Homework 4 (Solution)

Math 3607

Summer 2021

Tae Eun Kim

Table of Contents


```
clear
```

Problem 1 (Improved Triangular Substitutions)

Note. All functions written for this problem (`backsub`, `forelim`, and `ltinverse`) are all found at the very last section titled "Functions Used". This way, they can be called anywhere in the live script especially for the sake of testing. However, they are also included below in a Code Example environment for your convenience.

(a) The code provided in lecture can be easily improved to handle multiple triangular systems at once.

Backward Substitution (multiple systems)

The following version can handle $UX = B$ where $B \in \mathbb{R}^{n \times p}$, which demands $X \in \mathbb{R}^{n \times p}$ as well. Pay attention to how the added dimensions in X and B are handled in the code below.

```
function X = backsub(U,B)
% BACKSUB X = backsub(U,B)
% Solve multiple upper triangular linear systems.
% Input:
%   U    upper triangular square matrix (n by n)
%   B    right-hand side vectors concatenated into an (n by p) matrix
% Output:
%   X    solution of UX=B (n by p)
    [n,p] = size(B);
    X = zeros(n,p);       % preallocate
```

```
        for j = 1:p           % iterate over columns
            for i = n:-1:1    % plain back. subs.
                X(i,j) = ( B(i,j) - U(i,i+1:n)*X(i+1:n,j) ) / U(i,i);
            end
        end
    end
```

Note that the function can handle a single system $U\mathbf{x} = \mathbf{b}$ without any issue using the same syntax as the original version. So we can use it to fully replace the original version presented in class, hence no change in the function name.

Forward Elimination (multiple systems)

Similar modification is made below for forward elimination.

```
function X = forelim(L,B)
% FORELIM X = forelim(L,B)
% Solve multiple lower triangular linear systems.
% Input:
%   L    lower triangular square matrix (n by n)
%   B    right-hand side vectors concatenated into an (n by p) matrix
% Output:
%   X    solution of LX=B (n by p)
    [n,p] = size(B);
    X = zeros(n,p);      % preallocate
    for j = 1:p          % iterate over all columns
        for i = 1:n      % plain for. elim.
            X(i,j) = ( B(i,j) - L(i,1:i-1)*X(1:i-1,j) ) / L(i,i);
        end
    end
end
```

(b) The full program `ltinverse` is listed at the end of the file. The key segment of the program is just one line

```
X = forelim(L, eye(size(L)));
```

Note how directly the code translates the mathematical description of the problem of solving $LX = I$ to fine the inverse $X = L^{-1}$ using forward elimination. One more to note is that `eye(size(L))` generates the identity matrix that is of the same size as L, which reads very naturally.

Testing 1. Using L_1:

```
L1 = [2 0 0;
      8 -7 0;
      4 9 -27];
L1inv = [1/2 0 0;                % analytical inverse
         4/7 -1/7 0;
         50/189 -1/21 -1/27];  % numerical inverse
L1invNum = ltinverse(L1);
```

One could output `L1inv` and `L1invNum` and compare them entry by entry. Another way, which is almost always preferred, is to calculate the norm of the difference of the two matrices. A small norm means a small difference.

```
norm(L1inv - L1invNum)
```

```
ans =
     0
```

Looking good!

Testing 2. We proceed in the same fashion as above, now using L_2.

```
L2 = [1 0 0 0;
      1/3 1 0 0;
      0 1/3 1 0;
      0 0 1/3 1];
L2inv = [1 0 0 0;
         -1/3 1 0 0;
         1/9 -1/3 1 0;
         -1/27 1/9 -1/3 1];
L2invNum = ltinverse(L2);
norm(L2inv - L2invNum)
```

```
ans =
     0
```

Fantastic!

