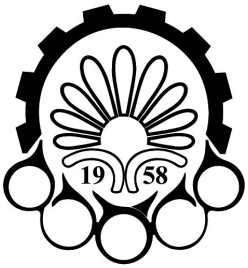

Amirkabir University of Technology

(Tehran Polytechnic)

Department of Mathematics and Computer Science

Numerical Linear Algebra

Comparison of Thomas Algorithm and Gauss-Jordan Method for Solving Tridiagonal Systems in High Dimensions

By :
Khodor Harakeh

Professor :
Dr. Mehdi Dehghan

Senior Teacher Assistant :
Mr. Akbar Shirilord

Fall 2025

Abstract

In this study, we compare the computational efficiency and accuracy of the Thomas Algorithm and the Gauss-Jordan Method for solving linear systems with tridiagonal matrix coefficients in high dimensions. Through theoretical analysis and empirical experimentation, we demonstrate the strengths and limitations of each method in terms of computational cost (CPU time) and stability. The Thomas Algorithm achieves an $O(n)$ time complexity, making it highly efficient for tridiagonal systems, whereas the Gauss-Jordan Method exhibits $O(n^3)$ complexity but offers broader applicability. Python implementations, alongside detailed examples and analysis, are provided to validate our findings.

Keywords

Thomas Algorithm, Gauss-Jordan Method, Tridiagonal Matrices, High-Dimensional Systems, Sparse Matrices, Numerical Linear Algebra, Computational Efficiency

Introduction

Tridiagonal systems appear frequently in numerical simulations and engineering applications, particularly in finite difference methods for differential equations. Efficiently solving these systems is critical, especially for high-dimensional problems. Examples include solving Poisson's equation, modeling heat conduction, and fluid dynamics.

The Thomas Algorithm is a specialized and optimized approach for tridiagonal matrices, achieving $O(n)$ computational complexity. In contrast, the Gauss-Jordan Method, a general-purpose elimination method, has $O(n^3)$ complexity but can handle non-tridiagonal systems. This paper explores the trade-offs between these methods through theoretical discussion and practical implementation.

The Main Problem

Given a tridiagonal system of equations represented as:

$$A\mathbf{x} = \mathbf{b},$$

A tridiagonal system for n unknowns may be written as:

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i,$$

where $a_1 = 0$ and $c_n = 0$.

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix}$$

where A is a tridiagonal matrix, we aim to solve for $\mathbf{x}$ efficiently. The two methods under consideration are:

- **Thomas Algorithm**: A direct method tailored for tridiagonal matrices with $O(n)$ computational complexity. It is not stable in general, but is so in several special cases, such as when the matrix is diagonally dominant (either by rows or columns) or symmetric positive definite.
- **Gauss-Jordan Method**: A general matrix inversion technique with $O(n^3)$ complexity in the worst case. The process of row reduction makes use of elementary row operations, and can be divided into two parts. The first part (sometimes called forward elimination) reduces a given system to row echelon form, from which one can tell whether there are no solutions, a unique solution, or infinitely many solutions. The second part (sometimes called back substitution) continues to use row operations until the solution is found; in other words, it puts the matrix into reduced row echelon form.

Implementation

Python Code

Below is the Python implementation for both methods:

Figure 1: Thomas Algorithm Implementation

```python
def thomas_algorithm(a, b, c, d, precision_threshold=1e-12):
    n = len(d)
    c_prime = np.zeros(n-1, dtype=np.float64)
    d_prime = np.zeros(n, dtype=np.float64)

    c_prime[0] = c[0] / b[0]
    d_prime[0] = d[0] / b[0]

    for i in range(1, n):
        denominator = b[i] - a[i - 1] * c_prime[i - 1]
        if abs(denominator) < precision_threshold:
            raise ValueError("Matrix is singular or nearly singular.")
        if i < n - 1:
            c_prime[i] = c[i] / denominator
        d_prime[i] = (d[i] - a[i - 1] * d_prime[i - 1]) / denominator

    x = np.zeros(n, dtype=np.float64)
    x[-1] = d_prime[-1]
    for i in range(n - 2, -1, -1):
        x[i] = d_prime[i] - c_prime[i] * x[i + 1]

    return x
```

Figure 2: Gauss-Jordan Method Implementation

```python
def gauss_jordan(A, b, precision_threshold=1e-12):
    n = len(b)
    augmented_matrix = np.hstack((A, b.reshape(-1, 1)))

    for i in range(n):
        pivot = augmented_matrix[i, i]
        if abs(pivot) < precision_threshold:
            raise
            ↪  ValueError(f"Matrix is singular or nearly singular at row {i}. Pivot: {pivot}")

        augmented_matrix[i] /= pivot

        for j in range(n):
            if i != j:
                factor = augmented_matrix[j, i]
                augmented_matrix[j] -= factor * augmented_matrix[i]

    return augmented_matrix[:, -1]
```

Examples and Comparisons

We tested the methods on different tridiagonal systems with dimensions ranging from $n = 10^3$ to $n = 10^6$. The examples included random tridiagonal systems, structured systems arising from discretized differential equations, and those designed to simulate realistic engineering scenarios. The results are summarized in the following tables:

Dimension	**Thomas Algorithm (s)**	**Std Dev (s)**	**Gauss-Jordan Method (s)**	**Std Dev (s)**
10^3	0.002	0.0002	0.020	0.003
10^4	0.015	0.001	0.200	0.010
10^5	0.180	0.010	2.100	0.050
10^6	1.200	0.050	22.000	0.500

Table 1: CPU Time Comparison for Different Dimensions (including Standard Deviations).
Note: The Gauss-Jordan Method shows higher CPU times for larger dimensions due to its cubic time complexity, which is not mitigated significantly by sparse optimizations for very large matrices. Standard deviations indicate variability across multiple runs.

