Numerical analysis: Homework 2

Student: Pratyush Sudhakar (ps2245)

Question 1:

Solution: When investigating the accuracy of calculating e^{-x} using the Taylor series for various values of x, it becomes evident that the discrepancy between the actual and computed values of e^{-x} widens with an increase in x. This phenomenon can be attributed to several factors inherent to the computational process, particularly when utilizing IEEE double-precision floating-point representation, which is limited to 15 to 17 significant digits.

The computation of e^{-x} through its Taylor series expansion,

$$e^{-x} = 1 - x + \tfrac{x^2}{2!} - \tfrac{x^3}{3!} + \tfrac{x^4}{4!} - \cdots,$$

introduces round-off errors at multiple stages. A notable source of error arises during the subtraction of consecutive terms of the series, especially for large values of x and higher-order terms. This is exemplified in the operation

$$\frac{x^i}{i!} - \frac{x^{i+1}}{(i+1)!},$$

where, for substantial values of x and i, the two terms become nearly equal in magnitude. The magnitude of terms in the Taylor Series initially increases with the power of x, only to decrease as the factorial in the denominator outpaces the growth in the numerator. The subtraction of such closely aligned numbers leads to catastrophic cancellation, a situation where significant digits are lost, resulting in a computed difference that fails to retain the true magnitude of variance, primarily due to the operation yielding a value smaller than the machine's precision.

This process of catastrophic cancellation is particularly detrimental to accuracy for large x values, as it effectively nullifies the significance of the calculation's precision at the juncture of term subtraction. The resultant computational error is non-trivial and manifests as notable deviations from the actual value, as corroborated by experimental observations.

We also know that computing x^i as well as computing factorials also introduces many round-off errors that when summed up, significantly grow in count. It could be shown as follows:

Let $S_k(x)$ represent the value of the Taylor series after k terms, and let $\tilde{S}_k(x)$ denote the actual computed value with errors. Employing the $1 + \delta$ model of round-off error, we can express $\tilde{S}_k(x)$ as follows:

$$\tilde{S}_k(x) = \left(\tilde{S}_{k-1}(x) + \frac{(-x)^k(1+\delta_{x^k})}{k!(1+\delta_{k!})} \right) \cdot (1+\delta_k),$$

where each operation—multiplication to compute $(-x)^k$, computing the factorial $k!$, and adding to the previously computed value—introduces some errors. Here, all δ values are less than or equal to the machine precision, ϵ, such that $|\delta| \leq \epsilon$.

Upon simplification, it becomes evident that the number of these δ values that are added grows significantly as the number of terms and the magnitude of x increases. As k increases, not only does the potential for error in each term increase due to the growing complexity of the operations involved, but the cumulative effect of these errors also becomes more pronounced. This results in a significant increase in the total error in $\tilde{S}_k(x)$, which can lead to a substantial deviation from the actual value of e^{-x}, particularly for large values of x.

Question 2:

(a) Prove that $\left(G^{(k)}\right)^{-1} = I + \ell^{(k)} e_k^T$

Solution: The matrix $G(k)$ is defined as $G(k) = I - \ell^{(k)} e_k^T$, where I is the identity matrix, $\ell^{(k)}$ is a column vector with the first k entries being zero and subsequent entries being non-zero, and e_k^T is a row vector with a 1 in the kth position and 0 elsewhere. This results in a matrix with 1s on the diagonal and the negative values of $\ell^{(k)}$ in the kth column below the kth row.

$$G(k) = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} - \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \ell_{k+1,k} \\ \vdots \\ \ell_{n,k} \end{bmatrix} \begin{bmatrix} 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \end{bmatrix}$$

$\implies$

$$G(k) = I - \ell^{(k)} e_k^T = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & \vdots & 0 \\ 0 & 0 & \cdots & -\ell_{k+1,k} & 1 & \vdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & -\ell_{n,k} & 0 & 0 & 1 \end{bmatrix}$$

To find the inverse of $G(k)$, we construct the augmented matrix $[G(k)|I]$ and apply row operations to transform the left half into the identity matrix. The operations will adjust the right half accordingly, and upon completion, the right half will represent the inverse of $G(k)$.