Further Demonstration

Here is an additional demonstration of testing your code. Unlike the previous ones, I will compare the code `ltinverse` against MATLAB's built-in solver. For this demo, I will use a 10×10 lower triangular matrix created by

```
n = 10;
L = tril(magic(n))
```

```
L = 10×10
    92     0     0     0     0     0     0     0     0     0
    98    80     0     0     0     0     0     0     0     0
     4    81    88     0     0     0     0     0     0     0
    85    87    19    21     0     0     0     0     0     0
    86    93    25     2     9     0     0     0     0     0
    17    24    76    83    90    42     0     0     0     0
    23     5    82    89    91    48    30     0     0     0
    79     6    13    95    97    29    31    38     0     0
    10    12    94    96    78    35    37    44    46     0
    11    18   100    77    84    36    43    50    27    59
```

Inspection of Residuals

We can find its inverse by asking MATLAB to solve $LX = I$ using the backslash, which is essentially what happens when `inv` is called.

```
Linv = L\eye(n);    % or Linv = inv(L);
```

Big Question. We do not know the true inverse. How can we check the computed result?

Since we do not know the true analytic inverse, we cannot calculate the (*forward*) error associated with `Linv`, and so it is hard to say whether the result is good or bad. However, if MATLAB did a good job, then it must be the case that `L*Linv` is very close to the identity. In other words, `L*Linv - eye(n)` must be really close to the zero matrix. This difference $LX - I$ of the two sides is called the *residual* and it is an example of *backward* error; see Section 1.3 of **FNC** for more information.

```
norm(L*Linv-eye(n))
```

```
ans =
      1.99225401607197e-15
```

The norm of the residual is at the level of machine epsilon.

Likewise, let's calculate the inverse using our code and compute the norm of the residual:

```
myLinv = ltinverse(L);
norm(L*myLinv - eye(n))
```

```
ans =
      3.00841133167772e-15
```

This looks good as well.

As seen above,

Residuals provide one way of checking the computation when the true answer is unknown.

Lastly, let's see how the two results compare:

```
norm(Linv - myLinv)
```

```
ans =
      7.90892925731122e-17
```

The two results show a very good agreement. If one is certain that one of the two is highly accurate, then this confirms that the other is also an accurate computation of the inverse.

Problem 2 (PLU Factorization)

Below is a solution.

```
function [L,U,P] = myplu(A)
% MYPLU   PLU factorization code (instructional version)
% Input:
%   A = square matrix
% Output:
%   P,L,U = permutation, unit lower triangular, and upper triangular
%           matrices such that P*A = L*U.
    n = length(A);
    P = eye(n);   % preallocate P
    L = eye(n);   % preallocate L
    % A will be overwritten below to be U

    for j = 1:n-1

        [~, iM] = max(abs(A(j:n, j)));   % find the index of pivot element
        iM = iM + j - 1;                 % adjust the index

        if j ~= iM                       % row interchange if necessary
            P([j iM], :) = P([iM j], :); % update P
            A([j iM], :) = A([iM j], :); % update A
            L(:, [j iM]) = L(:, [iM j]); % update L by emulating P*G*P
            L([j iM], :) = L([iM j], :); %    where P is an elem. perm. mat.
        end

        for i = j+1:n                    % introduce zeros below diagonal
            L(i,j) = A(i,j) / A(j,j);    % row multiplier
            A(i,j:n) = A(i,j:n) - L(i,j)*A(j,j:n); % row replacement
        end

    end
    U = triu(A);  % to ensure that U come out as a clean upper-trian. mat.
end
```

