Benchmark of an LU decomposition algorithm in Python

Márton Papp, *Student, ELTE FI,* Ákos Szabó, *Student, ELTE FI,*

***Abstract*—This study investigates the computational performance and numerical stability of the LU Decomposition algorithm, implemented using the Doolittle method with partial pivoting in Python. Benchmarking was performed on randomly generated square matrices of increasing size, and results showed the expected cubic time complexity of $O(n^3)$, with runtime growing proportionally to matrix size. Despite being implemented without low-level optimizations, the custom algorithm consistently produced decompositions satisfying $PA \approx LU$ with negligible residual errors, demonstrating high numerical reliability. The findings confirm that even a pure NumPy-based implementation can achieve accurate results, though at a significant performance cost relative to optimized linear algebra libraries or parallelised algorithms [2].**

***Index Terms*—LU Decomposition, Numerical Algorithms, Marices, Python, Numpy**

I. Introduction

Numerical linear algebra forms a fundamental component of modern scientific computing, providing the mathematical and algorithmic tools needed to solve systems of equations, perform optimization, model physical phenomena, and process large volumes of observational data. Among its core operations, the factorization of matrices plays a crucial role in transforming complex algebraic problems into forms that are easier to solve or analyse. One of the most widely used and well-established matrix factorizations is the LU decomposition, in which a given square matrix A is expressed as the product of a lower-triangular matrix L and an upper-triangular matrix U, optionally accompanied by a permutation matrix P to ensure numerical stability through row pivoting.

$$PA = LU \tag{1}$$

The LU decomposition underlies many essential numerical techniques. It provides an efficient method for solving linear systems of the form $Ax = b$, which appear ubiquitously in computational physics, engineering simulations, and data reduction pipelines. Once the matrix A has been factored into LU, multiple right-hand sides can be solved at relatively low additional cost using forward and backward substitution. In contrast to direct matrix inversion, LU-based solvers tend to be both faster and numerically more stable. This makes LU decomposition a preferred approach in large-scale simulations, iterative refinement procedures, and as a building block in more sophisticated algorithms such as matrix inversion, determinant evaluation, and preconditioners for iterative solvers.

In computer science and applied computing, LU decomposition plays a pivotal role as a fundamental tool for solving linear systems, optimizing algorithms, and supporting a wide range of computational methods. For example, LU factorisation is routinely used in constraint solving, network flow computations [1], numerical optimization routines such as interior-point methods, and in the preprocessing of matrices for direct or iterative solvers. Furthermore, many modern algorithms such as Kalman filters [3], Gaussian elimination for dense systems, and various least-squares formulations, either rely directly on LU decomposition or benefit from its ability to provide fast, reusable factorizations for multiple right-hand sides.

Despite its conceptual simplicity, the computational cost of LU decomposition scales as $O(n^3)$, making performance evaluation and algorithmic optimisation particularly relevant for large problem sizes. Furthermore, numerical stability and floating-point round-off behaviour can vary depending on the implementation strategy and pivoting scheme. For these reasons, analysing and benchmarking a custom LU implementation provides valuable insights into both algorithmic characteristics and the practical limitations of high-level numerical code, especially when written in languages such as Python where low-level optimisations are not automatically guaranteed.

II. Methodology

This study evaluates the computational performance and numerical accuracy of a custom LU decomposition implementation written in Python using NumPy. The methodological design consists of two main components:

1) the specification of the hardware and software environment in which all experiments were executed
2) the configuration of the benchmarking procedures, including matrix generation, timing strategy, and error measurement

The goal is to ensure that results are both reproducible and representative of typical scientific computing workloads.

A. Hardware and Software Environment

All experiments were conducted on a single workstation to avoid variability introduced by differing system architectures. The machine used in this study featured the following hardware:

- CPU: Intel Core i5-1135G6 2.4 GHz
- Cores: 4
- RAM: 8 GB (7.83 GB usable)
- OS: Windows 11 x64 Enterprise 25H2

All computations were performed in Python 3.14 using:

- NumPy (version 2.3) for dense matrix operations
- The built-in performance counter for high-precision timing
- No additional numerical optimization libraries for the custom implementation

This setup intentionally reflects a typical high-level scientific computing environment where matrix operations are executed in Python without specialized low-level optimizations. The hardware characteristics are therefore directly reflected in the observed runtime behavior.