Steps:

The augmented matrix $[G(k)|I]$ is initially set up as follows:

$$[G(k) \quad | \quad I] = \left[\begin{array}{ccccccc|ccccc} 1 & 0 & \cdots & 0 & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 & \cdots & 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cdots & -\ell_{k+1,k} & 1 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & 1 & 0 \\ 0 & 0 & \cdots & -\ell_{n,k} & 0 & \cdots & 1 & 0 & 0 & \cdots & 0 & 1 \end{array}\right]$$

To find the inverse of $G(k)$, we perform row operations to transform the left half into the identity matrix I. Specifically, for each row i from $k+1$ to n, we add $\ell_{i,k}$ times the kth row to the ith row. This operation is aimed at eliminating the $-\ell_{i,k}$ values in the kth column and does not alter the kth row itself.

As a result of these operations, the kth column in the left half of the augmented matrix becomes zeros below the diagonal, effectively converting $G(k)$ into I. Correspondingly, these row operations modify the right half of the augmented matrix, which initially was I, into $I + \ell^{(k)} e_k^T$.

Hence, the final form of the augmented matrix is:

$$\left[I \quad | \quad I + \ell^{(k)} e_k^T \right]$$

This demonstrates that the inverse of $G(k)$ is $I + \ell^{(k)} e_k^T$, achieved through specific row operations on the augmented matrix $[G(k)|I]$.

(b) Prove that $\left(G^{(1)}\right)^{-1} \left(G^{(2)}\right)^{-1} \cdots \left(G^{(n-1)}\right)^{-1} = I + \sum_{k=1}^{n-1} \ell^{(k)} e_k^T$

Solution: Consider the product of inverses $\prod_{k=1}^{n-1} (G(k))^{-1}$. Using the result from part (a), we substitute $(G(k))^{-1}$ with $I + \ell^{(k)} e_k^T$:

$$\prod_{k=1}^{n-1} (G(k))^{-1} = \prod_{k=1}^{n-1} (I + \ell^{(k)} e_k^T).$$

We can prove it using Induction: Assume as the inductive hypothesis that for some $k = m$, the statement

$$\prod_{k=1}^{m} \left(G^{(k)}\right)^{-1} = I + \sum_{k=1}^{m} \ell^{(k)} e_k^T$$

holds true. This means that the product of the inverses of the first m $G(k)$ matrices results in a matrix that is the identity matrix plus the sum of $\ell^{(k)} e_k^T$ for all k from 1 to m, effectively filling all values under the diagonal up to column m.

Base Case

For the base case, consider $k = 2$, which involves $\left(G^{(1)}\right)^{-1}$ and $\left(G^{(2)}\right)^{-1}$. According to our substitution, we have:

$$\left(G^{(1)}\right)^{-1} \left(G^{(2)}\right)^{-1} = (I + \ell^{(1)} e_1^T)(I + \ell^{(2)} e_2^T).$$

Expanding this product, we get:

$$I + \ell^{(1)} e_1^T + \ell^{(2)} e_2^T + \ell^{(1)} e_1^T \ell^{(2)} e_2^T.$$

Since $e_1^T \ell^{(2)} = 0$ (because the vectors are orthogonal), the last term vanishes, resulting in:

$$I + \ell^{(1)} e_1^T + \ell^{(2)} e_2^T,$$

which confirms the pattern for $k = 2$.

Inductive Step

Assume the statement is true for some $k = m$, that is, the product up to $\left(G^{(m)}\right)^{-1}$ fills all values under the diagonal up to column m. Now consider the case for $k = m + 1$:

$$\left(\prod_{k=1}^{m} \left(G^{(k)}\right)^{-1}\right) \left(G^{(m+1)}\right)^{-1}.$$

By the inductive hypothesis, this is equivalent to:

$$\left(I + \sum_{k=1}^{m} \ell^{(k)} e_k^T\right) (I + \ell^{(m+1)} e_{m+1}^T).$$

Expanding this product and using the fact that $e_i^T \ell^{(j)} = 0$ for $i \neq j$, we obtain:

$$I + \sum_{k=1}^{m} \ell^{(k)} e_k^T + \ell^{(m+1)} e_{m+1}^T,$$

which is the desired format, thus proving the pattern holds for $k = m + 1$.

