Algorithms that Changed The World

Oskar Aaron Oramus

December 2022

Abstract

Systems of equations are a powerful tool for representing and solving problems involving multiple equations and variables. These equations are used to model real-world situations. They can determine the motion of a particle in physics, and analyze complex systems that describe the behaviour of mechanical systems in engineering. In economics they are used to model the behaviour of the market, equations can describe the supply and demand, and then solve for the equilibrium price and quantity. Computer science uses them to solve optimisation problems given a set of constraints and an objective function. This paper aims to explore different ways of solving them.

I certify that all material in this report which is not my own work has been identified.

1 Introduction

The simplest mathematical equation is in the form

$$ax + b = 0 \tag{1}$$

where: a, b = constants $\in \mathbb{R}$
x = variable $\in \mathbb{R}$

It provides the fundamentals for modelling the real world. Many more complex mathematical concepts and techniques are built upon the foundation of linear equations. Many numerical methods for solving non-linear equations involve approximating them using linear equations which can be easily solved.

Building on the idea of the linear equation, we can use a system of linear equations to solve more complex problems.

$$ax + b = 0 \tag{2}$$

$$cx + d = 0 \tag{3}$$

To solve for equation 2 and 3 we can use a simple method where we make one of the coefficients equal, then we cancel them out when subtracting.

Given the problem:

$$ax + by = c \tag{4}$$

$$dx + ey = f \tag{5}$$

We can get rid of the y coefficient by making them equal. We multiply eq. 4 by e and eq. 5 by b.

$$aex + bey = ce \tag{6}$$

$$bdx + bey = bf \tag{7}$$

Subtract them.

$$aex - bdx = ce - bf \tag{8}$$

Solve for x.

$$x = \frac{ce - bf}{ae - bd} \tag{9}$$

Repeating the same for y, we end up with general solutions.

$$y = \frac{af - cd}{ae - bd} \tag{10}$$

We can see that the denominator for both equations is $ae - bd$. This is quite significant as if we put all of our coefficients into a matrix:

$$\begin{pmatrix} a & b \\ d & e \end{pmatrix} \tag{11}$$

If we calculate the determinant of the matrix, it is also equal to $ae - bd$. We also know that when a matrix has a determinant of 0, it has no inverse, which in our case means no solutions. This is also emphasised by the equations 9 and 10 having division by zero when that happens. We can see from equations 4 and 5 that having $a = 0$ and $c = 0$ or $b = 0$ and $e = 0$ we get rid of x or y coefficient respectively. Then we end up with two equations stating different relationships and causing a contradiction (no solutions).

Solving for two variables is quite trivial, but when the number of variables grows, things become more complicated. We can compact the notation by expressing the equations in a matrix form.

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n + b_1 = 0 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n + b_2 = 0 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n + b_m = 0 \end{cases} \tag{12}$$

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{22} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} \tag{13}$$

Then our final equation becomes [13]:

$$Ax = b \tag{14}$$

2 Solving Systems of Equations

When our matrix size is small, the problem can be solved by rearranging. Given the equation

$$Ax = b \tag{15}$$

We can solve for x by multiplying by A^{-1} [8].

$$\begin{aligned} A^{-1}Ax &= A^{-1}b \\ x &= A^{-1}b \end{aligned} \tag{16}$$

"Winograd originally proved that matrix multiplication is no harder than matrix inversion, and the converse is due to Aho, Hopcroft and Ullman" [2, p. 671]. This means we can look at matrix multiplication algorithms to determine the inverse complexity. A naive inversion algorithm is of complexity $O(n^3)$, and Strassen algorithm is $O(n^{log_2(7)})$ [2, p. 671]. This does not scale well, so we need a better way of solving systems of equations.

2.1 Jacobi Method

The Jacobi method is an iterative algorithm for solving systems of equations.

The equation is decomposed into three components:

- D - Diagonal matrix
- L - Lower triangular matrix
- U - Upper triangular matrix

$$A = D + L + U \tag{17}$$

Our iterative solution becomes [4]

$$x^{k+1} = D^{-1}(b - (L + U)x^k) \tag{18}$$

We still have an inverse to compute, however, it's much easier as the inverse of a diagonal matrix is equal to reciprocals of each element [3].

