MATH 315, Fall 2020
Homework 4

James Nagy

Homework Team *(8)*: *(Diego Gonzalez, Yuting Hou, Mike Lin, Julin Ye)*

This assignment is associated with Chapter 3, Solution of Linear Systems. It illustrates numerical methods for solving linear systems of equations, i.e. $Ax = b$, and also some complexities introduced when solving this problem on a computer.

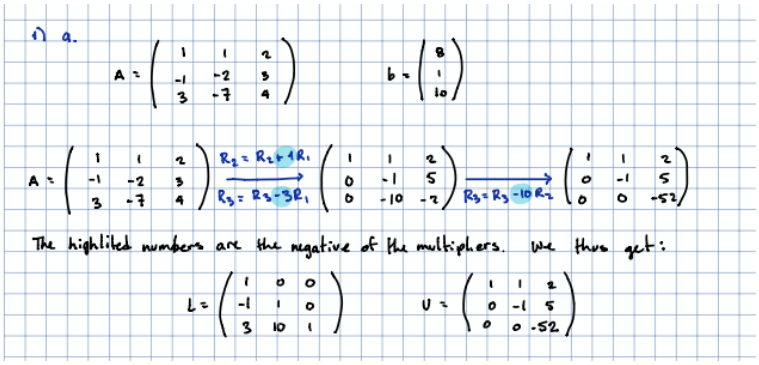
NOTE: For hand calculations in the first problem, you do not need to LaTeX the details and steps – you can do this with pen or pencil, scan or save as a PDF file, and attach to the LaTeX parts.

1. Consider the following system of equations:

$$\begin{aligned} x_1 + x_2 + 2x_3 &= 8 \\ -x_1 - 2x_2 + 3x_3 &= 1 \\ 3x_1 - 7x_2 + 4x_3 &= 10 \end{aligned}$$

(a) Use Gaussian elimination to find the $A = LU$ factorization of A (by hand). Which built-in MATLAB function can be used to determine the LU factorization? Discuss the results you obtain when you implement this function.

Answer: This is our handwritten solution:

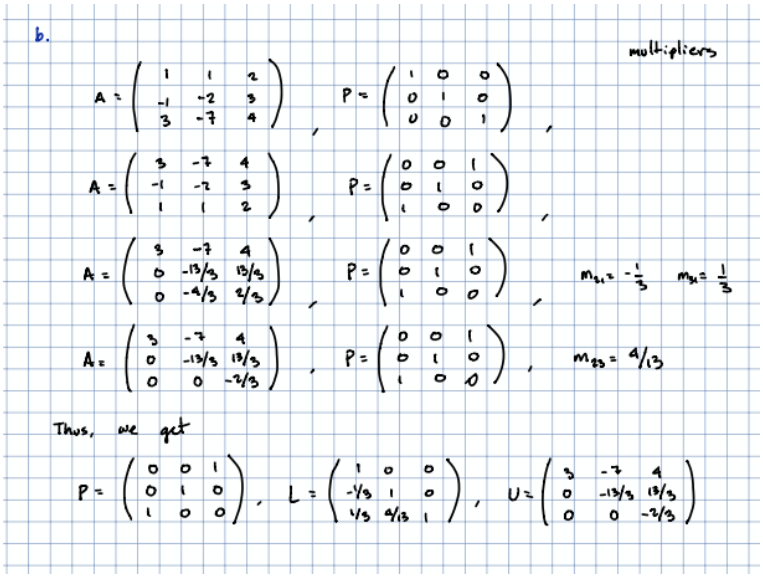


The MATLAB function that performs the LU factorization is `[L,U] = lu(A)`. However, when we execute this function in MATLAB, we get a different result.

```
L =

    0.3333   -0.7692    1.0000
   -0.3333    1.0000         0
    1.0000         0         0


U =

    3.0000   -7.0000    4.0000
         0   -4.3333    4.3333
         0         0    4.0000
```

As we can see, L isn't really a lower triangular matrix here. This is because when MATLAB computes the LU factorization, it always does pivot permutations to ensure the stability of the problem. Furthermore, both L and U have different values because MATLAB converts the L matrix into a "unit" matrix. When taking into account both of these nuances, we are able to see that the answer that MATLAB provides is the same as our solution computed by hand.