As an instructional version, it is not the most compact or the most optimized code. See, for instance, the following code from Section 2.7 of **NCM**, written by Moler:

```
function [L,U,p] = lutx(A)
%LUTX  Triangular factorization, textbook version
%   [L,U,p] = lutx(A) produces a unit lower triangular matrix L,
%   an upper triangular matrix U, and a permutation vector p,
%   so that L*U = A(p,:)

%   Copyright 2014 Cleve Moler
%   Copyright 2014 The MathWorks, Inc.

    [n,n] = size(A);
    p = (1:n)';

    for k = 1:n-1

        % Find index of largest element below diagonal in k-th column
        [r,m] = max(abs(A(k:n,k)));
        m = m+k-1;
```

```
            % Skip elimination if column is zero
            if (A(m,k) ~= 0)

                % Swap pivot row
                if (m ~= k)
                    A([k m],:) = A([m k],:);
                    p([k m]) = p([m k]);
                end

                % Compute multipliers
                i = k+1:n;
                A(i,k) = A(i,k)/A(k,k);

                % Update the remainder of the matrix
                j = k+1:n;
                A(i,j) = A(i,j) - A(i,k)*A(k,j);
            end
        end

        % Separate result
        L = tril(A,-1) + eye(n,n);
        U = triu(A);
    end
```

The key difference is that `lutx` produces a permutation vector `p` in lieu of a permutation matrix `P`. It is highly recommended that you read this code carefully line by line.

Testing. We will test the code using a randomly generated matrix A. First, since $PA = LU$, we will compute the norm of the residual, $\|PA - LU\|$, and see if it is small.

```
%rng(1234)
n = 500; A = randn(n);
[L,U,P] = myplu(A);
norm(P*A-L*U)
```

```
ans =
     1.66009400631911e-13
```

Next, we will compare the results against the ones obtained by MATLAB's built-in `lu` function.

```
[L1,U1,P1] = lu(A);
disp([norm(L-L1), norm(U-U1), norm(P-P1)])
```

```
     3.72683670014656e-13     1.06576354741282e-11                        0
```

Problem 3 (Determinant)

(a) Recall from linear algebra that $\det(LU) = \det(L)\det(U)$. In addition, recall that the determinant of a triangular matrix is the product of its diagonal entries. Since L in the LU factorization is a unit lower triangular matrix, it follows that

$$\det(A) = \det(LU) = \det(L)\det(U) = (1 \cdot 1 \cdots 1)(u_{11}u_{22}\cdots u_{nn}) = u_{11}u_{22}\cdots u_{nn}.$$

(b) The previous part suggests that we can write the following program computing the determinant of a given matrix:

```
function D = determinant(A)
    [~,U] = mylu(A);
    D = prod(diag(U));
end
```

Since L is not needed in the computation, the LU factorization routine is called with `[~,U] = mylu(A)`. Let's test it.

```
for n = 3:7
    A = magic(n);
    myDet = determinant(A);
    Det = det(A);
    if n == 3      % print header
        fprintf(' %2s %20s %20s %12s\n', 'n', 'myDet', 'Det', 'rel. error')
        fprintf(' %57s\n', repmat('-', 1, 57))
    end
    fprintf(' %2d %20.8g %20.8g %12.4g\n', n, myDet, Det, abs((myDet-Det)/Det))
end
```

```
 n                myDet                  Det    rel. error
 ---------------------------------------------------------
 3                 -360                 -360             0
 4         3.623768e-13       -1.4495072e-12          1.25
 5              5070000              5070000     7.348e-16
 6                    0       -8.0494971e-09             1
 7        -3.480528e+11        -3.480528e+11     3.507e-16
```

Problem 4 (LM 10.1--12a,b,d)

All vectors appearing are in $\mathbb{R}^n$; all matrices are in $\mathbb{R}^{n\times n}$.

(a) We calculate $\mathbf{x} = ABCD\mathbf{b}$ as below

```
x = A * (B * (C * (D * b)))
```

What MATLAB does is:

1. form a vector `D*b`: $\sim 2n^2$ flops
2. left-multiply the previous result by `C`: $\sim 2n^2$ flops
3. left-multiply the previous result by `B`: $\sim 2n^2$ flops

4. left-mulitply the previous result by `A`: $\sim 2n^2$ flops

In total, it takes $\sim 8n^2$ flops to calculate `x`.

If it were to be computed by, say,

```
x = (A * (B * (C * D))) * b
```

the computation of `C*D` itself takes $\sim 2n^3$ flops already.

Lesson. Try to avoid (matrix) × (matrix) multiplication if possible!