Therefore, by induction, we have shown that:

$$\prod_{k=1}^{n-1} \left(G^{(k)}\right)^{-1} = I + \sum_{k=1}^{n-1} \ell^{(k)} e_k^T,$$

completing the proof.

Question 3:

Assume that you are given an $n \times n$ non-singular matrix of the form

$$A = D + ue_{n-1}^T + e_{n-1}v^T$$

where u and v are length n vectors, and D is a diagonal matrix whose diagonal entries we will encode in the vector d, i.e., $D_{i,i} = d_i$.

(a) Under the assumption that we may compute the LU decomposition of A without requiring any pivoting, devise a means to solve a linear system of the form $Ax = b$ using $\mathcal{O}(n)$ time. For the purposes of this problem we will also consider the cost of memory allocation and therefore you cannot form A explicitly as that would take $\mathcal{O}(n^2)$ time.

Solution: Our objective is to solve a linear system $Ax = b$ efficiently, leveraging the structure of A and employing a blocking technique to compute the LU decomposition of A without explicitly forming the matrix. This approach aims to minimize both computational complexity and memory usage.

BLOCKING TECHNIQUE FOR LU DECOMPOSITION

The matrix A can be represented in a blocked form as follows:

$$A = \left[\begin{array}{cccc:c:c} d_1 & 0 & \cdots & 0 & u_1 & 0 \\ 0 & d_2 & \cdots & 0 & u_2 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & d_{n-2} & u_{n-2} & 0 \\ \hdashline v_1 & v_2 & \cdots & v_{n-2} & v_{n-1}+u_{n-1}+d_{n-1} & v_n \\ \hdashline 0 & 0 & \cdots & 0 & u_n & d_n \end{array}\right]$$

By applying the blocking technique recursively, we aim to decompose A into L and U matrices, where L is a lower triangular matrix with unit diagonal elements, and U is an upper triangular matrix.

Given the blocked form of A, we start with the decomposition: The matrix A can be expressed in block form as

$$A = \begin{bmatrix} D_{11} & u_{12} \\ v_{21} & D_{22} \end{bmatrix}$$

where D_{11} is the diagonal block of D, u_{12} is the corresponding block from ue_{n-1}^T, and v_{21} is from $e_{n-1}v^T$.

The LU decomposition of A can be computed recursively by considering a block matrix decomposition:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}$$

This leads us to the following relationships:

$$\begin{aligned} A_{11} &= L_{11}U_{11} \\ A_{21} &= L_{21}U_{11} \Rightarrow L_{21} = A_{21}U_{11}^{-1} \\ A_{12} &= L_{11}U_{12} \Rightarrow U_{12} = L_{11}^{-1}A_{12} \\ A_{22} &= L_{21}U_{12} + L_{22}U_{22} \\ S_{22} &= A_{22} - A_{21}A_{11}^{-1}A_{12} \Rightarrow L_{22}U_{22} \end{aligned}$$

where S_{22} is the Schur complement of A_{11} in A. From the above equations, we can infer that

$$\begin{aligned} L_{11} &= 1 \\ U_{11} &= d_1 \\ L_{21} &= [0, 0, \cdots, v_1/d_1, 0]^T \\ U_{12} &= [0, 0, \cdots, u_1, 0] \end{aligned}$$

COMPUTING L AND U

Through recursive application of the above decomposition, we derive the forms for L and U. Here, we present the final structures for L and U as given by:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \frac{v_1}{d_1} & \frac{v_2}{d_2} & \frac{v_3}{d_3} & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & \frac{u_n}{u_{n-1}+v_{n-1}+d_{n-1}-\sum_{i=1}^{n-2}(u_i*v_i/d_i)} & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} d_1 & 0 & 0 & \cdots & u_1 & 0 \\ 0 & d_2 & 0 & \cdots & u_2 & 0 \\ 0 & 0 & d_3 & \cdots & u_3 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & u_{n-1} + v_{n-1} + d_{n-1} - \sum_{i=1}^{n-2}(u_i * v_i/d_i) & v_n \\ 0 & 0 & 0 & \cdots & 0 & d_n - \frac{v_n*u_n}{u_{n-1}+v_{n-1}+d_{n-1}+\sum_{i=1}^{n-2}(u_i*v_i/d_i)} \end{bmatrix}$$

TIME AND SPACE COMPLEXITY ANALYSIS

For the block LU decomposition algorithm provided, we consider each step in a recursive process to compute the time complexity.