2.1.1 Why it works

Given the equation system we would like to solve:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned} \tag{19}$$

We re-write these equations with respect to x [6].

$$\begin{aligned} x_1 &= \frac{b_1 - a_{12}x_2 - a_{13}x_3}{a_{11}} \\ x_2 &= \frac{b_1 - a_{21}x_1 - a_{23}x_3}{a_{22}} \\ x_3 &= \frac{b_1 - a_{31}x_1 - a_{32}x_2}{a_{33}} \end{aligned} \tag{20}$$

We calculate new values for [x].

$$\begin{aligned} x_1^{i+1} &= \frac{b_1 - a_{12}x_2^i - a_{13}x_3^i}{a_{11}} \\ x_2^{i+1} &= \frac{b_2 - a_{21}x_2^i - a_{23}x_3^i}{a_{22}} \\ x_3^{i+1} &= \frac{b_3 - a_{31}x_2^i - a_{32}x_3^i}{a_{33}} \end{aligned} \tag{21}$$

These can be written as

$$[x]_{i+1} = (diag[A])^{-1} \cdot \left\{ [b] - \left([A] - diag[A] \right) [x]_i \right\} \tag{22}$$

Where diag[A] is the diagonal matrix

$$diag[A] = \begin{pmatrix} a_{11} & & & \\ & a_{22} & & \\ & & \ddots & \\ & & & a_{mm} \end{pmatrix} \tag{23}$$

We then check how much the values have changed. This is down using the Mean Absolute Percentage Error (MAPE) formula.

$$\begin{aligned} \epsilon_1^{i+1} &= \left| \frac{x_1^{i+1} - x_1^i}{x_1^{i+1}} \right| \\ \epsilon_2^{i+1} &= \left| \frac{x_2^{i+1} - x_2^i}{x_2^{i+1}} \right| \\ \epsilon_3^{i+1} &= \left| \frac{x_3^{i+1} - x_3^i}{x_3^{i+1}} \right| \end{aligned} \tag{24}$$

We then continue iterating until all values of ϵ are sufficiently small.
The final formula becomes [6]:

$$[x]_{i+1} = [x]_i + [D]^{-1} \cdot \left\{ [b] - [A][x]_i \right\} \tag{25}$$

2.1.2 Complexity

To calculate the complexity of the algorithm we look at each section separately.
Solving $x^{k+1} = c$ requires n divisions.
Computing $x = (L + U)x^k + b$ requires a number of additions and multiplications proportional to the non-zero entries.

Worst case becomes $O(n^2)$ [9].

As we only need to store the current values of the variables in the system, the space complexity becomes $O(n)$.

The algorithm will iterate until a stopping criterion is met.

$$||b - Ax^k|| < \epsilon \text{ or } ||x^{k+1} - x^k|| < \epsilon \tag{26}$$

2.1.3 Features & Limitations

Additionally, the Jacobi method can be parallelised for massive equation systems. This can be done row-wise, or column-wise, using cyclic shifting or global reduction [4].

However there is one major issue with the Jacobi method, it may not always converge to the correct solution. This can happen if the system of equations is not diagonally dominant [6]. The definition of a diagonally dominant matrix is:

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}| \tag{27}$$

"A square matrix is said to be diagonally dominant if for each row, the magnitude of the diagonal element must be greater than or equal to the sum of magnitudes of all other elements in that row" [6].

Algorithm 1 Jacobi method for solving a system of linear equations

```
Matrix A                          ▷ Initial guess x^0, diagonally dominant
Vector b                          ▷ Right hand side values of the equation
k = 0
while ||x^{k+1} − x^k|| > ε do
    for i := 1 until n do
        σ = 0
        for j := 1 until n do
            if j ≠ i then
                σ = σ + a_ij x_j^k
            end if
        end for
        x_i^{k+1} = (b_i − σ)/a_ii
    end for
    Increment k
end while
```

2.2 Gauss-Seidel Method

Gauss-Seidel method typically converges faster than the Jacobi method, which makes it a more efficient choice for many applications. It only converges when:

- A is symmetric positive-definite or
- A is strictly diagonally dominant

A matrix is called symmetric positive-definite when: $A = A^T$ and every eigen value is positive [12] [7].

The matrix [A] becomes decomposed into the sum of a lower triangular matrix [L'] and upper triangular matrix [U'] [5].

