FCM Report

Ahmer Nadeem Khan

16 October 2024

1 Executive Summary

For this program, we will be creating routines and analyzing the algorithms for finding the LU factorization of a given non-singular matrix. We will consider three pivoting strategies for our routines: No pivoting, Partial Pivoting, and Complete Pivoting. We consider the relevant mathematics underlying these three problems. We then solve concrete full-rank square systems and apply the LU factorization, then two subsequent triangular solves to solve the system. We also consider the relevant mathematics for each pivoting strategy. For the flow of our solutions, we will first derive the algorithms mathematically, then provide concrete code in Python. The code in C++ will only be used for algorithm analysis, and basic validation with toy examples, and comprehensive testing drivers and reporting will be done in Python using the functionalities of the NumPy, SciPy and Matplotlib libraries.

2 Statement of the Problems

We will state here the 5 type of subroutines that we design and implement:

- The first subroutine takes a non-singular matrix A stored in 2D array and a flag denoting a pivoting strategy ('no', 'partial', and 'complete') and returns the LU factorization stored in the same 2D data structure, $A \leftarrow LU$.
- The second subroutine takes A in LU factored form stored in a 2D array, and returns the matrix product A.
- The third set of subroutines apply row/column permutations to an arbitrary matrix or vector when applicable. Multiple routines are needed because of how we store permutations, and the added complexity with column permutations when complete pivoting is done.
- The last set of routines solve the system $Ax = b$ iteratively for the three pivoting strategies.

2.1 Note on Storage and Languages

Our initial algorithms are run on C++, where the storage patterns and control flow of the algorithm is most obvious. In our subroutines, when we initialize a static and local 2D array, the values are automatically garbage. We then complete our user initialization of a lower triangular matrices in the specific required indexes, using $\frac{n(n+1)}{2}$ storage locations. We consider the other garbage values "undefined" and consider it free store. We demonstrate this effect in the source file.

Storing 1D arrays does not have this complication as we pre-compute the size of the required arrays, and only use as many storage locations. Since size must be defined and be constant as good practice, we manually change the size variables (RDIM and CDIM, for example) to test and validate.

In Python, we use the NumPy array structures to reproduce the algorithm, but here we start with initialized 2D = arrays, and then only specify the values that we want. The NumPy arrays are close to C++ arrays in their implementation. The reason we translate our code to Python is because of the ease of testing using standard libraries and plotting errors, graphs and tables. We also use Jupyter notebook to better control flow and help in testing.

3 Description of the Mathematics

3.1 LU Factorization

The LU factorization decomposes a given square, non-singular matrix $A \in \mathbb{R}^{n\times n}$ into the product of a lower triangular matrix L and an upper triangular matrix U. If A has no need for pivoting, the factorization is written as:

$$A = LU, \tag{1}$$

If pivoting is required to ensure numerical stability, LU factorization includes permutation matrices. Depending on the pivoting strategy, the factorization can be generalized as:

$$P_r A P_c = LU, \tag{2}$$

where:

- $P_r \in \mathbb{R}^{n\times n}$ is a *row permutation matrix* used to swap rows for stability.
- $P_c \in \mathbb{R}^{n\times n}$ is a *column permutation matrix* used to swap columns (for complete pivoting).

The permutation matrices are orthogonal transformations on A. It can be shown that PA always has an LU factorization given that A is non-singular.

3.2 Forward and Backward Substitution

After the factorization $P_r A = LU$ is obtained, the system of linear equations $Ax = b$ can be solved efficiently by substituting:

$$LUx = P_r b. \tag{3}$$

This is done in two steps:

1. **Forward substitution**: Solve $Lc = P_r b$ for c.
2. **Backward substitution**: Solve $Ux = c$ for x.