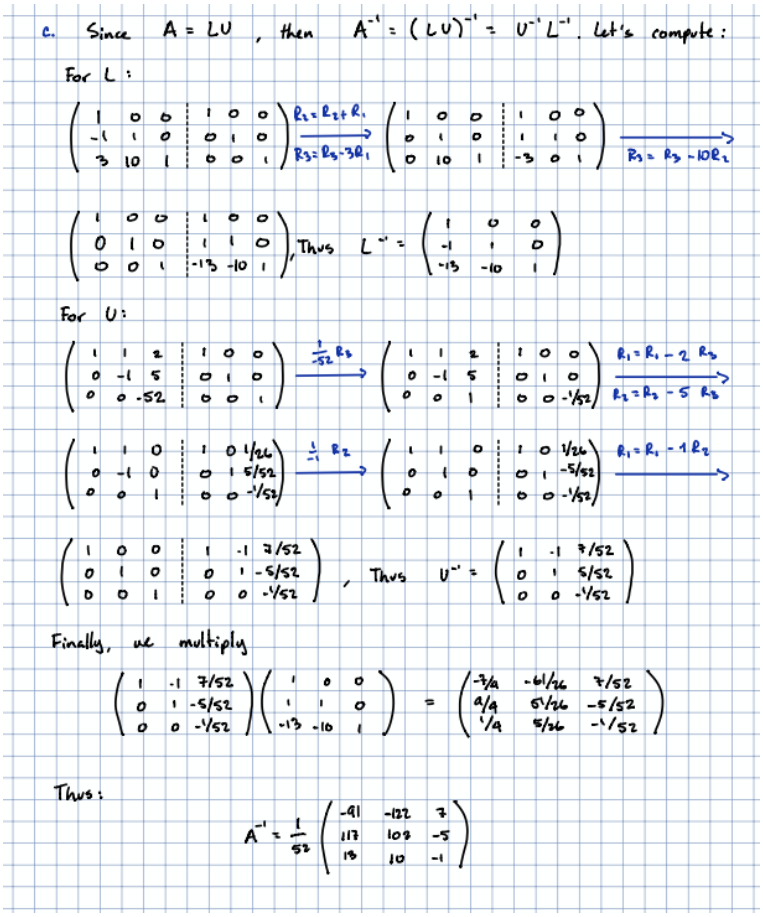
(b) Use Gaussian elimination to find the $PA = LU$ factorization of A (by hand). Which built-in MATLAB function can be used to determine this factorization?
Answer: The MATLAB function that performs the $PA = LU$ factorization is `[L,U,P] = lu(A)`. It is important include the `[L,U,P]` at the start, as this function can also be used for other matrix factorizations.



(c) Use your factorization from 1a to determine A^{-1}. (*Hint: Recall that* $A * A^{-1} = A * X = I$.) Discuss one benefit of using the factorization for this problem. (*Hint: The next problem might give you some ideas.*)

Answer: Given a coefficient matrix A of dimensions $n \times n$ and a problem $Ax = b$, we know that computing the LU factorization facilitates the process of getting a solution. Namely, the solution comes by computing L and U, then solving $Ly = b$ using forward substitution, and finally solving $Ux = y$ using backward substitution.

Our problem here is to determine A^{-1}. We know that A^{-1} is the matrix X that would satisfy $A * X = I$, where I is the identity matrix. If we separate both the X and I matrices into their respective columns, then we are left with n problems of the form $Ax_k = i_k$, where x_k and i_k are the kth columns of X and I respectively. Here, we may use the LU factorization to find all the x_k solutions using only forward and backward substitutions as stated before, and put all the x_k vectors as the columns for A^{-1}. The main benefit of this approach is that only forward and backward substitutions are needed after computing the LU factorization, making the computation less intensive.



2. In this problem we consider solving $Ax = b_k$ for a given $n \times n$ matrix A, and multiple right hand side vectors, $b_1, b_2, \ldots, b_m$. We will assume these vectors are stored in an $n \times m$ matrix B; that is,

$$B = \begin{bmatrix} b_1 & b_2 & \cdots & b_m \end{bmatrix}.$$

We could:

- Use Gaussian elimination to solve each system $Ax = b_k$, or
- first compute the $PA = LU$ factorization, and then compute the solutions by using only backward and forward solves.

The purpose of this problem is to compare the time it takes for these two approaches. Here is code to illustrate how to do this comparison with 50 right hand side vectors:

```
function [t1, t2] = CompareSpeed(n)
% (add comments here)
n_rhs = 50; n_runs = 10;
A  = rand(n); B = rand(n,n_rhs);
t1_sum = 0;
for j = 1:n_runs
    tic
    for k = 1:n_rhs
        x = A\B(:,k);
    end
    t1_sum = t1_sum + toc;
end
t1 = t1_sum/n_runs;

t2_sum = 0;
for j = 1:n_runs
    tic
    [L, U, P] = lu(A);
    for k = 1:n_rhs
        x = U\(L\(P*B(:,k)));
    end
    t2_sum = t2_sum + toc;
end
t2 = t2_sum/n_runs;
```

(a) Implement and run the above function, and test for $n = 50$.
Answer: The script returned a value of 0.0051 for `t1` and a value of $6.3567e-04$ for `t2`.

(b) Add a few lines of comments (see the line `(add comments here)`) to explain the purpose of this function, inputs and outputs.

(c) What is the purpose of the loop `for j = 1:n_runs`?
Answer: The purpose of those outer loops with `for j = 1:n_runs` is to solve the same linear system of equations a number of `n` times in order to get a better approximation of the time that it actually takes to perform these two approaches. If we were to only do it once, then our times would likely be very imprecise, as the exact time that it takes for the computer to perform any such algorithm varies because of the other tasks that are being executed by the computer. Thus, to get

a better approximation, we should run the same algorithms multiple times and take the averages.

(d) Write a script that will run the function for $n = 50, 100, 200, 400, 800, 1600$, and plot the timings using both `plot` and `semilogy`. Your plots should have matrix size, n, on the x-axis, and time to solve on the y-axis. That is, your plots should look similar to the following, but you should add a legend to each of the plots to indicate which approach is associated with the blue curve, and which is associated with the red.