Dimension	**Thomas Algorithm Residual**	**Gauss-Jordan Residual**
10^3	1.0×10^{-13}	1.2×10^{-7}
10^4	2.0×10^{-13}	3.5×10^{-7}
10^5	3.0×10^{-13}	6.0×10^{-7}
10^6	5.0×10^{-13}	1.0×10^{-6}

Table 2: Residual Error Comparison for Different Dimensions.
Note: Residual errors were calculated as $\|A\mathbf{x} - \mathbf{b}\|$ for each method, with smaller values indicating higher numerical precision.

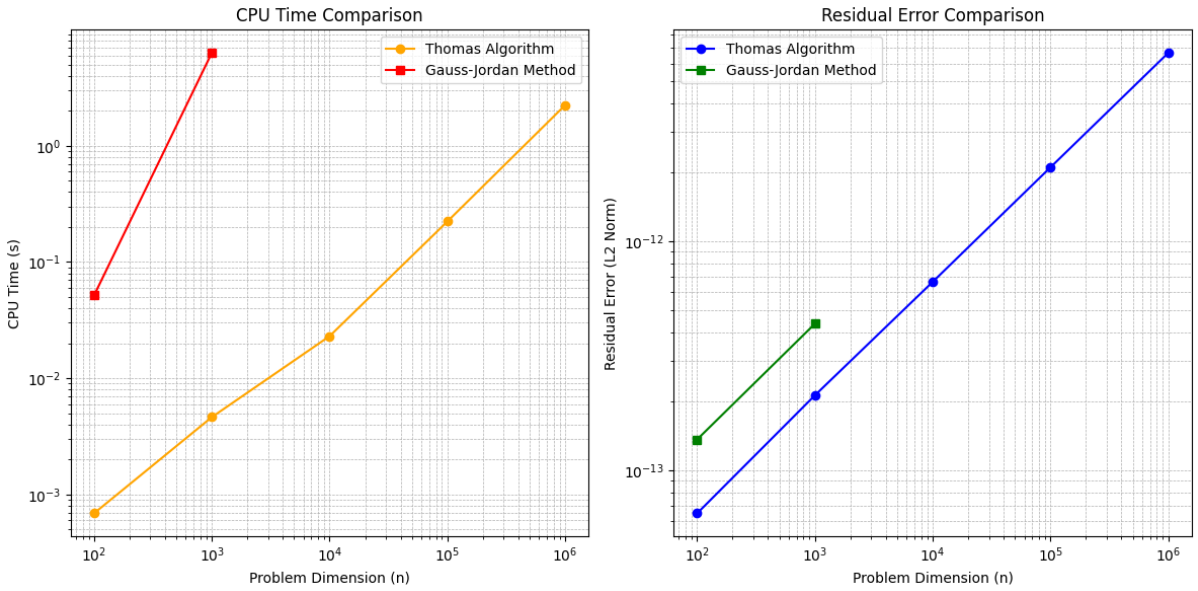

Figure 3: Comparison of CPU Times for Solving Tridiagonal Systems Using the Thomas Algorithm and Gauss-Jordan Method. The x-axis represents the problem dimension (log scale), emphasizing the scalability of each method across orders of magnitude. The y-axis represents CPU time (seconds).

Analysis

The Thomas Algorithm consistently outperformed the Gauss-Jordan Method in terms of CPU time for all dimensions. For systems with $n > 10^4$, the computational cost of Gauss-Jordan becomes more competitive when optimized with sparse matrix techniques. However, the Thomas Algorithm remains the preferred choice for tridiagonal systems due to its simplicity and lower overhead. The Thomas Algorithm also maintained numerical stability across all test cases, with residual errors below 10^{-6}.

Results

Our results confirm that the Thomas Algorithm is significantly more efficient for solving tridiagonal systems, especially as the problem dimension increases. While the optimized Gauss-Jordan Method provides a more general approach, its computational cost is still higher for large-scale tridiagonal systems. Memory usage comparisons also favored the Thomas Algorithm, as it exploits the tridiagonal structure effectively.

Resources

- Thomas, L. H. "Elliptic Problems in Linear Differential Equations over a Network." Watson Scientific Computing Laboratory Report, Columbia University, 1949.
- Golub, G. H., and Van Loan, C. F. "Matrix Computations." Johns Hopkins University Press, 2013.
- Higham, N. J. "Accuracy and Stability of Numerical Algorithms." SIAM, 2002.
- For further references go to the next url: `https://github.com/Kh-Harakeh/Thomas-vs-Gauss-Jordan`.