(b) To compute $\mathbf{x} = BA^{-1}\mathbf{b}$ efficiently, do

```
x = B * (A \ b);
```

This way:

1. calculate $A^{-1}\mathbf{b}$ using backslash operator `A\b`: $\sim \frac{2}{3}n^3$ flops (because `\` does a pivoted GE in general)
2. left-multiply the previous result by `B`: $\sim 2n^2$ flops (since it is a (matrix) × (vector) multiplication.)

So, all in all, it take $\sim \frac{2}{3}n^3$ flops. (Recall that we only retain the dominant term in the asymptotic notation.)

(d) To efficiently compute $\mathbf{x} = B^{-1}(C + A)\mathbf{b}$, do

```
x = B \ ((C + A) * b)
```

By doing this:

1. form `C + A`: $\sim n^2$ flops
2. multiply it by `b`: $\sim 2n^2$ flops
3. left-"divide" by `B` using `\` (pivoted GE): $\sim \frac{2}{3}n^3$ flops

Altogether, it takes $\sim \frac{2}{3}n^3$ flops.

Problem 5 (LM 10.2--1)

(a) **No**. The norm of a matrix A is 0 if and only if A is the zero matrix by definition.

(The following is an extra explanation. The previous sentence is good enough justification for your submission.) Any nontrivial singular matrix has a positive norm, *e.g.*,

```
A = [1 0;
     0 0]
```

```
A = 2×2
     1     0
     0     0
```

is evidently singular ($\det A = 0$), but its norm is 1.

```
norm(A, 1)
```

```
ans =
     1
```

```
norm(A, 2)
```

```
ans =
     1
```

```
norm(A, Inf)
```

```
ans =
     1
```

```
norm(A, 'fro')
```

```
ans =
     1
```

(b) **Yes**. The alternate definition given in (10.20) on p. 1481 is useful when A^{-1} does not exists, *i.e.,* when A is singular. Suppose A is singular. Then the equation $A\mathbf{x} = \mathbf{0}$ has a nontrivial solution $\mathbf{x}$ and so $\min_{\|\mathbf{x}\|_p=1} \|A\mathbf{x}\|_p = 0$.

Therefore, in light of the alternate definition, the condition number of a singular matrix is infinite. MATLAB adheres to this convention:

```
cond(A)
```

```
ans =
   Inf
```

(c) **Yes**, because

$$\kappa_p(A) = \|A\|_p \|A^{-1}\|_p = \|(A^{-1})^{-1}\|_p \|A^{-1}\|_p = \kappa_p(A^{-1}).$$

(d) **No.** This was explained in lecture. The essense of the argument was that

$$1 = \|I\|_p = \|AA^{-1}\|_p \leq \|A\|_p \|A^{-1}\|_p = \kappa_p(A).$$

(e) The last bullet point on p. 1481 is helpful here which states that

$$\kappa_2(A) = \sqrt{\frac{\lambda_{\max}(A^{\mathrm{T}}A)}{\lambda_{\min}(A^{\mathrm{T}}A)}}.$$

In case A is symmetric, the above reduces to

$$\kappa_2(A) = \frac{\max\limits_i |\lambda_i|}{\min\limits_i |\lambda_i|}.$$

Exercise. Confirm it!

Let's use the second formulation to construct a matrix $A \in \mathbb{R}^{10\times 10}$ with the desired properties. To keep things simple, take A not just symmetric, but diagonal as in

$$A = \begin{bmatrix} d_1 & & & \\ & d_2 & & \\ & & \ddots & \\ & & & d_{10} \end{bmatrix}, \text{ with } d_j\text{'s all positive.}$$