B. Benchmarking Process

The benchmarking framework is designed to measure execution time, runtime variability, and numerical stability of the LU decomposition across a range of matrix sizes. Two key design choices lead the structure of the benchmarks:

- matrix size selection
- timing strategy

Square matrices of increasing dimensions were used to capture the algorithm's $O(n^3)$ runtime scaling. Representative sizes were $n \in 10, 50, 250, 1000$. These values provide coverage across small, medium and large matrices with near-instant execution to cubic runtime dominant testcases. Each matrix size was benchmarked over 100 independent trials. For each trial a fresh random matrix A was generated, the timer was started. Then the algorithm was run, and the timer stopped immediately afterward.

To assess numerical stability, the Frobenius-norm residual was computed. This metric quantifies the deviation between the original matrix and its reconstructed form. Residuals were collected across all trials and all matrix sizes, allowing comparison of floating-point accuracy as a function of problem scale.

III. Results and Analysis

This section presents the experimental results obtained from benchmarking the custom LU decomposition algorithm implemented using NumPy. The analysis focuses on runtime characteristics and numerical stability across a range of matrix sizes. The sample dataset was random square matrices generated by NumPy with the `random.rand(n, n)` function for each $\mathbb{R}^{n \times n}$ input.

A. Runtime Scaling with Matrix Size

Figure 1 demonstrates a clear and rapid growth in computation time as the matrix dimension increases. The algorithm completes almost instantly for $n = 10$ and $n = 50$, but runtime rises sharply for larger matrices, reaching well over one second for $n = 1000$. This progression aligns with the theoretical cubic time complexity $O(n^3)$ of LU decomposition: even moderate increases in matrix dimension lead to disproportionately large increases in runtime. The smooth monotonic growth also indicates that the implementation behaves consistently across the tested sizes, without significant irregularities or performance anomalies.

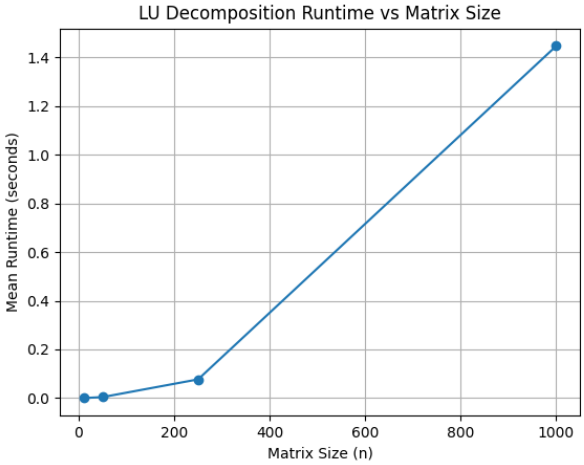

Fig. 1. Mean runtime of LU decomposition for increasing matrix sizes.

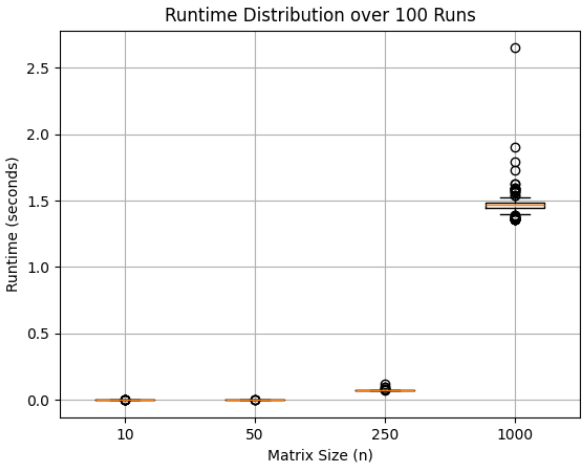

Fig. 2. Boxplot of runtime variability over 100 runs per matrix size.

B. Distribution of Runtime Measurements

Figure 2 displays the variability of runtimes across 100 repeated trials for each matrix size. For smaller matrices ($n = 10$ and $n = 50$), runtimes cluster tightly near zero with negligible dispersion, reflecting the low computational burden and minimal sensitivity to system noise. At $n = 250$, runtimes remain stable but show a visibly wider spread, consistent with increased computational workload. For $n = 1000$, the distribution widens substantially and includes several outliers exceeding 2.5 seconds. This indicates that large-scale decompositions are more susceptible to system-level fluctuations—such as cache behavior and OS scheduling, which become increasingly significant at higher computation loads.