The goal is to solve

$$[A][x] = [b] \tag{28}$$

Given that $[A] = [L'] + [U']$ we get

$$\begin{aligned} ([L'] + [U'])[x] =& [b] \\ [L'][x] + [U'][x] =& [b] \end{aligned} \tag{29}$$

It is known that $[L'][x] = [b]$ is extremely fast and efficient to solve using forward-substitution [5]. We arrange the matrix equation to take advantage of this.

$$\begin{aligned} [L'][x] + [U'][x] =& [b] \\ [L'][x] =& [b] - [U'][x] \\ [x] =& [L']^{-1}([b] - [U'][x]) \end{aligned} \tag{30}$$

The update equation is derived from last expression.

$$[x]_{i+1} = [L']^{-1}\left([b] - [U'][x]_i\right) \tag{31}$$

The main difference between Jacobi and Gauss-Seidel algorithms is that Jacobi uses the old iteration value for each variable, while Gauss-Seidel uses the latest updated values of each variable [10]. The full algorithm is

Algorithm 2 Gauss-Seidel method for solving a system of linear equations

```
Matrix A                                  ▷ Initial guess φ, diagonally dominant
Vector b                                  ▷ Right hand side values of the equation
while until convergence or iteration limit do
    for i := 1 until n do
        σ = 0
        for j := 1 until n do
            if j ≠ i then
                σ = σ + a_ij φ_j
            end if
        end for
        φ_i = (b_i − σ)/a_ii
    end for
end while
```

2.2.1 Complexity

Gauss-Seidel improves on the Jacobi algorithm by re-using the calculations and using the already updated values. The complexity however stays the same at $O(n^2)$ [11]. Similar to Jacobi, the space complexity is also O(n) as we just need to store the values of the variables in the system of equations.

3 Comparison between the algorithms

I've implemented a "naive" algorithm, that simply finds the inverse of a matrix using cofactors, matrix of minors and transposition. This was done to test if the above-mentioned methods are actually faster. The tests were done 10 times each and then averaged to remove any outliers. Performance can be

affected by cold memory[1], memory location, variance, layout bias measurements, environment variable size or simple link order [1]. A way to reduce those factors is to perform multiple tests and then average them.

The tests were done by creating a diagonally dominant matrix of size n randomly based on a fixed seed to not introduce any variation. At every step, the solutions are also verified by calculating $Ay - b$ and testing against ϵ, where y $= A^{-1}b$.