We employ the triangular solves from the program 1 to compute x. For complete pivoting, we get another added complexity because of the column permutations $Q = Q_1 \dots Q_n$. To determine what is required from the algorithm we can see the following derivation:

$$\begin{aligned}
Ax &= b \\
PAx &= Pb \quad \text{(Apply transformation)} \\
PA(QQ^{-1})x &= Pb \quad \text{(Insert identity via } QQ^{-1}\text{)} \\
(PAQ)(Q^{-1}x) &= Pb \quad \text{(Group transformations)} \\
U\tilde{x} &= \tilde{b} \quad \text{(Let } \tilde{x} = Q^{-1}x \text{ and } \tilde{b} = Pb\text{)} \\
x &= Q\tilde{x} \quad \text{(Recover the original solution)}
\end{aligned}$$

4 Description of the Algorithm and Implementation

Each algorithm follows immediate update wherever possible. For LU factorization, the only real algorithmic challenge comes from the data structure that we are working with. We start with a single 2D-array, and update entries within the same data structure. The entries for L are obtained by dividing a column by the pivot entry (exclusive), and then we compute the Schur complement in the active part of the matrix which iteratively drives the upper triangle (inclusive) of A to U. Implicitly, we use the pivot entry to force 0s below the pivot entry and keep track of the transformation which are characterized very simply (this was done in class). We know apriori how to compute L from these transformations, and this is reflected in how we assign values for L within this data structure. The algorithm continues for n-1 iterations (see Appendix for exact computations).

For partial and complete pivoting, we simply add additional functionality of discerning a preferable pivot entry from possible candidate entries. For partial pivoting, we check each column and pick the largest entry in absolute value, and permute this row to the top of the active part to be able to proceed with

the LU algorithm. For complete pivoting, we check the entire active part of the matrix to find the largest entry in absolute value, and drive this to the pivot entry by a row and column interchange if necessary. We then keep track of all of these swaps in a single 1D array of type $[0 \dots n-1]$ where the entry in the i-th index tells which row the i-th row is swapped with on the i-th step (this is the flow of the algorithm) - similarly for the column permutation matrix. The relevant transformation for systems have been discussed already.

For the triangular solves, we simply proceed with forward and backward substitution as discussed in class. With partial pivoting, we first employ the effect of our permutations to b, and then proceed with the solvers as before. With complete pivoting, we additionally compute Qx at the end of the algorithm where Qx has the effect of the reverse of our column permutations (i.e. in the reverse order). This produces the true x in the right coordinate system.

5 Algorithm Appendix

These section should be used with the Jupyter source file, where additional comments explain algorithmic choices and the previous section. The algorithm for the LU product is copied exactly from program 1.

Algorithm 1 LU Factorization without Pivoting

Input: $A \in \mathbb{R}^{n \times n}$, non-singular matrix
Output: contains both L (unit lower) and U (upper) matrices in place
for $i = 0, \dots, n-1$ **do**
    for $j = i+1, \dots, n-1$ **do**
        $A[j,i] \leftarrow A[j,i]/A[i,i]$ ▷ Store $L[j,i]$
        for $k = i+1, \dots, n-1$ **do**
            $A[j,k] \leftarrow A[j,k] - A[j,i] \times A[i,k]$
        end for
    end for
end for

Algorithm 2 LU Factorization with Partial Pivoting

Input: $A \in \mathbb{R}^{n \times n}$, non-singular matrix
Output: A contains both L (unit lower) and U (upper) matrices in place, P_r is the row permutation
for $i = 0, \ldots, n-1$ **do**
    `max_row` $\leftarrow \arg\max_{j \geq i} |A[j, i]|$ ▷ Find pivot
    if `max_row` $\neq i$ **then**
        Swap row i with row `max_row` in A
        Update P_r
    end if
    for $j = i + 1, \ldots, n-1$ **do**
        $A[j, i] \leftarrow A[j, i] / A[i, i]$ ▷ Store $L[j, i]$
        for $k = i + 1, \ldots, n-1$ **do**
            $A[j, k] \leftarrow A[j, k] - A[j, i] \times A[i, k]$
        end for
    end for
end for

Algorithm 3 LU Factorization with Complete Pivoting

Input: $A \in \mathbb{R}^{n \times n}$, non-singular matrix
Output: A contains both L (unit lower) and U (upper) matrices in place, P_r and P_c are the row and column permutations
for $i = 0, \ldots, n-1$ **do**
    (`max_row`, `max_col`) $\leftarrow \arg\max_{j,k \geq i} |A[j, k]|$ ▷ Find pivot
    if `max_row` $\neq i$ **then**
        Swap row i with row `max_row` in A
        Update P_r
    end if
    if `max_col` $\neq i$ **then**
        Swap column i with column `max_col` in A
        Update P_c
    end if
    for $j = i + 1, \ldots, n-1$ **do**
        $A[j, i] \leftarrow A[j, i] / A[i, i]$ ▷ Store $L[j, i]$
        for $k = i + 1, \ldots, n-1$ **do**
            $A[j, k] \leftarrow A[j, k] - A[j, i] \times A[i, k]$
        end for
    end for
end for

