CS513, Spring 2007
Prof. Amos Ron
Assignment #5 Solutions
Prepared by Houssain Kettani

1 Question 1

1. Let a_{ij} denote the entries of the matrix A. Let $A^{(m)}$ denote the matrix A after m Gaussian elimination operations. Denote the entries of A^m by $a_{ij}^{(m)}$. Thus,

$$a_{ij}^{(1)} = \begin{cases} a_{ij} & i = 1, 1 \leq j \leq n \\ 0 & j = 1, 2 \leq i \leq n \\ a_{ij} - \frac{a_{i1}}{a_{11}} a_{1j} & i, j \in [2, n] \end{cases}$$

Since A is symmetric, $a_{ij} = a_{ji}$. Thus, $a_{ij}^1 = a_{ji}^1$ for $i, j \in [2, n]$. Therefore, the submatrix is symmetric after 1 Gaussian elimination operation. Assume that the submatrix $a_{ij}^{(m-1)}$, $i, j \in [m, n]$ is symmetric, where $A^{(m-1)}$ is A after $m-1$ operations. Thus,

$$a_{ij}^{(m)} = \begin{cases} a_{ij}^{(m-1)} & i \in [1, m], j \in [1, n] \\ 0 & j \in [1, m], i \in [m+1, n] \\ a_{ij}^{(m-1)} - \frac{a_{im}^{(m-1)}}{a_{mm}^{(m-1)}} a_{mj} & i, j \in [2, n] \end{cases}$$

Since the submatrix $a_{ij}^{(m-1)}$, $i, j \in [m, n]$ is symmetric, $a_{ij}^{(m-1)} = a_{ji}^{(m-1)}$, $i, j \in [m, n]$. Thus, $a_{ij}^{(m)} = a_{ji}^{(m)}$ for $i, j \in [m+1, n]$. Therefore, the submatrix, $a_{ij}^{(m)}$, $i, j \in [m+1, n]$ is symmetric after m Gaussian elimination operation.
By induction, a symmetric matrix A has a symmetric submatrix $a_{ij}^{(m)}$, $i, j \in [m+1, n]$.

2. Since the submatrix we use in the Gaussian elimination operation is symmetric, instead of processing all of the elements in each row of the submatrix, we only need to process the elements above the diagonal. That is, the computation $L_{ij} = \frac{a_{ij}}{a_{jj}}$, performed for $i \in [j+1, n]$ on the j^{th} operation, can be rewritten as $L_{ij} = \frac{a_{ji}}{a_{jj}}$. The first form requires that an element below the diagonal be accessed, but the second form accesses only elements of a_{ij} that are above the main diagonal.
The next step of the computation, performed for each L_{ij}, is to set $a_{ik} = a_{ik} - L_{ij} a_{jk}$, for $k \in [j+1, n]$. If $i > k$, we would be changing elements below the diagonal. But since we do not need to access elements below the diagonal to compute the L_{ij} for the matrix L, we have no reason to change those elements. Thus, we can leave elements of A below the diagonal alone, and only change the upper triangular elements. The upper triangle after all the operations is the matrix U.
The following MATLAB function is developed for this purpose:

```
function [L,U] = sym_lu(A)
% This function performs LU factorization for
% a symmetric matrix A.  It returns the lower
% and upper triangular matrices as two separate
% matrices, to make checking easier.  The lower
% triangle of A is never accessed.  The matrix
% must be nonsingular.

% We will need to know the dimension of the
% symmetric matrix.
dim = size(A,1);

% Initialize L to the identity matrix; the lower
% elements will be filled in.
L = eye(dim);

% For each row in A,
for i=1:dim-1,

   % For each row under row i,
   for j=i+1:dim,

      % Check to see if we will encounter divide by zero.
      if abs(A(i,i)) <= 1e-12
         disp('The matrix is singular.')
         L=NaN;
         A=NaN;
         return;
      end

      % Compute the factor and store it in L(j,i).
      % In the ordinary LU algorithm,
      % L(j,i) = A(j,i) / A(i,i).
      % But this algorithm cannot access A(j,i) since it
      % cannot accss the lower triangle.  Since A is
      % symmetric, A(j,i) = A(i,j) even after a number of
      % elimination operations.
      L(j,i) = A(i,j) / A(i,i);

      % Multiply the upper triangular part of row i
      % by the factor.  Subtract this result from the
      % upper triangular part of row j.
      A(j,j:dim) = A(j,j:dim) - A(i,j:dim).*L(j,i);

  end
```

```
end

% The U factor is the upper triangle of A, with zeros
% in the lower triangle.
U = triu(A);
```

3. We use the number of flops to check the complexity of the previous algorithm. Let N denote the total number of flops. Then

$$N = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (1 + \sum_{k=j}^{n} 1),$$

where the first and second summations correspond to the first and second loops, respectively. The third summation corresponds to the last line in the second loop. Therefore,

$$N = \frac{n^3}{6} + \frac{n^2}{2} - \frac{2n}{3}$$

4. The preceding equation can easily be verified numerically. In addition, a plot of $\frac{flops(symm)}{flops(asymm)}$ shows that term converges to 0.5. This makes sense since flops(symm) is O($\frac{n^3}{6}$) and flops(asymm) is O($\frac{n^3}{3}$).