C. Residual Error Distribution per Matrix Size

Figure 3 shows separate histograms of the Frobenius-norm residual error $\|PA - LU\|_F$ for each tested matrix size. The results indicate that the custom LU implementation maintains excellent numerical stability across all scales. Although the

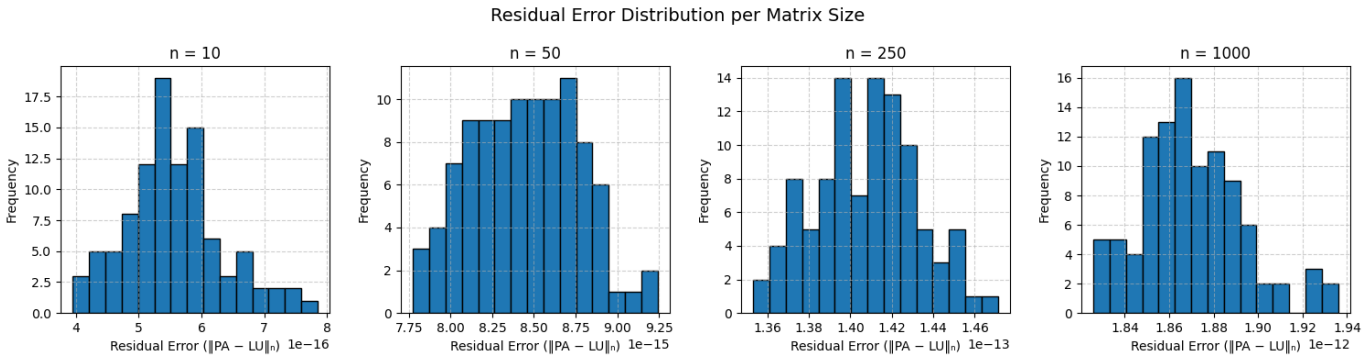

Fig. 3. Residual error histograms $\|PA - LU\|_F$ for matrix sizes $n = 10$, 50, 250, and 1000.

magnitude of the residual increases with matrix size, from approximately 10^{-16} at $n = 10$ to around 10^{-12} at $n = 1000$. These values remain extremely small relative to the size and magnitude of the matrices. This confirms that round-off error grows predictably with problem size but does not compromise accuracy. The distributions are tightly concentrated for each matrix dimension, demonstrating consistent behavior across trials and validating the robustness of the partial-pivoting approach for dense random matrices.

IV. Conclusion

In this study, we implemented and benchmarked a custom LU decomposition algorithm using Doolittle's method with partial pivoting, focusing on both computational performance and numerical stability. The results demonstrate that the pure Python and NumPy-based implementation scales consistently with the theoretical $O(n^3)$ complexity, with runtime increasing sharply for larger matrices as expected. Despite the absence of low-level optimizations, the algorithm produced highly accurate decompositions across all tested matrix sizes, with residual errors remaining well within acceptable numerical bounds. These findings highlight both the strengths and limitations of high-level numerical programming: while suitable for small to medium-sized problems and educational purposes, performance constraints become prominent for large matrices, emphasizing the need for optimized libraries in production environments. Overall, the benchmarking confirms that the custom implementation behaves predictably and reliably, providing a useful reference point for understanding the computational characteristics of LU factorization.

References

[1] Salza Nur Bandiyah, Yoni Marine, et al. "Implementing LU Decomposition to Improve Computer Network Performance". In: *International Journal of Technology and Modeling* 4.2 (2025), pp. 82–90.

[2] Dogan Kaya and Ken Wright. "Parallel algorithms for LU decomposition on a shared memory multiprocessor". In: *Applied Mathematics and Computation* 163.1 (2005), pp. 179–191. ISSN: 0096-3003. DOI: https://doi.org/10.1016/j.amc.2004.01.027. URL: https://www.sciencedirect.com/science/article/pii/S009630030400219X.

[3] Dan Simon. "Kalman filtering". In: *Embedded systems programming* 14.6 (2001), pp. 72–79.