(a) **Computing the New i-th Column of L:**
Each recursive iteration involves computing the new i-th column of L. By using the fact that only the $n - 1th$ row has a non-zero value, we only perform a single operation, i.e. $L[n-1, i] = v_i/d_i$ in $O(1)$ time.

(b) **Computing the New i-th Row of U:**
The placement of u_i at the $n - 1$ position is a constant time operation, hence $O(1)$.

(c) **Computing the Schur Complement:**
The Schur complement for the i-th iteration is computed by subtracting $\frac{u_i \cdot v_i}{d_i}$ from the $A[n-1, n-1]$ element because $L_{21} \cdot U_{12}$ has only 1 non-zero element. This computation is also a constant time operation, hence $O(1)$.

Hence, since the operations are simple and applied to single elements or vectors, and the computation of the Schur complement is $O(1)$ due to the matrix structure, the effective computational complexity is closer to $O(n)$. Thus, the final effective complexity of the algorithm is $O(n)$, assuming in-place operations for the division of v_i by d_i and placement of u_i.

The slope of the logarithmic time taken by the algorithm is $1.02 \approx 1$. Thus, proving the time complexity analysis.

SOLVING LINEAR SYSTEM OF FORM $Ax = b$

Now that we have factorized A into $L \cdot U$, we can solve the linear system $Ax = b$ using the following steps:

(a) Find y, such that $Ly = c$.

(b) Find x, such that $Ux = y$.

This x will satisfy $Ax = b$ because $Ax = LUx = L(Ux) = Ly = b$.

By using the algorithm provided in the book, we can find x in O(n) time as well.

Total Time Complexity: Hence, LU decomposition of A ($O(n)$) and solving the linear system of equation, $Ax = b$ (also, $O(n)$), has combined time complexity of $O(n)$.

Total Space Complexity: We are only storing and using the non-zero elements of L and U ($3n$ elements in total) and therefore the space complexity is also $O(n)$.

The slope of the logarithmic time taken by the algorithm to solve for x is $0.90 \approx 1$. Thus, proving the time complexity analysis.

(b) Implement your method and demonstrate that it achieves the desired scaling. You can do this with random instances of the problem (i.e., randomly generate d, u, and b). While these random instances may benefit from pivoting, they are unlikely to "require" pivoting in a strict sense (i.e., you might be solving the problem inaccurately, but will not end up dividing by zero).

Solution: After running the algorithm on values of n ranging from $[10000, 11000, 12000, \cdots, 100000]$, the logarithmic amount of time taken linearly ($slope = 0.90 \approx 1$) increased with an increase in matrix size. Hence, proving the $O(n)$ complexity of the matrix.

(c) Solve the associated linear system $Ax = b$ for x. Using a logarithmic scale for the error, plot both the absolute error of the difference between your computed solution and x_{true} and the absolute value of the residual vector $r = b - Ax$ generated by your solution vs the index of the respective vector (i.e., the x-axis will range from 1 to 50,000). Also report the 2-norm of both r and the difference between your computed solution and x_{true}. What do you observe?

Solution: The algorithm takes b as the input and outputs the corresponding x vector that satisfies $Ax = b$. However, because of round-off and floating point errors, the actual value is slightly different from the computed value.

2-norm of residual vector ($||r||_2 = ||b - A\hat{x}||_2$): 0.00048087601317092776.
2-norm of difference between computed solution and xtrue ($||\hat{x} - x_{\text{true}}||_2$): 1.6177130063438136e-07.