2 Question 2

d

1. The following MATLAB file is developed to perform LU factorization without pivoting.

```
function [L,U] = lu_np(A)
% This function performs LU factorization for
% a matrix A. The function returns the lower
% and upper triangular matrices as separate
% matrices to make checking easier. The matrix
% must be nonsingular and square.

% We will need to know the dimension of the matrix.
dim = size(A,1);

% Initialize L to the identity matrix.
L = eye(dim);
```

```
% For each row in A,
for i=1:dim-1,

   % For each row under row i,
   for j=i+1:dim,

      % Check to see if we will encounter divide by zero.
      if abs(A(i,i)) <= 1e-14
         disp('The matrix is singular.')
         U = NaN;
         L = NaN;
         return;
      end

      % Compute the factor.
      L(j,i) = A(j,i) / A(i,i);

      % Multiply the "nonzero" elements of row i by the
      % factor.  Subtract this result from the "nonzero"
      % elements of row j.
      A(j,i+1:dim) = A(j,i+1:dim) - A(i,i+1:dim).*L(j,i);

  end

end

% The U factor is the upper triangle of A, with zeros
% in the lower triangle.
U = triu(A);
```

We could also use the MATLAB function [L,U] = lu(A), so that $A = LU$.

2. For LU-factorization using partial pivoting, we use the MATLAB function: [L,U,P] = lu(A), where P is the permutation matrix, such that $PA = LU$.

3. For LU-factorization using full pivoting, the following MATLAB code does the job:

```
function [L,U,P,Q] = lu_fp(A)
% This function performs LU factorization for a matrix A. The function
% returns the lower and upper triangular matrices as separate matrices
% to make checking easier.  It also returns matrices P and Q indicating
% the row  and column exchanges, respectively. The function  uses full
```

```
% pivoting, such that  PAQ = LU.  The matrix must be nonsingular and square.

% We will need to know the dimension of the matrix A.
dim = length(A);

% Initialize P and Q to the identity matrices.
P = eye(dim);
Q = eye(dim);


% For each row in A,
for i=1:dim-1,

   % Find the element with largest magnitude in each
   % submatrix  which will be the new pivot.
   pivot = max(max(abs(A([i:dim],[i:dim]))));

  % find the indeces of the new pivot
  [x,y] = find(abs(A([i:dim],[i:dim])) == pivot);
  if i~=1;
    x(1) = x(1) + (i-1);
    y(1) = y(1) + (i-1);
  end;

  % interchange the rows and columns of the new pivot
  % with the old one
  A([i,x(1)],:) = A([x(1),i],:);
  A(:,[i,y(1)]) = A(:,[y(1),i]);

  % store the changes in the matrices P and Q
  P([i,x(1)],:) = P([x(1),i],:);
  Q(:,[i,y(1)]) = Q(:,[y(1),i]);

  % Compute the factor.
  A(i+1:dim,i) = A(i+1:dim,i) / A(i,i);

  % Multiply the "nonzero" elements of row i by the
  % factor.  Subtract this result from the "nonzero"
  % elements of row j.
  A(i+1:dim,i+1:dim) = A(i+1:dim,i+1:dim) - A(i,i+1:dim)*A(i+1:dim,i);
  end

   % For each row under row i,
   for j=i+1:dim,
```

```
    % Check to see if we will encounter divide by zero.
   if abs(A(i,i)) <= 1e-12
     disp('The matrix is singular.')
      U = NaN;
      L = NaN;
      return;
   end
 end


% The U factor is the upper triangle of A, with zeros
% in the lower triangle.
U = triu(A);

% The L factor is the lower triangle of A, with zeros
% in the lower triangle and ones along the main diagonal.
L = tril(A,-1) + eye(dim);
```

The rest of MATLAB files needed to answer this question were straight forward. It is obvious that the complexity goes higher in the following order: no-pivoting, partial-pivoting, then full-pivoting.

Let A_1 denote the matrix $a_{ij} = \frac{1}{i+j-1}$. Let A_2 denote the matrix $a_{ij} = i^{j-1}$. Note that A_1 can be constructed using the MATLAB function `hilb`. Note also that you were asked to construct A_2 in HW 3, using no `for` loops.

Know that if the matrix is ill-conditioned, then all the three algorithms will be unstable, thus will give big errors. If there exist any hope in a matrix, then the full-pivoting algorithm should be stable. In practice, full-pivoting is not used, since it does not give much stability improvement over partial-pivoting, let alone that its complexity is much higher.

For the matrices A_1 and A_2, many students made the analyses with small size matrices only, making them deduce that those matrices are well-conditioned! The fact is those matrices are very ill-conditioned. For such matrices, a size of 20x20 gives a huge condition number.

Finally, as far as random matrices were concerned, such matrices are almost always well-conditioned. This is due to the fact that the rows and columns of these matrices are very linearly independent, i.e. to create the matrix, the numbers are assigned at random; thus the probability that there are two rows or columns that are nearly multiples of each other is extremely low. Whenever this is the case, you may expect low condition numbers.