Algorithm 4 Multiplication of Unit Lower Triangular Matrix and Upper Triangular Matrix (Stored in One 2D Array)

Input: Combined matrix LU of size $n \times n$, where L is unit lower triangular and U is upper triangular, stored in a single 2D array.
Output: Matrix M of size $n \times n$, the result of $L \times U$
Initialize M as a zero matrix of size $n \times n$ ▷ Output matrix
Step 1: Compute the first row and first column
for $i \leftarrow 0$ to $n-1$ **do**
    $M[0][i] \leftarrow LU[0][i]$ ▷ First row is directly from U
end for
for $i \leftarrow 1$ to $n-1$ **do**
    $M[i][0] \leftarrow LU[i][0] \times LU[0][0]$ ▷ First column involves only L and U
end for
Step 2: Compute the remaining entries of M
for $i \leftarrow 1$ to $n-1$ **do**
    for $j \leftarrow 1$ to $n-1$ **do**
        $min\_limit \leftarrow \min(i, j)$
        $index \leftarrow 0$
        for $counter \leftarrow 0$ to min_limit **do**
            $M[i][j] \leftarrow M[i][j] + LU[i][counter] \times LU[counter][j]$
            $index \leftarrow index + 1$
        end for
        if $i \leq j$ **then**
            $M[i][j] \leftarrow M[i][j] + LU[index][j]$ ▷ Handle diagonal element multiplication with implicit 1 for L
        end if
    end for
end for

6 Description of the Experimental Design and Results

Testing has been done extensively on the Jupyter notebook, where the graphs are also available. For the testing design, we create functions for each of our subroutines, generate a variety of problems and conduct evaluations using chosen metrics. We also force double precision for our algorithm by specifying float-type when initializing matrices and using Python.

6.1 Problem Generation

To generate a non-singular matrix A of size $n \times n$, we first construct a unit lower triangular matrix L and an upper triangular matrix U. The product of these matrices, $A = LU$, ensures that A has full rank because both L and U have linearly independent columns. The full-rank matrix A is obtained by multiplying

L and U. We use a specified range of values for both L and U. To test for no-pivoting, we compute matrices that have a Cholesky factorization by finding matrices of the form LL^t for a random L with positive diagonal elements. This matrix will be positive semi-definite and not require any pivoting. For matrices with structure that scales with dimension, we will reply on the empirical tests and generate matrices of that kind and test for structure in our solutions/factors. Finally, we generate a random vector in n-dimensional real space to test our system solver. We generate matrices for various sizes (n=3 to n=500) and for varies values, in $[-500, 500]$ for example. We also test for different conditions on the matrices, e.g. all integer values, or integer values for smaller sizes of n. We also test for smaller values for the matrices say in the interval $(-1, 1)$.

6.2 Empirical Tasks

For the empirical tasks, since I am only intending to work with examples, I introduced new factorization functions where I use library codes and work with matrix structures for easy validation of the empirical results. These are included in a separate script file, where the outputs can be viewed.

- For task 1, no pivoting simply gives $L = I$ and $U = A$ since by definition U is triangular. This is easy to see. Partial Pivoting also gives the same result as the other elements in each column are always 0 so there is only 1 pivot candidate. For complete pivoting, since we have increasing diagonals, L is the same, U is now the reverse order of A on the diagonal and both P and Q have 1s on the anti-diagonal and 0s everywhere else. This is because at each iteration, we bring the (n-i)-th entry to i by a successive row and column exchange of the ends where the non-trivial entries lie. For the decreasing case, all pivoting strategies result in no permutations and the result in the very first instance of this task. This is also clear because of how the larger values are arranged.

- For task 2, no pivoting fails immediately. For partial pivoting, we have P again completely flipped, and $L = I$ and $U = anti(A)$ where anti means the diagonal switches to the main one instead. With complete, you have no row permutations, $Q = anti(I), L = I$ and U with the order of the elements switched. For the decreasing case, we have analogous results.

- For task 3 and no pivoting, the algorithm drives the anti-diagonal to 0 from the bottom left, and the update is made only to the main diagonal from the bottom right, until they it reaches the middle. For partial and complete pivoting, there is no particular structure in general, and the arrangement of the values on the diagonals determines how the algorithm proceeds. An example is covered in the code.