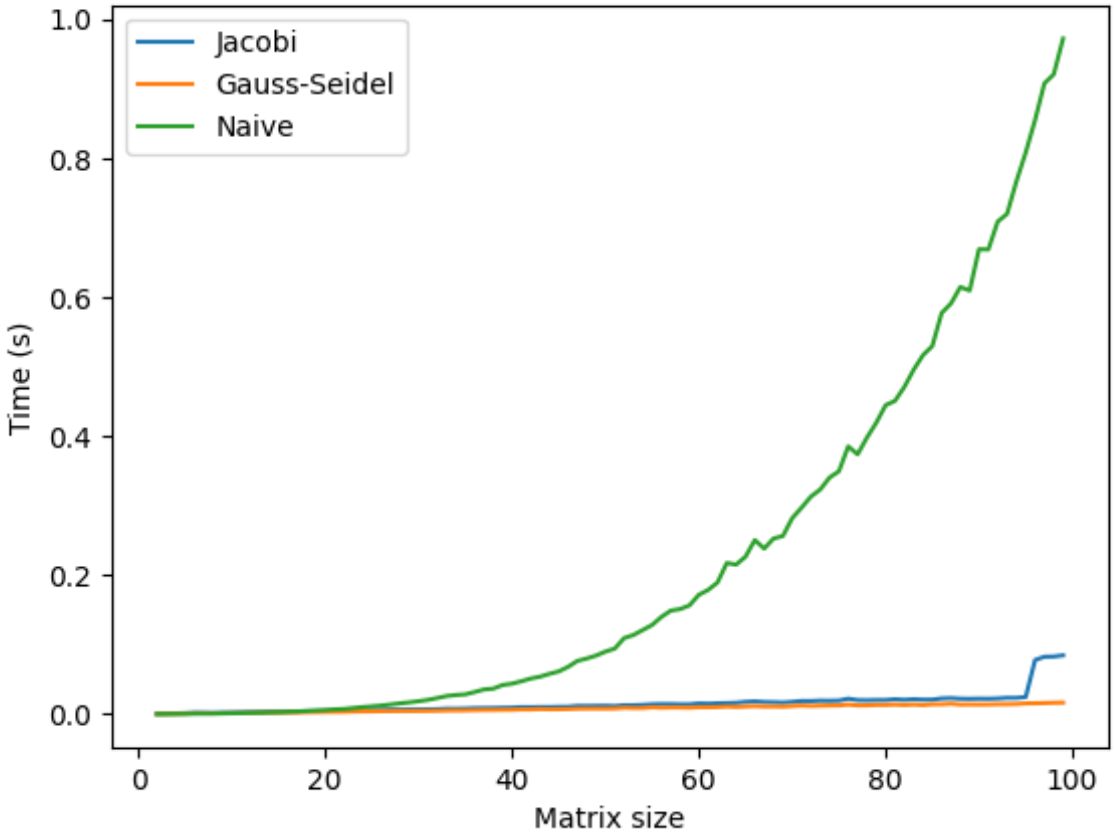

Figure 1: Average of 10 tests done for each matrix size from 2 to 100.

The naive method explodes very quickly. We can compare just the Jacobi and Gauss-Seidel method which will allow us to create a bigger graph.

[1]Cold memory here refers to a memory location that hasn't been accessed in a long time by the CPU.

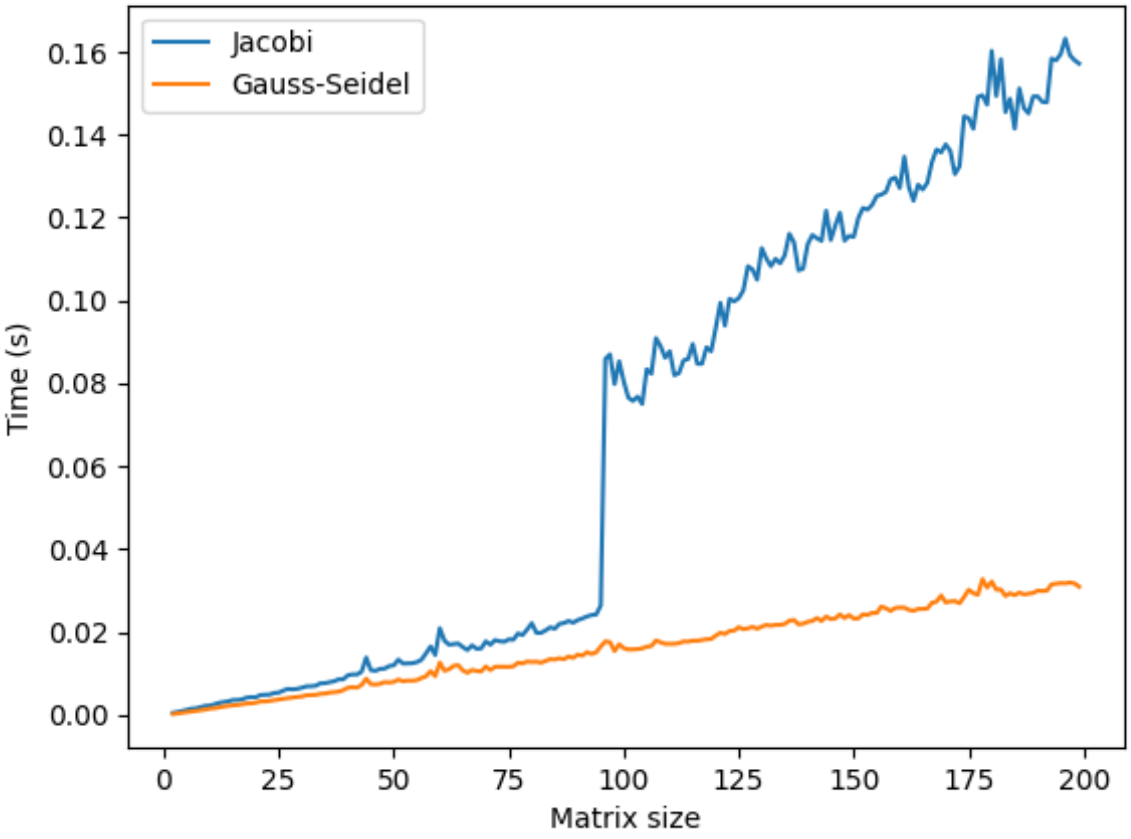

Figure 2: Average of 10 tests done for each matrix size from 2 to 200.