We know from linear algebra that $\det A = d_1 d_2 \cdots d_{10}$ and the diagonal entries $d_1, d_2, \ldots, d_{10}$ are its eigenvalues. Since we want $\kappa_2(A) = 1$, we need all d_j's to have the same (absolute) value, say d. Then $\det A = d^{10} = 10^{-20}$, which implies that $d = 10^{-2}$. Therefore,

$$A = \begin{bmatrix} 10^{-2} & & & \\ & 10^{-2} & & \\ & & \ddots & \\ & & & 10^{-2} \end{bmatrix} = 10^{-2}\, I,$$

will do. Let's confirm this on MATLAB:

```
A = 1e-2*eye(10);
det(A)
```

```
ans =
                   1e-20
```

```
cond(A)
```

```
ans =
     1
```

Functions Used

Backward Substitutions

```
function X = backsub(U,B)
% BACKSUB X = backsub(U,B)
```

```
% Solve multiple upper triangular linear systems.
% Input:
%   U    upper triangular square matrix (n by n)
%   B    right-hand side vectors concatenated into an (n by p) matrix
% Output:
%   X    solution of UX=B (n by p)
    [n,p] = size(B);
    X = zeros(n,p); % preallocate
    for j = 1:p
        for i = n:-1:1
            X(i,j) = ( B(i,j) - U(i,i+1:n)*X(i+1:n,j) ) / U(i,i);
        end
    end
end
```

Forward Elimination

```
function X = forelim(L,B)
% FORELIM X = forelim(L,B)
% Solve multiple lower triangular linear systems.
% Input:
%   L    lower triangular square matrix (n by n)
%   B    right-hand side vectors concatenated into an (n by p) matrix
% Output:
%   X    solution of LX=B (n by p)
    [n,p] = size(B);
    X = zeros(n,p); % preallocate
    for j = 1:p
        for i = 1:n
            X(i,j) = ( B(i,j) - L(i,1:i-1)*X(1:i-1,j) ) / L(i,i);
        end
    end
end
```

Inverse of a lower triangular matrix

```
function X = ltinverse(L)
% LTINVERSE X = ltinverse(L)
% Find the inverse of a lower triangular matrix.
% Input:
%   L    lower triangular square matrix (n by n)
% Output:
%   X    inverse of L, i.e., LX = I. (n by n)
    if ~istril(L) || diff(size(L))~=0 % if input is not lower triangular nor square
        error('The input must be a lower triangular square matrix');
    end
    X = forelim(L, eye(size(L)));
end
```

PLU factorization

```
function [L,U,P] = myplu(A)
    n = length(A);
    P = eye(n);   % preallocate P
```

```
    L = eye(n);    % preallocate L
    % A will be overwritten below to be U

    for j = 1:n-1

        [~, iM] = max(abs(A(j:n, j)));    % find the index of pivot element
        iM = iM + j - 1;                  % adjust the index

        if j ~= iM                        % row interchange if necessary
            P([j iM], :) = P([iM j], :); % update P
            A([j iM], :) = A([iM j], :); % update A
            L(:, [j iM]) = L(:, [iM j]); % update L by emulating P*G*P
            L([j iM], :) = L([iM j], :); %    where P is an elem. perm. mat.
        end

        for i = j+1:n                     % introduce zeros below diagonal
            L(i,j) = A(i,j) / A(j,j);     % row multiplier
            A(i,j:n) = A(i,j:n) - L(i,j)*A(j,j:n); % row replacement
        end

    end
    U = triu(A);  % to ensure that U come out as a clean upper-trian. mat.
end
```

Determinant based on LU factorization

```
function D = determinant(A)
% DETERMINANT D = determinant(A)
% Calculate the determinant of A using LU factorization.
% Input:
%   A   square matrix
% Output:
%   D   determinant of A
% Dependency: mylu (see below)
    [~,U] = mylu(A);
    D = prod(diag(U));
end
```

LU factorization routine

This is needed for the function `determinant`.

```
function [L,U] = mylu(A)
    n = length(A);
    L = eye(n);
    for j = 1:n-1
        for i = j+1:n
            L(i,j) = A(i,j) / A(j,j);
            A(i,j:n) = A(i,j:n) - L(i,j)*A(j,j:n);
        end
    end
    U = triu(A);
end
```