Foundations of Computational Math 1 Fall 2022

Programming assignment 2

General Task

Consider the case where $A \in \mathbb{R}^{n\times n}$ is nonsingular and the system $Ax = b$ is to be solved via LU factorization without pivoting, using partial pivoting, and using complete pivoting. Implement a code that is capable of performing these three tasks, based on a user selection, and one or more test routines designed to evaluate and validate the correctness of the code. Your code should also detect situations where the factorization may not proceed and exit gracefully. You may not use standard libraries such as LAPACK or built-in matrix routines for pieces of your routines with the exception of the test routines.

The assignment has two major components; i) developing the algorithms and ii) testing the given example.

1 Code Structure and Tasks

1.1 Construction of the data

Step 1: Generate a matrix $A \in \mathbb{R}^{n\times n}$

- Generate $L \in \mathbb{R}^{n\times n}$ and $U \in \mathbb{R}^{n\times n}$ so they are nonsingular unit lower triangular and upper triangular matrices, respectively. Evaluate their product as A. Take care of the magnitude of the elements of L and U. Nicely conditioned problems tend to be specified by off-diagonal elements in L smaller than 1 in magnitude and diagonal elements in U that are not too small compared to off-diagonal elements in U.
- *Remark. Although A is generated from given L and U, if you run your routine with partial or complete pivoting nontrivial permutation matrices P and/or Q from the end of this assignment, you may result to obtain $\tilde{L}$ and $\tilde{U}$ such that $PAQ = \tilde{L}\tilde{U}$.*
- Two types of matrix A you will test (each type tests for two matrix sizes: $n = 10$ and $n = 100$):
  1. **Randomly generated matrices tend to be nonsingular and reasonably conditioned.** You can make sure any matrix is nonsingular by adding to the diagonal elements until the matrix is diagonally dominant. This guarantees the success of the factorization if run without pivoting. As noted above, if you allow pivoting the routine may so do even for a diagonally dominant matrix depending on the form of dominance and the pivoting strategy used. It is also useful to note that after generating such a matrix you can apply random permutations P and Q to generate a new matrix A that will not be diagonally dominant but will still be nonsingular.
  2. **Generate a symmetric positive definite A from a nonsingular lower triangular L_1 via $A = L_1 L_1^T$.** A is nonsingular by definition and will succeed without pivoting. (Note that your code will still produce an LU factorization since it is not assumed to exploit symmetry. However, if no pivoting is used, $A = LU = LDL^T$ where D is diagonal and positive. So you can check if your computed U satisfies this compatibility condition with L.)

Step 2: Given a matrix $A \in \mathbb{R}^{n\times n}$, one can also generate a vector $b \in \mathbb{R}^n$ based on a chosen solution $x \in \mathbb{R}^n$. That is, for given A, choose a random values for x and generate b via the matrix-vector product $b = Ax$.

1.2 Develop LU algorithm

Step 3: LU factorization for A, i) without pivoting, ii) using partial pivoting, and iii) using complete pivoting (extra credit).

Routine requirement:

- *Input:* The matrix A stored in a simple 2-dimensional array-like data structure and other relevant parameters such as n.
- *Output 1:* Return the matrices L and U stored within the array that contained A on input, i.e., you should implement the in-place algorithm strategy. The 1 values on the diagonal of L, the 0 values in the upper triangular portion of L and the 0 values in the lower triangular portion of U should not be stored at any point in the code.
- *Output 2:* The routine should also return the permutation matrices P and/or Q appropriate to the pivoting strategy used. These permutation matrices should not be stored as full matrices within any array. The matrices, or equivalently the set of elementary permutations that are their factors, can be represented by at most n integers each.
- **For the full credit: suggested memory efficient algorithms should be employed.** A flag indicating what form of the factorization should be also attempted, e.g., no pivoting, partial pivoting, or complete pivoting.

Example of Step 3:

Given $A = \begin{bmatrix} 3 & 17 & 10 \\ 2 & 4 & -2 \\ 6 & 18 & -12 \end{bmatrix}$ with partial pivoting case, we will get $LU = PA$ with

$$P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \; L = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ 1/3 & -1/4 & 1 \end{bmatrix}, \; U = \begin{bmatrix} 6 & 18 & -12 \\ 0 & 8 & 16 \\ 0 & 0 & 6 \end{bmatrix}.$$

Your LU factorization routine will give you the output:

$$Matrix = \begin{bmatrix} 6 & 18 & -12 \\ 1/2 & 8 & 16 \\ 1/3 & -1/4 & 6 \end{bmatrix}, \text{ and pivoting vector } p = [3, 1, 2],$$

where L is stored as the strictly lower triangular part and U is stored as the upper triangular part. Note that the diagonal entries of $Matrix$ are the diagonal entries of U. We do not need to store the diagonal entries of L since we know they are all one.

Storing Permutations:

Let $n = 4$ and consider the following elementary permutations:

- P_1 swaps rows 1 and 4
- P_2 swaps rows 2 and 3
- P_3 swaps rows 3 and 4

Accumulate effects of P_i:

$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} \xrightarrow{P_1} \begin{pmatrix} 4 \\ 2 \\ 3 \\ 1 \end{pmatrix} \xrightarrow{P_2} \begin{pmatrix} 4 \\ 3 \\ 2 \\ 1 \end{pmatrix} \xrightarrow{P_3} \begin{pmatrix} 4 \\ 3 \\ 1 \\ 2 \end{pmatrix}$$

Therefore you can use a 1-dimensional array to store the permutation matrix.

