Course 6.336 - Introduction to Numerical Algorithms
(Fall 2003)
Solutions to Problem Set #3

Problem 3.1

a) When assuming a row-wise factorization of M, we can think of it as a column-wise factorization of M^T. This will give us a representation of the factorization in the following form:

$$M = LQ, \tag{1}$$

Where L is a lower-triangular matrix and Q is an orthogonal matrix. Therefore we can easily modify the *qr.m* code:

```
function [q,l] = qr_rowwise(A)
% Function for computing a ‘‘row-wise QR’’ factorization of the square matrix A
N = size(A,1);
% for each row of the source matrix...
for i = 1:N
    % normalize row of A and put it into Q(i,:)
    l(i,i) = norm(A(i,:));
    % what if l(i,i) == 0 ???
    q(i,:) = A(i,:)./l(i,i);
    % for each row below the current...
    for j = i+1:N
        l(j,i) = A(j,:)*q(i,:)';
        % orthogonalize A(j, :) with respect to the vector Q(i,:)
        A(j, :) = A(j,:) - l(j,i)*q(i,:);
    end;
end;
```

Now we can use this code to solve the system $Mx = b$:

```
% Solving system Mx=b using row-wise QR.
[Q,L] = qr_rowwise(M);
x = Q'*(L\b);
```

b) In the case of LU factorization of a tridiagonal matrix A, on each step we do the following:

1. Compute multiplier $m_{i+1,i} = \frac{a_{i+1,i}}{a_{i,i}}$ (one operation)

2. Update $\tilde{a}_{i+1,i+1} = a_{i+1,i+1} - m_{i+1,i} * a_{i,i+1}$ (one multiply-add operation[1])

We do this procedure $(N-1)$ times, therefore the total cost is about $2N$ operations.

Now, let's compute the cost of QR factorization for this case. I will refer to the original QR decomposition; the costs of our decomposition is the same as for original QR. Note that the matrix Q will posess the upper-triangular form with nonero lower sub-diagonal!

Our QR decomposition for the case of tridiagonal matrix will be:

- For each column i of the source matrix *($(N-1)$ cycles)*
  - Normalize this column *(the average cost is $N/2$ multiply-add operations[2] for obtaining norm plus $N/2$ to divide each element, because in average the column is half-full)*
  - For each column j to the right of the i^{th} column *(2 cycles needed, since all others will have zero projection!)*
    - Compute inner product of the i^{th} column and j^{th} column *(average $N/2$ multiply-add operations)*
    - Substract the projection from j^{th} column *(average $N/2$ multiply-add operations)*

The total cost, therefore, is roughly $3N(N+1) = 3N^2$ operations.

We can see that sparse QR factorization is more costly than sparse LU, and this is understandable, since QR produces dense matrix.

c) Let's assume that the initially norms of the matrix rows were equal. If during orthogonalization we substracted a significant part of some vector, this means that the number of significant digits in this vector has decreased. We normally do not want to minimize usage of this "unreliable" vector for subsequent orthogonalizations, therefore we need to process this vector last. We use the following strategy: on each step of orthogonalization search all rows to the bottom of current and find the row with maximal length, then swap it with current row. We need to swap corresponding rows in the matrix L as well. We can keep track of this pivoting by swapping the corresponding rows of a unity matrix (i.e. forming so called permutation matrix):

[1] Since the cost of operation is dictated mainly by memory access, therefore we count a multiply-add operation as one operation, HOWEVER, to count them separately is perfectly OK!

[2] Again, we can view this as $2N/2 = N$ ordinary operations

```
function [q,l, perm] = qr_rowwise_piv(A)
% Computing a row-wise QR factorization with pivoting
N = size(A,1);
l = zeros(N,N);
% permutation matrix, initially unity
perm = eye(N);
% for each row of the source matrix...
for i = 1:N

    % search to the bottom of the matrix for the row with biggest norm
    current_norm = norm(A(i,:));
    current_row = i;
    for row = i+1:N
        if (norm(A(row,:)) > current_norm)
            current_norm = norm(A(row,:));
            current_row = row;
        end;
    end;
    A = swap_rows(A, i, current_row);
    l = swap_rows(l, i, current_row);
    perm = swap_rows(perm, i, current_row);
    % end of pivoting!

    l(i,i) = norm(A(i,:));
    % what if l(i,i) == 0 ???
    q(i,:) = A(i,:)./l(i,i);
    for j = i+1:N   % for each row below the current...
        l(j,i) = A(j,:)*q(i,:)';
        %orthogonalize A(j, :) to Q(i,:)
        A(j, :) = A(j,:) - l(j,i)*q(i,:);
    end;
end;
```

The function *swap_rows* simply swaps two rows in the matrix:

```
function A = swap_rows(A, i, j)
temp_row = A (i, :);
A(i, :) = A(j, :);
A(j, :) = temp_row;
```

Now we have the following equality: $PA = LQ$, where P is the permutation matrix we have created. Therefore we can solve our system $Mx = b$

using the following code:

```
% Solving system Mx=b using row-wise QR with pivoting.
[Q,L,P] = qr_rowwise_piv(M);
x = Q'*(L\(P*b));
```

d) Now it's time to analyze the singular case. If the matrix M is singular, this means that it contain a linearly-dependent rows, therefore sooner or later we will find a row of A with zero norm (all entries are zero). We can deal with this by inserting zeros to the rows of the matrix Q and puting a unity onto the corresponding diagonal of L. If the system doesn't have a solution (i.e. the system is **overdetermined**), we will find something, which is NOT a least-squares solution by means of minimal residual! However, if we have an **underdetermined** system of equations $Mx = b$, which has an infinite number of solutions, we will get the solution x, which has smallest norm. This is another type of linear least-squares tasks, which we do not study in this class.

Now, the code for factorization:

```
function [q,l, perm] = qr_rowwise_piv(A)
% Computing a row-wise QR factorization with pivoting, singular case included
N = size(A,1);
l = zeros(N,N);
% permutation matrix, initially unity
perm = eye(N);
% for each row of the source matrix...
for i = 1:N
    % search to the bottom of the matrix for the row with biggest norm
    current_norm = norm(A(i,:));
    current_row = i;
    for row = i+1:N
        if (norm(A(row,:)) > current_norm)
            current_norm = norm(A(row,:));
            current_row = row;
        end;
    end;
    A = swap_rows(A, i, current_row);
    l = swap_rows(l, i, current_row);
    perm = swap_rows(perm, i, current_row);
    % end of pivoting!
    l(i,i) = norm(A(i,:));
```

```
    if (l(i,i) ~= 0)
        q(i,:) = A(i,:)./l(i,i);
        for j = i+1:N   % for each row below the current...
            l(j,i) = A(j,:)*q(i,:)';
            %orthogonalize A(j, :) to Q(i,:)
            A(j, :) = A(j,:) - l(j,i)*q(i,:);
        end;
    else
        % singular case!
        l(i,i) = 1;
        q(i,:) = zeros(1,N);
    end;
end;
```

We solve the factorized system in the same way we did for nonsingular case.

For example, let's pick $M = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$ and $b = \begin{bmatrix} 10 \\ 20 \end{bmatrix}$. The solution we get is $x = \begin{bmatrix} 10 \\ 0 \end{bmatrix}$ is not a least-squares solution. However, if we change right-hand side to $b = \begin{bmatrix} 10 \\ 10 \end{bmatrix}$, we will get the same solution - note that this is a vector with minimal norm, which solves this equation.