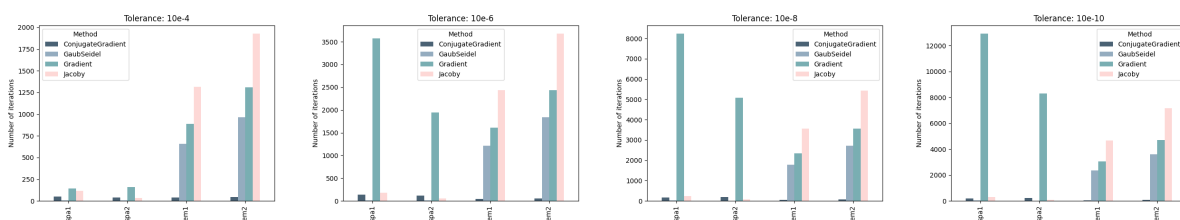
In any case, as we might have expected, as the **tolerance decreases**, the **number of iterations to reach convergence increases**: the approximate solution comes closer, in relative terms, to the optimal solution (from what we concluded earlier), but the computation time required increases, as do the iterations.



Method	ConjugateGradient	GaubSeidel	Gradient	Jacoby
Matrix				
spa1	177	24	8233	247
spa2	196	12	5087	78
vem1	53	1778	2336	3552
vem2	66	2714	3566	5425

Method	ConjugateGradient	GaubSeidel	Gradient	Jacoby
Matrix				
spa1	200	31	12919	313
spa2	240	15	8285	99
vem1	59	2338	3058	4671
vem2	74	3589	4696	7174

As we decrease the tolerance more and more ($\text{tol} = \{10^{-8}, 10^{-10}\}$), the behaviors emphasized by the 10^{-6} tolerance are confirmed: for the "spa*" matrices all the methods convention "fast" except the "Gradient method," while for the "vem*" matrices the **"Conjugate Gradient method" stands out**. We can, at this point, decrease the size of the graphs to capture some "global" aspects.



With a global view of the plots, we can confirm the behavior described above.

In general, **more tolerance leads to computation time and more iterations for convergence**. If we were to decree a "winner," in this case it would be, without a shadow of a doubt, the **"Conjugate Gradient method."** Drawing a parallel with the result obtained in the previous paragraphs, we can see that, contrary to what one might expect, **the computation time**, although it is related to the **number of iterations**, carries with it differences, this is because the single iteration can be more or less onerous. Pe example, Jacoby's method, although it requires a large number of iterations, especially in the presence of highly sparse matrices, has

stood out positively in the study related to computational time, as its single iteration is not very onerous.

We proceed, therefore, by concluding our analysis and grouping the considerations.

Some considerations and conclusion

We conclude our study with some **final considerations**. First, it should be pointed out that the considerations we can draw are given from an analysis on four test matrices, thus **not a statistically significant sample**. However, we can draw parallels with proven **theoretical concepts** to substantiate our argument. In addition, it is important to remember that our study has as a constraint the use of **sparse matrices**, which, however, as we know, are the "real" matrices that computers find themselves, every day, saving and computing.

We start by pointing out that the analysis showed, as we expected, that asking for **greater accuracy**, by decreasing the tolerance, to our iterative methods, go **increase the iteration steps** required for convergence and the associated computation time. However, we could see that the latter two concepts are not always closely related; in fact, the **computational time depends on the complexity of the individual iteration**: the "Jacobi method" required many iterations to reach convergence; however, the time complexity of it was not significant. In general, all methods arrived at convergence in less than the maximum 20000 iterations set.

The "**Conjugate Gradient method**" is the iterative solver that **stood out** with respect to all three performances ("number of iterations, "relative error," and "computation time"). This aspect was predictable at the theoretical level: in fact, compared to the other methods, it is the **only one that guarantees convergence in, at most, n iterations**, where $n \times n$ is the size of the matrix. This method is particularly advantageous for large and sparse matrices; however, **convergence is only guaranteed for symmetric and positive definite** matrices (ours as input). In contrast, "the Gradient method," may require a large number of iterations to converge if the matrix is not well conditioned or has widely varying eigenvalues.

"**Jacobi's method**" is also a **good method**, simple to implement and efficient for sparse matrices, however, its convergence can be slow for ill-conditioned matrices although, its being **computationally inexpensive**, plays to its advantage. The "**Gauss-Seidel method**," on the other hand, has faster convergence than Jacobi because it uses newer information during the iterative process; it works well with diagonally dominant or symmetrically defined positive matrices, but can prove **computationally expensive**.

In general, the "**Conjugate Gradient method**" is **often considered the best** of the four methods for positive definite symmetric linear systems because it converges in a limited number of iterations. However, if the coefficient matrix does not satisfy this property, the conjugate gradient method is not appropriate.

For **non-symmetric or positive undefined linear systems**, the **choice between "Jacobi" and "Gauss-Seidel"** depends on the specific characteristics of the matrix. "Jacobi" may be a reasonable choice because of its simplicity and computationally efficient, while "Gauss-Seidel" offers faster convergence by taking advantage of newly computed approximations.

Ultimately, the choice of method depends on the characteristics of the linear system, such as the structure of the coefficient matrix, the size of the system, and specific goals, such as speed of

convergence and computational complexity. In particular, we have shown empirically that **there is no more efficient and suitable solver for all possible systems of equations.**

I really care about the success of this project, so if you have any problems, please do not hesitate to contact me m.piazzalunga2@campus.unimib.it

--- THANK'S ---

Mattia **Piazzalunga**, project 1-bis of “Methods of **Scientific Computing** project”