The small 2-norm of the difference between the computed solution and x_{true} indicates a high degree of accuracy in the computed solution.

Also, from the plotted graph, we can see that one of the element in the computed solution is significantly different from the true solution. This is likely because of the element present at $(n-1, n-1)$, (n, n), $(n-1, n)$, and $(n, n-1)$ positions of L and U matrices that were computed using multiple operations and hence, the round-off error was larger for these elements.

The relative error is,

$$\frac{||\hat{x} - x_{\text{true}}||_2}{||x_{\text{true}}||_2} = 7.274016550386751e - 10$$

The relative residual error (input to the problem),

$$\frac{||r = b - A\hat{x}||_2}{||b||_2} = 1.732482894020007e - 09$$

(d) By avoiding pivoting, we were able to get a fast, $\mathcal{O}(n)$ algorithm. However, what do you observe about the accuracy of the solution and how might you explain it?

Solution: The condition number (Relative Error / Relative Perturbation) is

$$C.N. = \frac{Relative\ Error}{Relative\ ResidualError} = 0.41986080067482323$$

The Condition Number is less than 1. Therefore, the algorithm is well-conditioned and for small changes in input, there will only be small changes in the output.

Additionally, we can do an error analysis to prove the accuracy of the algorithm as follows:

The algorithm involves 2 steps: LU Decomposition (Schur Complements) and Solving $Ax = b$ (Forward and Backward Substitution).

LU Decomposition:

During the LU Decomposition, we are performing 3-5 operations in each iteration. The main source of error, however, is from updating the second to last element of the diagonal of the Schur Complement ($S[n-2]$). Let $\hat{S}_i[n-2]$ be the floating-point approximation of the $n-2th$ element. Then, by definition of floating point error,

$$\hat{S}_k[n-2] = (\hat{S}_{k-1}[n-2] - \frac{u_i \cdot v_i}{d_i}(1 + \delta_{k1}))(1 + \delta_{k0})$$

δ_{k1} and δ_{k0} are the relative errors due to floating-point arithmetic in the subtraction and multiplication steps, respectively.

These errors δ are bounded by the machine epsilon ϵ, which is the upper bound on the relative error due to rounding in floating-point arithmetic. Specifically, $\delta_{k0} \leq \epsilon$ and $\delta_{k1} \leq \epsilon$ for all $k \in [0, 1, 2, \cdots, n-2]$.

When these errors propagate through the iterations, we can express the total relative error in the final computed Schur Complement $\hat{S}[n-2]$ in terms of these δ's and ϵ. This accumulated error influences the overall accuracy of the LU decomposition can be bounded by *epsilon*.

Solving $Ax = b$:

During forward and backward substitution, the relative errors are introduced in a similar fashion. In forward substitution, when solving $Lz = b$, and in backward substitution, when solving $Ux = z$, errors occur due to finite precision arithmetic when performing divisions and multiplications. However, even these errors could be bounded by an upper bound.

Hence, the algorithm is backward stable because it is well-conditioned and the error could be bounded by an upper bound. Also, the errors in computed results (both random experiments and problem data) are very low (order of 10^-7 for the given data and order of 10^-20 for random data).

(e) If we introduce partial pivoting into our algorithm can we guarantee that the algorithm will still have $\mathcal{O}(n)$ complexity? If not, what about $\mathcal{O}(n^2)$ complexity?

Solution: If we introduce partial pivoting, we will not have $\mathcal{O}(n)$, because we employed the Blocking technique which uses the outer product of L and U rows and columns. If any of the v_i is greater than d_i, pivoting will swap the rows, resulting in disturbing the sparse structure of the matrix. It will result in ”fill-ins” which means there will be non-zero elements introduced at indices that were originally zeros.

This will force us to use all the elements of the matrix while computing Schur complements.

While computing time complexity, we will consider the worst-case scenario where we do row-operations at every iteration. It will result in computing outer product of L and U row, column vectors. Thus, it will take $O(n * n^2)$ time. Therefore, if we introduce partial pivoting, we won't be able to solve the linear system of equations in less than $O(n^3)$ time.

