Assignment 1: High Performance Dense Linear Solver

Christian Wiskott, 01505394.

April 19, 2020

Part 3: Unblocked Right-looking LU Factorization with Partial Pivoting

In this part of the assignment, the goal was to write an efficient algorithm for calculating an unblocked right-looking LU-Factorization *with pivoting* for a randomly generated, non-singular matrix $A \in \mathbb{R}^{n \times n}$. With this factorization, A is decomposed into an upper triangular matrix U, a lower triangular matrix L and a permutation matrix P, such that $PA = LU$. The main idea is similar to the native LU-Factorization, but seeks to swap the rows, in order to place the largest element of the current column in the main diagonal, and subsequently divide by the largest element to produce an normalized matrix L, such that all elements of $L_{ij} \leq 1$. Any swapping actions are also applied to the permutation matrix P, to not alter the systems of equations, i.e. such that $PA = LU$ is still fulfilled.

This should result in a more stable algorithm and higher accuracy, which will be evaluated in this paper, by showcasing a direct comparison of both methods.

Performance Evaluation

In order to evaluate the performance of the written code, the run-time was measured, such that only the actual computation of the resulting matrices U and P were included in the measurement. Similar to the native LU-Factorization, the computation required three explicit for-loops which scale with $\mathcal{O}(n^3)$ and can be examined in the provided code. As an addition, the code features the search for the largest element and the subsequent swapping of the rows of the matrices U and P, which should result in an increased run-time. This implementation utilized the same method as in

Part 2 of the assignment and constructed only one matrix U for the computation of A (instead of L and U) and one matrix for P, which required less memory to store the matrices.

Fig. 1 shows the run-time analysis of the LU-factorization *with pivoting* for problem sizes within the range of [100, 10000]. The data has been fitted with a cubic function, which shows (with high correlation) a cubic increase w.r.t the matrix dimension and coincides with the expected $\mathcal{O}(n^3)$-scaling. Due to this scaling, the required computation time increased very rapidly and reached more than 200 seconds for size 10k $\times$ 10k. This led to the termination of the evaluation, since higher dimension would result in exceptionally long run-times.

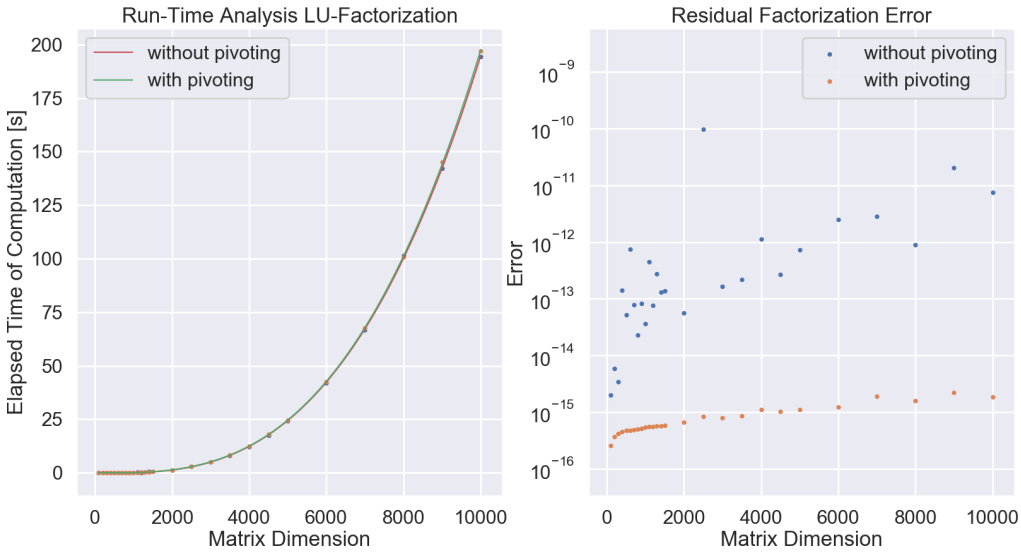

Figure 1: Run-time and Error Analysis

To minimize the run-time of the algorithm, the Python module $<$ *numba* $>$, which translates the code into much faster machine code, was used in the same manner as in Part 2 of the Assignment (please refer to the respective document for more details on the parallelization of the code). The same exact function with the above mentioned additions during the computation, was used in order to compare the effect the *pivoting* had on the run-time and the accuracy. This is visualized in fig. 1. Here it can be seen, that the run-time of both methods are essentially equal, with the non-pivoting method being around 1% faster at dimension 10k.

The vast improvement of a factor of $\sim 10^{-3}$ for the relative factorization error

at dimension 10k however, paints a clear picture of the effect of the *pivoting* on the calculations, since the errors are generally much lower and provide more stable results, which coincides with the theory stated in the introduction of the paper. Given, that virtually no additional run-time had to be afforded and the apparent advantages w.r.t. the errors, the *pivoting* is clearly superior to the native LU-Factorization.

In order to provide a reference value for the run-time, the code was executed on the server the university provided. Fig. 2 showcases the comparison of both variants of the LU-Factorization.

Since the server doesn't feature the module *numba*, the non-parallelization method was evaluated, same as in Part 2 of the assignment and showed similar results, with the run-time being roughly equal and showing lower errors. The problem at dimension 5k, was solved in around 340 seconds.

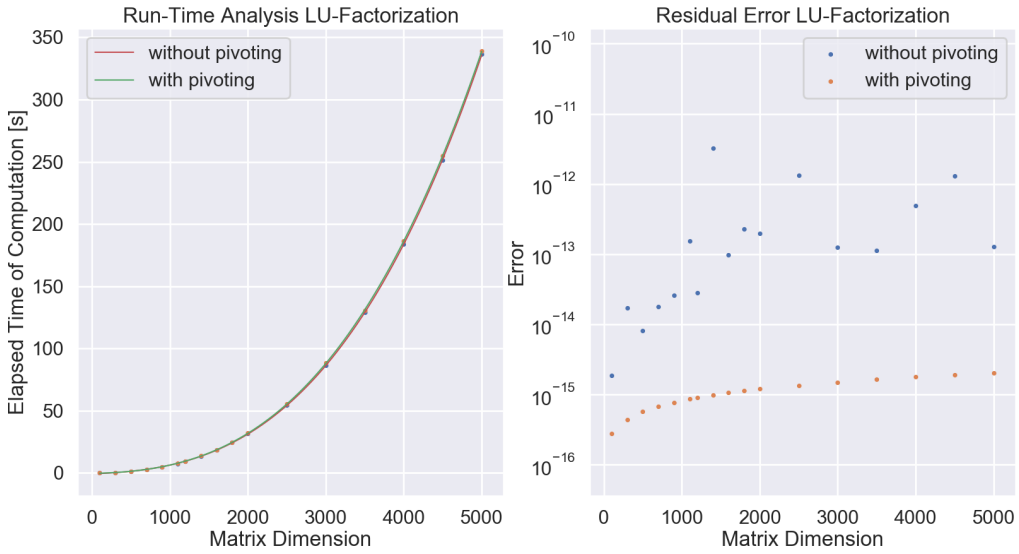

Figure 2: Run-time and Error Analysis using the provided university server