- Let's now look at task 8 the Cholesky factorization. For a symmetric positive definite (SPD) matrix A, the LU factorization can be written as $A = LU$ without requiring pivoting. To derive the Cholesky factor, extract

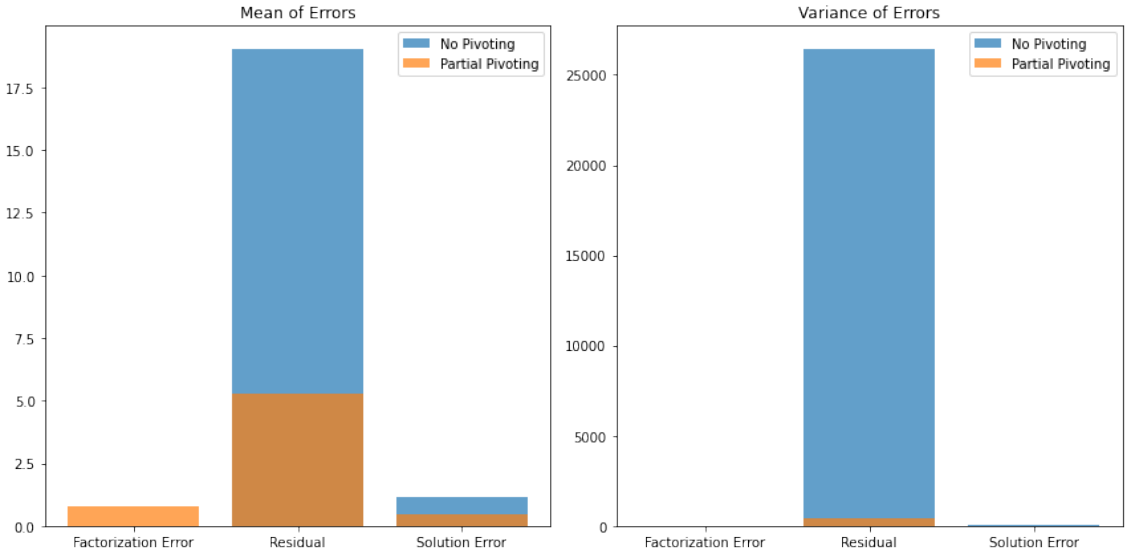
the diagonal matrix D from the diagonal elements of U, giving the decomposition $A = LDL^T$. Define $S = D^{1/2}$, where $S_{ii} = \sqrt{D_{ii}}$. The Cholesky factor L_0 is then computed as $L_0 = LS$, resulting in the decomposition $A = L_0 L_0^T$. The matrix L_0 obtained through this process is identical to the Cholesky factor found directly via the Cholesky decomposition. This is also done in the script file for an example.

