Homework 5 (Solution)

Math 3607

Tae Eun Kim

Table of Contents


```
clear
```

Problem 1 (Triangular Substitutions)

(a) The codes provides 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
        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
```

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.

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
        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
```

(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)));
```

and 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.

I will compare the code with MATLAB's built-in solver. (Note that this is different from how the question asked to confirm the code.)

```
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
```

We can find its inverse by asking MATLAB to solve $LX = I$ using the backslash:

```
Linv = L\eye(n);
```

The residual $LX - I$ has a small norm

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

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

at the level of machine epsilon.

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.

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

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

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

As expected, the two results show a very good agreement.

Problem 2 (PLU Factorization)

G.E. with partial pivoting.

This was one of the exercises in the lecture. This provides a good starting point for PLU factorization code.

```
function x = GEpp(A, b)
% GEPP pivoted Gaussian Elimination (instructional version)
% Input:
%   A = square matrix
%   b = right-hand side vector
% Output:
%   x = vector solving A*x = b
    % row reduction to upper triangular system
    S = [A, b];                         % augmented matrix
    n = size(A, 1);
    for j = 1:n-1
        [~, iM] = max(abs(A(j,j:end))); % index of pivot element
        iM = iM + j - 1;                % adjusted index
        if j ~= iM                      % pivot if necessary
```

```
            piv = [j iM];
            S(piv, :) = S(flip(piv), :).
        end
        for i = j+1:n
            mult = -S(i,j)/S(j,j);      % R_i --> R_i + (-S(i,j)/S(j,j))*R_j
            S(i,:) = S(i,:) + mult*S(j,:);
        end
    end
    % back subs (pretending that there is no 'backsub' routine)
    U = S(:,1:end-1);
    beta = S(:,end);
    x(n) = beta(n)/U(n,n);
    for i = n-1:-1:1
        x(i) = (beta(i) - U(i,i+1:end)*x(i+1:end))/U(i,i);
    end
end
```

Note that it is written as a self-contained program which contains a hard-coded backward substitution routine. One can confirm with the knowledge of Gaussian transformation matrices that the multiplier $a_{i,j}/a_{j,j}$ is directly related to the construction of L matrix. See how it is coded below.

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)));
        iM = iM + j - 1;                % adjustment
        if j ~= idx_adj
            piv = [j, iM];              % pivot
            P(piv, :) = P(flip(piv), :); % update permutation matrix
            A(piv, :) = A(flip(piv), :); % update A
            L(:, piv) = L(:, flip(piv)); % update L by emulating P*G*P
            L(piv, :) = L(flip(piv), :); % where P is an elem. perm. mat.
        end
        for i = j+1:n
            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);
        end
    end
    U = triu(A);  % to ensure that U come out as a clean upper-trian. mat.
end
```

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
        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--8)

Let $P_{1,4}$ be the 5-by-5 elementary permutation matrix obtained by interchanging its 1st and 4th rows:

$$P_{1,4} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

(a) $P_{1,4}A$

(b) $AP_{1,4}$

(c) $P_{1,4}AP_{1,4}$

(d) By carrying out the described column operations onto I, we obtain the following permutation matrix, call it Q:

$$Q = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

To reiterate the same column operations onto A, multiply Q to its right: AQ. Note that each time Q is multiplied to a matrix (from the right), it shifts first four columns to the left in a cyclic fashion while leaving the fifth one intact. Hence, 4 repeated multiplication by Q will leave A unchanged, in other words, $Q^4 = I$. Note that $Q^k = I$ for any k that is a multiple of 4.

The action of Q can be decomposed into:

1. Swap 1st and 2nd columns. → column order: (2,1,3,4,5)
2. Swap 2nd and 3rd columns. → column order: (2,3,1,4,5)
3. Swap 3rd and 4th columns. → column order: (2,3,4,1,5)

It follows that $Q = P_{1,2}P_{2,3}P_{3,4}$. The order is important here.

```
I = eye(5);
P12 = I(:,[2 1 3 4 5]);
P23 = I(:,[1 3 2 4 5]);
P34 = I(:,[1 2 4 3 5]);
Q = P12*P23*P34
```

```
Q = 5×5
     0     0     0     1     0
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     0     1
```

```
R = P34*P23*P12
```

```
R = 5×5
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     1     0     0     0     0
     0     0     0     0     1
```

Note that AQ cycles first four columns of A

```
A = reshape(1:25,5,5);
A*Q
```

```
ans = 5×5
     6    11    16     1    21
     7    12    17     2    22
     8    13    18     3    23
     9    14    19     4    24
    10    15    20     5    25
```

while RA cycles first four rows of A

```
R*A
```

```
ans = 5×5
     2     7    12    17    22
     3     8    13    18    23
     4     9    14    19    24
     1     6    11    16    21
     5    10    15    20    25
```

Problem 5 (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 already take $\sim 2n^3$ 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) $\times$ (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 6 (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

$$\kappa_2(A) = \frac{\max_i |\lambda_i|}{\min_i |\lambda_i|}.$$

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
```

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

```
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
```