Iterative Solvers - The Jacobi Algorithm

Charlie Wells[*]

Abstract
The Jacobi algorithm is an iterative method for solving systems of linear equations. These systems generally take the form $Ax = b$ where A is a square matrix, x and b are vectors, and we are trying to find each element in the x vector. These systems are ubiquitous, naturally arising in many fields, including mathematics, physics, economics, engineering, and computer science. These systems are commonly used in optimisation problems to minimise or maximise some quantity. The remainder of this report will further discuss concrete applications for linear systems of equations. I will also provide the mathematical background, pseudocode, and a video link detailing the python implementation for the Jacobi algorithm and compare it to other iterative methods, exploring the benefits and drawbacks of each.

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

Signature: Charlie Wells - cow203@exeter.ac.uk

[*] cow203@exeter.ac.uk

1. Systems of Linear Equations

1.1. Introduction to Linear Systems

A linear equation is an equation that we can represent with the form $a_1x_1 + ... + a_nx_n = b$, where $x_1, ..., x_n$ are variables and $a_1, ..., a_n$ are coefficients. A system of linear equations (also known as a linear system) is a collection of one or more linear equations involving the same variables. We can write these systems in the form $A\boldsymbol{x} = b$, where A is a square matrix (a matrix with the same number of rows and columns) of coefficients, $\boldsymbol{x}$ is a vector containing each of the variables which we are typically trying to find, and b is a vector which contains the result of multiplying A with x.
For example, with a linear system consisting of two equations, the following are equivalent:

$$A\boldsymbol{x} = b \quad \longrightarrow \quad \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \quad \longrightarrow \quad \begin{cases} a_{11}x_1 + a_{12}x_2 = b_1 \\ a_{21}x_1 + a_{22}x_2 = b_2 \end{cases}$$

1.2. Applications of Linear Systems

Systems of linear equations have many direct uses and applications across numerous fields, including the natural sciences, economics, mathematics, and engineering. These systems of linear equations also act as the foundation for algorithms used daily in many different contexts. For example, Google's PageRank algorithm, described by Brin and Page [1], realizes the importance of linear equations [2] and is fundamental to how we browse the internet. We can see this by considering the PageRank algorithm in matrix-vector form, which yields the equation:

$$L\boldsymbol{r} = \boldsymbol{r} \quad \longrightarrow \quad \begin{bmatrix} L_{11} & \dots & L_{1n} \\ \vdots & \ddots & \\ L_{n1} & & L_{nn} \end{bmatrix} \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix} = \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix} \quad \longrightarrow \quad \begin{cases} L_{11}r_1 + \cdots + L_{1n}r_n = r_1 \\ \vdots \\ L_{n1}r_1 + \cdots + L_{nn}r_n = r_n \end{cases}$$

where L is the linking matrix and $\boldsymbol{r}$ is the popularity vector that we wish to solve, which matches the form of the equation above.

For more complex models, we may encounter non-linear systems (for which the change of input is not proportional to the change of output) [3]. These non-linear systems are more difficult to solve than linear systems, with notable examples including the Navier-Stokes equations in fluid dynamics and Lotka-Volterra (predator-prey) equations in biology. However, through a process known as Linearization, we can convert non-linear systems of equations to a linear system and solve them through standard methods [4]. For example, the Navier-Stokes equation:

$$\rho \frac{D\vec{u}}{Dt} = -\nabla p + \rho \vec{g} + \mu \nabla^2 \vec{u}$$

has numerous uses in modeling fluid flow, weather patterns, climate, and even rendering natural phenomena in video games [5]. Attempting to solve for $\vec{u}$ would result in a partial differential equation with linear and quadratic terms, meaning that the equation would be non-linear. However, Bamieh et al. [6] have shown that through Linearization, we can represent this equation as a linear system and hence solve it using standard methods for solving systems of linear equations. The same holds true for the previously mentioned Lotka-Volterra equations [7] and many other non-linear systems which are fundamental to modeling real-world phenomena.

2. Iterative Method

There are numerous methods for solving systems of linear equations, for example, we can use direct methods such as Gaussian elimination, substitution, or computing the inverse of A. While these will produce correct values for $\boldsymbol{x}$, they are often slow, for example, calculating the matrix inverse has a time complexity of $O(n^3)$. While this may be fine for a dense matrix (with many non-zero elements), for a sparse matrix this is extremely inefficient.

Iterative algorithms solve linear equations by only performing multiplications by A and a few vector operations. These iterative algorithms do not provide an exact answer for $\boldsymbol{x}$ but will instead converge toward a solution, reducing the error the longer they run (the more iterations they perform). The iterative method typically follows the process as described by Aanjaneya. [8]:

- Write a matrix equation such that it is equivalent to our linear system.
- We make an initial guess for the value(s) of $\boldsymbol{x}$
- We determine value(s) for $\boldsymbol{x}$ by using iteration:

$$\boldsymbol{x}^{(k+1)} = T\boldsymbol{x}^{(k)} + c$$

Iterative methods do come with certain limitations. These techniques can only be applied to square linear systems (with n equations and n variables), however, this is a very common case. They are also not guaranteed to converge, in fact, each algorithm has different criteria for convergence, a few of which will be covered in a later section.