The sudden jump in Jacobi has been narrowed down to the implementation of "np.dot" method in the NumPy library which was used in the algorithm. At matrix size 95, it takes $\approx 33.8\mu s$ and at size 96 it takes $\approx 273.8\mu s$. For reproducibility, the file "numpy_dot_test.py" is included to showcase this.

4 Further Improvements

The Jacobi method and Gauss-Seidel method are both relatively simple and easy to implement, but they can be slow to converge. There are several ways that these methods could be improved to make them faster and more efficient.

One improvement could be to use a more sophisticated method to choose the values of the variables at each iteration. Currently, the Jacobi method takes the average of neighbouring values, a more sophisticated method could be used to choose better values that quicker converge to a solution.

Research published by IEEE has found an approach to improve the Gauss-Seidel method by employing the first three terms of Taylor expansion to approximate the reciprocals of diagonal entries and using the group inversion to obtain inversion of the lower triangle matrix [14].

5 Impact in the World

These algorithms have had a significant impact on the field of numerical analysis and have contributed to the development of more efficient methods for solving systems of linear equations.

They have major applications in the field of scientific computing, where they are used to solve systems of equations arising from the discretization of partial differential equations. This has led to

the development of more accurate and efficient models for simulating physical phenomena like fluid dynamics, electrical circuits and structural mechanics.

Additionally, these algorithms have widely been used in other areas of applied mathematics, such as optimisation and machine learning. These are massive scale optimisation problems and using naive methods is simply too slow.

References

[1] Emery Berger. ""Performance Matters" by Emery Berger". In: (). https://www.youtube.com/watch?v=r-TLSBdHe1A.

[2] Thomas H. Cormen et al. *Introduction to algorithms*. MIT Press, 2001.

[3] cuemath. "Inverse of Diagonal Matrix". In: (). https://www.cuemath.com/algebra/inverse-of-diagonal-matrix/.

[4] Steven Dong. "Parallel Jacobi Algorithm". In: (). https://www.brown.edu/research/projects/crunch/sites/brown.edu.research.projects.crunch/files/uploads/Parallel\%20Jacobi\%20-\%20MPI\%20code.pdf.

[5] EMPossible. "Lecture – Gauss-Seidel Method". In: (). https://www.youtube.com/watch?v=vN7fDiNxjss.

[6] EMPossible. "Lecture – Jacobi Iteration Method". In: (). https://www.youtube.com/watch?v=VH0TZlkZPRo.

[7] Aerin Kim. "What is a Positive Definite Matrix?" In: (). https://towardsdatascience.com/what-is-a-positive-definite-matrix-181e24085abd.

[8] Libretexts. *7.8: Solving Systems with Inverses*. Oct. 2021. URL: https://math.libretexts.org/Bookshelves/Algebra/Map%5C%3A_College_Algebra_(OpenStax)/07%5C%3A_Systems_of_Equations_and_Inequalities/708%5C%3A_Solving_Systems_with_Inverses.

[9] orionquest. "Iterative Methods". In: (). https://orionquest.github.io/Numacom/iterative.html.

[10] Ayush Prabhu. "What is the difference between the Gauss-Seidel and the Jacobi Method?" In: (). https://www.quora.com/What-is-the-difference-between-the-Gauss-Seidel-and-the-Jacobi-Method.

[11] Chris H. Rycroft. "Iterative methods for linear systems". In: (). https://math.berkeley.edu/~wilken/228A.F07/chr_lecture.pdf.

[12] Wikipedia contributors. *Gauss–Seidel method — Wikipedia, The Free Encyclopedia*. [Online; accessed 13-December-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Gauss%5C%E2%5C%80%5C%93Seidel_method&oldid=1126955589.

[13] Wikipedia contributors. *System of linear equations — Wikipedia, The Free Encyclopedia*. [Online; accessed 11-December-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=System_of_linear_equations&oldid=1124376016.

[14] Jing Zeng, Jun Lin, and Zhongfeng Wang. "An Improved Gauss-Seidel Algorithm and Its Efficient Architecture for Massive MIMO Systems". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 65.9 (2018), pp. 1194–1198. DOI: 10.1109/TCSII.2018.2801867.