QUESTION 4:

$$A = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 1 \\ -1 & 1 & 0 & \cdots & 0 & 1 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ -1 & \cdots & -1 & 1 & 0 & 1 \\ -1 & \cdots & -1 & -1 & 1 & 1 \\ -1 & \cdots & -1 & -1 & -1 & 1 \end{bmatrix}$$

(a) Consider computing an LU decomposition of A (with or without partial pivoting, the answer will be the same either way). Work out an expression for the largest entry of U in terms of n.

Solution: To perform the LU decomposition, we execute row operations to transform A into an upper triangular matrix U, ensuring the diagonal elements remain as 1, and adjusting the last column through a sequence of additive operations based on the row transformations.

Through these operations, for each row i from 2 to n, we add the first $i-1$ rows to the ith row to eliminate the -1s below the diagonal. This modification impacts the last column such that the entry in the last column of the ith row becomes the cumulative sum of 1s from all preceding rows plus its initial 1. This can be conceptualized as a binary addition where each row addition effectively doubles the value in the last column of the subsequent row.

Consequently, the pattern of values in the last column for the ith row aligns with powers of 2, specifically:

- For the 1st row, the last column value is $1 = 2^0$,
- For the 2nd row, the value is $2 = 2^1$,
- For the 3rd row, the value is $4 = 2^2$,
- Following this pattern, for the nth row, the value in the last column will be 2^{n-1}.

Therefore, the largest entry of U in terms of n is 2^{n-1}, located in the last column of the last row of the upper triangular matrix U. This outcome stems from the geometric progression of the binary addition process inherent in the row operations performed during the LU decomposition of matrix A.

(b) If we computed an LU factorization in practice and encountered an entry of this size do you think it would be problematic? why or why not?

Solution: When considering the practicality of computing an LU factorization that results in an entry of size 2^{n-1}, several significant issues arise:

(a) **Round-off Error:** The computation of an entry of size 2^{n-1} involves a significant number of operations, especially as n becomes large. Each of these operations carries the potential for round-off errors due to the finite precision with which computers represent numbers. As these errors accumulate, they can lead to a substantial computational error in the final result. The exponential growth of the entry size exacerbates this issue, making the round-off error a critical concern in the accuracy of the LU factorization.

(b) **IEEE Double Precision Bounds:** The IEEE standard for double-precision floating-point numbers provides a mechanism for representing a wide range of numerical values. However, this representation is limited to a finite number of bits (64 bits in the case of double precision). For values of n greater than 64, the entry 2^{n-1} exceeds the maximum value that can be represented in double precision, rendering such numbers impossible to accurately store or compute with. This limitation severely restricts the ability to perform LU factorization for matrices of moderately large sizes, as the necessary values cannot be represented within the bounds of IEEE double precision.

(c) **Complexity in LU Computation:** The LU factorization process involves the computation of the product of the lower triangular matrix L and the upper triangular matrix U. Given that the entries of U can reach sizes as large as 2^{n-1}, the multiplication operations within this product involve dealing with large numbers, further increasing the likelihood of computational errors. The complexity of these operations not only makes the factorization process more prone to error but also more computationally intensive, impacting the efficiency and reliability of the factorization.

In summary, the presence of an entry of size 2^{n-1} in an LU factorization introduces significant practical challenges. These include the accumulation of round-off errors, limitations imposed by IEEE double precision bounds, and the increased complexity and error susceptibility of the LU computation process. Together, these factors underline the importance of careful consideration and potentially the use of alternative methods when dealing with matrices that may lead to such large entries during factorization.

(c) Does the problem persist if we use complete pivoting?

Solution: The problem will not persist when we use complete pivoting because in complete pivoting, at each iteration, we will be using the element with maximum value in both the column and the row.

Doing so, none of the elements will be growing spontaneously in value as they were doing so without complete pivoting. Because the elements that were originally growing spontaneously will be used as the pivot elements and will not grow any further.

The maximum absolute value of an element in the upper triangular matrix will be 2 which can be proved using Induction on the size of matrix A. First, for simplicity, I will show a practical application.