Step 4: Solve $Ly = b$ (forward substitution) and Solve $Ux = y$ (backward substitution)

You need to write a solver routine for this step: with the input matrix containing L and U, pivoting information, and b, you will get the solution x.

4.1: Solve $Ly = b$ (forward substitution)

4.2: Solve $Ux = y$ (backward substitution)

Extra Credit For one of the above algorithm, either 4.1 or 4.2, test for both row-oriented and column-oriented methods. Compare the computational time and complexity of these two methods. Did you see any differences? Why?

- Row-oriented: The routine should return the solution y to $Ly = b$ in the same array that the vector b was input, i.e., the solution should overwrite b in its data structure.
- Column-oriented: The routine will return the solution y in a new array.

Requirement for the whole solver routine :

- Your solver routine (solution routine) should accept as input a vector b stored in a simple 1-dimensional array. The routine should also accept as input the 2-dimensional array containing the information specifying the L and U and the 1-dimensional arrays specifying P and Q if appropriate.
- The forward and backward solving $Ly = b$ and $Ux = y$ must also work only with the information specified in the 2-dimensional array and not an expanded version of L and U.

Step 5: Check the accuracy

1. Check the factorization accuracy
$$\frac{\|PAQ - LU\|}{\|A\|}$$
where $\|A\| \geq 1$, i.e relative error for large A.
   - Matrix norm: $\|\cdot\|_1$ and $\|\cdot\|_F$
   - In order to evaluate your code's function you must implement one or more tests. These will require various support routines. These include a routine that accepts as input the 2-dimensional array containing the information specifying the L and U matrices and return in a separate 2-dimensional array the product $M = LU$. Note that this matrix multiplication must use the information specifying L and U within the data structure. It must not expand them into separate arrays that include the 1 and 0 values that are not stored in the input array. You will also need a routine that accepts as input the 1-dimensional arrays specifying P and Q and applies them to vectors and matrices stored in 1-dimensional and 2-dimensional arrays.
2. If the solution is known by design of the problem, check
$$\frac{\|x - \tilde{x}\|}{\|x\|}$$
where $\tilde{x}$ is the computed solution and $\|x\| \geq 1$.
Vector norm: $\|\cdot\|_1$ and $\|\cdot\|_2$.
3. You should check the accuracy via the residual $b - A\tilde{x}$ and
$$\frac{\|b - A\tilde{x}\|}{\|b\|}$$
assuming $\|b\|$ for all attempts to solve a system, i.e., whether or not you know the true solution.
Vector norm: $\|\cdot\|_1$ and $\|\cdot\|_2$.

2 Correctness Test Task

Besides the above two types of matrices, you need to test the following linear system $A_{test}x = b_{test}$ to check the correctness of your code:

$$A_{test} = \begin{bmatrix} 2 & 1 & 0 \\ -4 & 0 & 4 \\ 2 & 5 & 10 \end{bmatrix}, \text{ and } b_{test} = \begin{bmatrix} 3 \\ 0 \\ 17 \end{bmatrix}$$

For this matrix A_{test}, you only need to finish the following tasks and attach the results to your report.

1.
   - Compute the output matrix $Matrix_p$ and pivoting vector p by applying LU factorization routine with partial pivoting to A_{test}, i.e. $[Matrix_p, p]$=lu-factorization(A_{test}, 'partial')
   - Compute the solution x_p using the above output $Matrix_p$ and p.
2.
   - Compute the output matrix $Matrix_c$, pivoting vectors p and q by applying LU factorization routine with complete pivoting to A_{test}, i.e. $[Matrix_c, p, q]$=lu-factorization(A_{test}, 'complete')
   - Compute the solution x_c using the above output $Matrix_c$, p and q.

You need to put all the above results $Matrix_p, p, x_p$ and $Matrix_c, p, q, x_p$ into your report to get credits for this task. Note that A_{test} is from the class notes.

Note:

- The textbook has relevant MATLAB coding examples. Programs 1, 2, and 3 in Chapter 3 implement triangular system solving when L and U are given in a 2-dimensional array. Note that some thought must be given when adapting these to the current assignment since L and U will be stored together in a single 2-dimensional array resulting from the factorization algorithm.
- Programs 4, 5, and 6 give in-place versions of LU without pivoting. This style is the form expected for the solution to the assignment, i.e., no matrix operations are implemented directly in terms of full dense matrices. These routines do not solve the assignment, however, since they lack the required pivoting capabilities.
- Program 9 implements complete pivoting and therefore performs one of the assigned tasks. However, its style is one that yields clarity and ease of implementation. It is **not acceptable** for the solution to the assignment since it is wasteful of computations and space. Notice how an entire matrix is used to represent the accumulated row and column permutations. The accumulation of Gauss transforms and permutations is accomplished by full matrix multiplication, each requiring $2n^3 + O(n^2)$ operations despite the fact that the matrices have the structure that we have seen to reduce complexity substantially. A solution program using this style will not receive credit.

Submission of Results

Expected results comprise:

- A document describing your solutions as prescribed in the notes on writing up a programming solution posted on the Canvas.
- The source code, makefiles, and instructions on how to compile and execute your code including the Math Department machine used, if applicable.
- Code documentation should be included in each routine. (You don't need to paste your code in the writing report).
- All text files that do not contain code or makefiles must be PDF files. **Do not send Microsoft word files of any type.**

These results should be submitted by 11:59 PM on the due date. Submission of results is to be done via Canvas.

Assessments

- You should do i) no pivoting and ii) partial pivoting. However, iii) complete pivoting will be considered as extra credit (10pts).
- Comparing row and column-oriented methods will be extra 10pts.