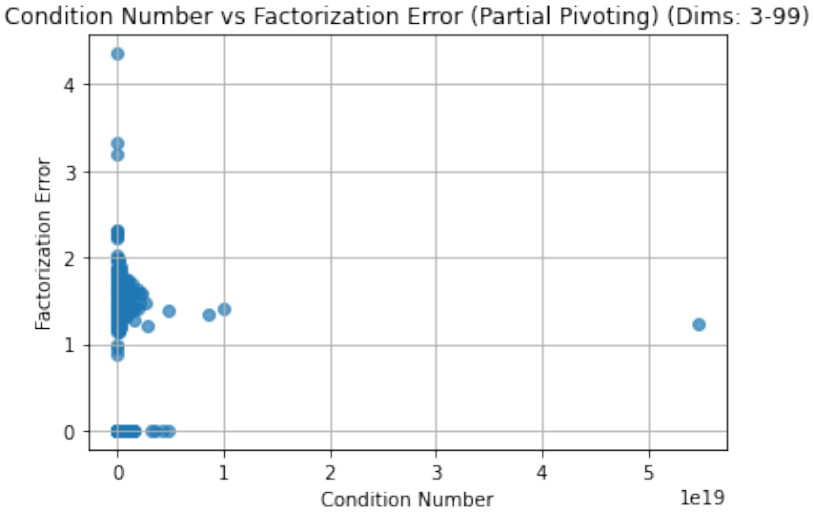
6.3 Testing Results

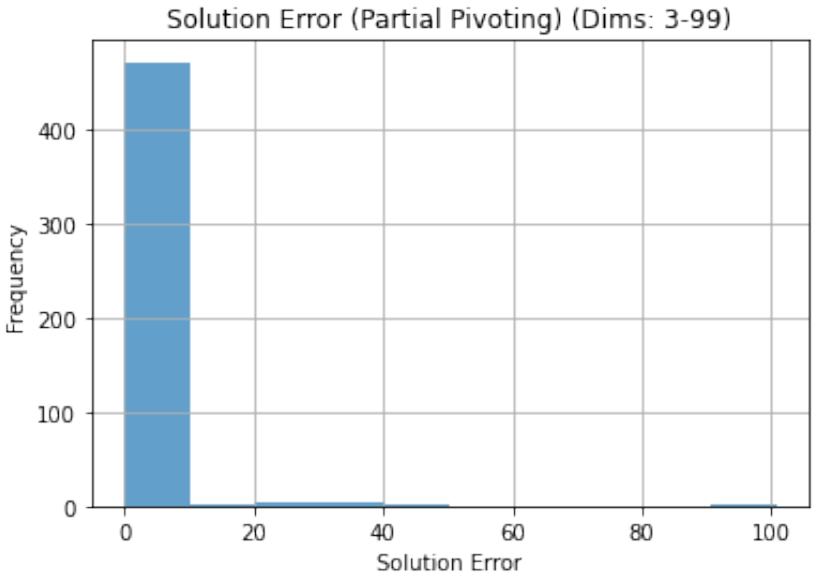
Firstly, I have chosen three metrics for the testing: the factorization error to test accuracy of the LU compute, and residual and solution error to check accuracy of the solvers and also implicitly the accuracy of the LU factorization. I have also used the infinity norm for matrices wherever appropriate because of the speed of computation that it allows. I have also generated appropriate plots and statistics for my metrics, specifying the range of dimension that these plots are from. Finally, I also generated plots for comparison of my metric to the condition number of the generated problem matrices. Please look at the Jupyter notebook and run the code to generate all graphs and plots. Here are samples of the statistics and plots generated:

```
No Pivoting:
Factorization Error - Mean: 0.000000, Variance: 0.000000
Residual - Mean: 37.151377, Variance: 101205.003011
Solution Error - Mean: 3.796198, Variance: 251.520520
```

```
Partial Pivoting:
Factorization Error - Mean: 0.746349, Variance: 0.587918
Residual - Mean: 85.570516, Variance: 2225864.932617
Solution Error - Mean: 2.041398, Variance: 48.289387
```


Mean of Errors
No Pivoting
Partial Pivoting
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
Factorization Error
Residual
Solution Error
Variance of Errors
No Pivoting
Partial Pivoting
25000
20000
15000
10000
5000
0
Factorization Error
Residual
Solution Error


Condition Number vs Factorization Error (Partial Pivoting) (Dims: 3-99)
Factorization Error
4
3
2
1
0
0
1
2
3
4
5
Condition Number
1e19



7 Conclusions

We can make conclusions about the numerical stability of our algorithms through the testing results. The factorization error for nice matrices (full rank, SPD) tends to generally stay low. The error is extremely low for SPD matrices specially, while it can be higher for just full-rank matrices. We can immediately see how much the structure of our matrices impacts the results. In certain cases the factorization error for partial pivoting seems to exceed no-pivoting entirely, but this also depended on the type of matrices. Higher dimensions tended to slow the algorithm down but did not have much effect on accuracy of the factorization - the only errors usually were runoff. The solution errors for partial pivoting were less than no pivoting; this makes sense since we mostly deal with swamping errors that may have appeared for very large values in our system. There was also much greater variance in the solution error, but significantly lower variance for partial pivoting. This also makes sense, since pivoting is numerically more stable. The residual error was very high, and went higher for pivoting; I suspect this is due to the ill-conditioning of the matrices, since many of them had very high condition numbers. For complete pivoting, there was much better performance on the solution metrics and also much less variance, but the factorization error was similar, and in certain instances even higher for very large dimension. This is potentially due to the high number of operations. Overall, the heirarchy of numerical stability is obvious from the results.

8 Program Files

There are two files included for this program: the Jupyter notebook for the routines and the driver, and a script file for the empirical tasks which can be run on any python compiler.