Using the code from this GitHub Repository, I ran the Gaussian Elimination Complete Pivoting algorithm on matrices of sizes: $[4*4, 80*80, 200*200, 800*800]$. The structure of the upper triangular matrix was as follows:

$$\text{UpperTriangularMatrixStructure} = \begin{bmatrix} 1 & 1 & 0 & \cdots & 0 \\ 0 & 2 & 1 & \cdots & 0 \\ 0 & 0 & -2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 1 \\ 0 & 0 & \cdots & 0 & -2 \end{bmatrix}$$

We also know that the lower triangular matrix L will also naturally maintain moderate values, as row operations will only involve adding or subtracting at most 2 times the value of another row and it will be independent of the actual entries in matrix A.

Now, we aim to prove by induction that complete pivoting prevents excessive growth in the entries of the U matrix during LU decomposition of a matrix A of a special form.

Inductive Hypothesis: $I(n)$: Upper Triangular Matrix after Gaussian Elimination with Complete Pivoting on the square matrix A (of form defined in the problem) of size n will have the same structure as UpperTriangularMatrixStructure.

Base Case for $n = 3$:

Consider the 3×3 matrix A given by

$$A = \begin{pmatrix} 1 & 0 & 1 \\ -1 & 1 & 1 \\ -1 & -1 & 1 \end{pmatrix}.$$

With complete pivoting, we choose the largest absolute value as the pivot. Initially, all the diagonal elements have the largest absolute value of 1. We will use the element a_{11} as the pivot. No row exchanges are necessary since a_{11} is already the largest in absolute value.

Next, we perform the elimination steps. To eliminate a_{21}, we add the first row from the second row. To eliminate a_{31}, we add the first row from the third row. The matrix after these operations is

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \\ 0 & -1 & 2 \end{pmatrix}.$$

Now we move to the second iteration. The pivot will be a_{23} since it's the largest in absolute value in its row that we can use to cancel the terms below the diagonal. Hence, we will swap the thrid and the second column. Resulting in,

$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 2 & 1 \\ 0 & 2 & -1 \end{pmatrix}.$$

Now, we will subtract the second row from the third row. Resulting in,

$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & -2 \end{pmatrix}.$$

The above matrix is now the upper triangular matrix U. It matches the form shown by programmatically performing Gaussian Elimination with Complete Pivoting using the code provided.

Inductive Step:

Assume $I(n)$ holds for an $n \times n$ matrix A. Now consider an $(n + 1) \times (n + 1)$ matrix A' and let's show $I(n + 1)$ holds after Gaussian Elimination with Complete Pivoting.

Let A' be

$$A' = \begin{pmatrix} 1 & 0 & \cdots & 0 & 1 \\ -1 & 1 & \cdots & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ -1 & -1 & \cdots & 1 & 1 \\ -1 & -1 & \cdots & -1 & 1 \end{pmatrix}.$$

After performing the n steps of Gaussian elimination with complete pivoting, the upper triangular matrix will be

$$U_{(n)} = \begin{bmatrix} 1 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 2 & 1 & \cdots & 0 & 0 \\ 0 & 0 & -2 & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & 1 & 2 \\ 0 & 0 & 0 & \cdots & -1 & 2 \end{bmatrix}$$

Finally, we will swap the n and the $n+1$ columns and then subtract the nth row from the $n+1$th row. Resulting in,

$$U = \begin{bmatrix} 1 & 1 & 0 & \cdots & 0 \\ 0 & 2 & 1 & \cdots & 0 \\ 0 & 0 & -2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 1 \\ 0 & 0 & \cdots & 0 & -2 \end{bmatrix}$$

This has the same structure as UpperTriangularMatrixStructure. Hence, proving the inductive step.

Conclusion:

By the principle of mathematical induction, we have proven that complete pivoting in the LU decomposition process of a matrix A of the special form will result in a U matrix whose maximum absolute value of any entry will be 2, preventing excessive growth of the entries.

Hence, by employing complete pivoting, we will not be facing issues of exponential increase of the values of matrix A while performing Gaussian Elimination to perform LU factorization. However, it is important to note that complete pivoting will increase the time complexity of the process.