2.1. Jacobi Algorithm Introduction

The Jacobi Algorithm is notably the first iterative method used for solving systems of linear equations, and is the foundation for many other iterative methods (such as the Gauss-Seidel and Successive Over-Relaxation method). It relies on the assumption that the system of equations has a unique solution $\boldsymbol{x}$ and there is no zero entry among the leading diagonal elements of A (the coefficient matrix). We showed previously that we can represent our linear system as a set of equations. By rearranging these equations, we can isolate each variable in $\boldsymbol{x}$ and perform our iterative process:

$$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 \quad \longrightarrow \quad \therefore x_1 = [b_1 - (a_{12}x_2 + ... + a_{1n}x_n)]/a_{11}$$

which holds for all x_i in $\boldsymbol{x}$. By making an initial guess for each value of $\boldsymbol{x}^{(0)}$ and substituting these values into the right-hand side of the equation, we obtain an approximation for $\boldsymbol{x}^{(1)}$. Which we can continue iteratively to obtain a solution up to a given error tolerance (as long as the matrix meets the appropriate convergence criteria).
The Jacobi method will converge as long as the coefficient matrix A is strictly diagonally dominant. That is, the magnitude of each element on the leading diagonal (where the row is equal to the column), is greater than the sum of the magnitudes of all other elements on that row. As a result, checking for convergence requires us to access each element in the matrix, and hence is an $O(n^2)$ operation where n represents the number of rows or columns (these will be equal as it is assumed the matrix is square).

As we are using the Jacobi method to solve systems of linear equations, any of the applications for linear systems can be applied to the Jacobi method. For instance, we may use the Jacobi method to solve the linear equations for Navier-Stokes, PageRank, or the Lotka-Volterra equations as discussed in **Section 1.2**.

2.2. Pseudocode and Complexity

As discussed previously, the algorithm is used to solve linear systems, so will take $A, \boldsymbol{x}$, and b as inputs where A is the square coefficient matrix, $\boldsymbol{x}$ is the vector of variables we are attempting to solve and b is a vector storing the result of $A * \boldsymbol{x}$. We will also need to supply either a max iteration number n, indicating how many times the algorithm should iterate before stopping, or an error tolerance e, which indicates how close each value of $\boldsymbol{x}^{(k)}$ and $\boldsymbol{x}^{(k+1)}$ should be before we consider $\boldsymbol{x}$ to have converged. The output of the Jacobi method is the vector $\boldsymbol{x}$ after we have either met the max iteration limit or determined it to be close enough to the correct answer by using our supplied error tolerance. We may also decide to output the number of times the algorithm has iterated, allowing us to compare it to other iterative methods.
The algorithm itself consists of iteratively following the equation derived above:

$$x_i = [b_i - (a_{ij}x_j)]/a_{ii} \quad , \quad i \neq j$$

for each x_i in $\boldsymbol{x}^{(k)}$, which will produce the vector $\boldsymbol{x}^{(k+1)}$, until one of the stopping conditions are met.

The following flowchart details how the Jacobi algorithm works:

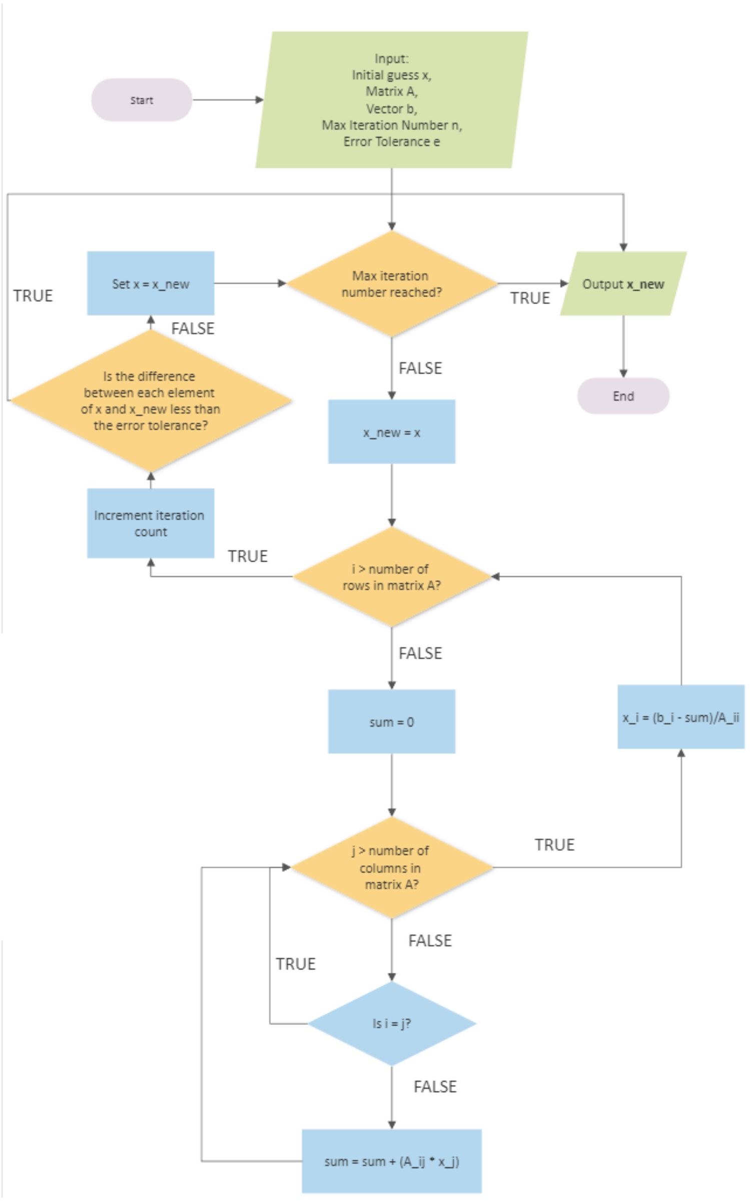

Fig. 1. A flowchart showing the Jacobi algorithm, the meaning of each input is explained above.

The Jacobi algorithm has a time complexity of $O(I * n^2)$ where n is the dimension of the row/column and I is the number of iterations. We can see in the flowchart that we are looping through each element in the matrix (as we loop through each row and have a nested loop through each column) which is an $O(n^2)$ operation. We can see that every other process is either vector multiplication or addition so can be done in linear time. This process is repeated until a stopping condition is reached, however, we cannot determine beforehand how many iterations this will take as it depends on either the max iteration variable or the error tolerance, so instead we will denote it with I in the time complexity. For this algorithm, we would need to store the matrix A, which has a space complexity of $O(n^2)$ where n is the dimension of the row/column, three vectors each with space $O(n)$, and variables with constant space. Hence, the space complexity of the Jacobi algorithm for a 2-dimensional matrix is $O(n^2)$. For higher dimensional matrices, we simply increase the power that n is raised to proportionally.

2.3. Comparisons, Limitations and Improvements

As discussed prior the Jacobi algorithm is reliant on the matrix being strictly diagonally dominant to converge. Comparatively, the Gauss-Seidel iterative method will converge under this condition, or if the matrix is positive-definite. The Gauss-Seidel method also will converge quicker than the Jacobi method if the convergence criteria are met. However, the Gauss-Seidel algorithm will use newly calculated values for x_1 in the calculation for x_2, whereas the Jacobi algorithm will only update these values each iteration. This means that the Jacobi algorithm favors parallelization [9], which is much more difficult to implement for the Gauss-Seidel method. Similarly, the Successive Over-Relaxation method has faster convergence and more lenient convergence criteria [10] than the Jacobi algorithm, so both these algorithms may be seen as a way to overcome the limitations of the Jacobi algorithm (namely its strict convergence criteria and slow rate of convergence).

3. Video Presentation

Available at: https://youtu.be/9CWipvLPS0A

Bibliography

[1] S. Brin and L. Page, “The anatomy of a large-scale hypertextual web search engine,” *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[2] A. Dode and S. Hasani, “Pagerank algorithm,” *10.9790/0661-1901030107*, vol. 19, pp. 2278–661, 02 2017.

[3] L. Hardesty, “Explained: Linear and nonlinear systems,” 2010.

[4] D. Cheng, X. Hu, and T. Shen, *Linearization of Nonlinear Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 279–313.

[5] J. Stam, “Real-time fluid dynamics for games,” 2003.

[6] M. Jovanović and B. Bamieh, “Modeling flow statistics using the linearized navier-stokes equations,” *Proceedings of the 40th IEEE Conference on Decision and Control (Cat. No.01CH37228)*, vol. 5, pp. 4944–4949 vol.5, 2001.

[7] A. Ionescu and F. Munteanu, “On the feedback linearization for the 2d prey-predator dynamical systems,” *Scientific Bulletin of the Polytechnic University of Timisoara. Transactions on Mathematics and Physics*, vol. 60, no. 74, p. 2, 2015.

[8] M. Aanjaneya, “Iterative methods,” 2017, https://orionquest.github.io/Numacom/iterative.html#:~:text=Convergence%20The%20Jacobi%20method%20is,c%20which%20requires%20n%20divisions.

[9] A. Margaris, S. Souravlas, and M. Roumeliotis, “Parallel implementations of the jacobi linear algebraic systems solver,” 03 2014.

[10] K. R. James and W. Riha, “Convergence criteria for successive overrelaxation,” *SIAM Journal on Numerical Analysis*, vol. 12, p. 137–143, 1975.