CS 322 Homework 2: Solutions

out:	Thursday 1 February 2007
due:	**Wednesday 7 February 2007**

Problem 1

We are given matrices $\mathbf{Q} = [\mathbf{q_1}, \mathbf{q_2}, \mathbf{q_3}]$ and $\mathbf{P} = [\mathbf{p_1}, \mathbf{p_2}, \mathbf{p_3}]$, and we wish to find the matrix $\mathbf{M}$ such that $\mathbf{MQ} = \mathbf{P}$. This looks similar to a system of linear equations; however, it is in the wrong form – we are used to solving $\mathbf{Ax} = \mathbf{b}$ where the unknown is in the middle, and in this problem it is on the left. We can fix that by applying the transpose to our problem, obtaining the modified system of equations $\mathbf{Q^T M^T} = \mathbf{P^T}$. Now our unknown is where we need it to be to solve using the algorithms we are accustomed to (in particular Gauss-Jordan elimination). We need to remember that when we solve the system we are obtaining the transpose of the matrix we actually want, and so once we've solved the system we need to apply the transpose to the solution in order to obtain the matrix $\mathbf{M}$ we are interested in.

We note that this is in fact a multi-RHS system: our right hand side is a matrix and not a vector. However, this does not make the problem any more difficult to solve; see the lecture notes on linear systems for more details.

Problem 2

```
function transform = computeLinearTransformation(origPts, newPts)

A = origPts';
B = newPts';
n = size(A,1);
C = [A B];
for j = 1:n
     % Get indices of rows to update
     r = [1:j-1 j+1:n];

     % Update all other rows to zero out the j-th column
     C(r,j:end) = C(r,j:end) - C(r,j) * C(j,j:end) / C(j,j);

     % Update the j-th row so that the j-th entry is one
     C(j,j:end) = C(j,j:end) / C(j,j);
end
transform = C(:,n+1:end)';
```

There are a few things to note about this code. First, we compute the transpose, form the augmented system, solve, and then apply the transpose again to get the matrix we need. Each iteration of our loop updates all of the rows except the current one at once using the outer product. One way of looking at the outer product is as multiple copies of a single row, each with a different scaling term applied to it (where the scaling terms come from the column vector in the outer product). In this case, we want to make multiple copies of the j-th row and scale each copy by the appropriate amount to eliminate the j-th variable in all other rows, so we perform the outer product of the j-th row with the column vector containing the appropriate scale factors (which are just $\mathbf{C}(i,j)/\mathbf{C}(j,j)$). The second thing to note is that we use some of the advanced indexing available in MATLAB. We form the vector $\mathbf{r}$, which is a row vector containing all integers from 1 to n except for j. We can then use it as an index into the matrix $\mathbf{C}$ – in essence we are using it to index every row but the j-th row. Finally, we know that the first $j-1$ entries in the row we are working on are already 0, and so any additions and scales involving those entries are unnecessary. Thus, we only have to transform the entries in the j through n-th columns of all rows; hence we operate on the columns j : end

Problem 3

One failure occurs when we take the bottom-left point on the left set of points and begin to slowly drag it right. As we cross the center of the image, our algorithm will fail with MATLAB complaining about a division by zero. This is symptomatic of a more general class of failures, ones in which the $\mathbf{C}(j,j)$ entry at some iteration of the Gauss-Jordan algorithm becomes very close to zero. There are other configurations of points that can cause the algorithm to fail on the second iteration, but the configuration described above causes it to consistently fail on the first iteration because the point in question is the first column of the matrix, and so the x-value is in $\mathbf{C}(1,1)$. By moving the point towards the center, we move x closer to zero, and at the center of the image when we divide by $\mathbf{C}(1,1)$ we are dividing by zero. This failure is fixed when we instead use the slash operator (or our code in problem 4) because pivoting will cause the problematic element to be replaced by a larger number.

Another failure occurs when we set all three points on the left in a straight line, or place two points on top of each other. This is because our points are now *linearly independent*; that is, we can express one point as some linear combination of the other two points. This means that the matrix $\mathbf{Q}$ is rank deficient, and so its inverse does not exist, and so the problem has no meaningful single solution. This failure is **not** fixed when we use the slash operator, because the failure is inherent to the problem – we are asking an ill-posed question without a meaningful answer, and so the slash operator will still fail to find a solution.

Problem 4

```
function transform = computeLinearTransformation(origPts, newPts)

A = origPts';
B = newPts';


n = size(A,1);
C = [A B];
permuteM = 1:n;
for j = 1:n
    % find pivot element
    [scale, ind] = max(abs(C(permuteM(j:n), j)));

    % set new pivot as the j-th row of our permutation
    permuteM([j j-1+ind]) = permuteM([j-1+ind j]);

    % generate list of all indices to update (everyone but the
    % current pivot)
    r = [permuteM(1:j-1) permuteM(j+1:n)];

    % Zero out every other entry in this column by adding a
    % scalar multiple of the pivot row
    C(r,j:end) = C(r,j:end) - C(r,j) * ...
                 C(permuteM(j),j:end) / C(permuteM(j),j);

    % Scale pivot row so diagonal entry is 1
    C(permuteM(j),j:end) = C(permuteM(j),j:end) / C(permuteM(j),j);
end

% Return transform matrix reordered by our permutation
transform = C(permuteM,n+1:end)';
```

One approach for pivoting is to physically swap the rows in memory. While that was an acceptable solution that got full credit, we have chosen to show another way of using MATLAB indexing to achieve greater performance. Rather than swapping the rows directly, we keep a list of what swaps we have made in the **permuteList** vector. The i-th entry in **permuteM** indicates what row in the original matrix is currently swapped into the i-th row of the permuted matrix. We use **max** and **abs** to find the index of the largest entry in the remaining columns and update **permuteM** to specify that we want to swap that row into the j-th row. Then, we use **permuteM** just like we used **r** before – as an index into what rows we want to update. Thus, the swaps are done automatically (and without copying) by MATLAB; by accessing them in a permuted order, MATLAB will return them in a permuted order, and it is as if we performed the permutations ourselves. For large matrices this can be more efficient since the swap never actually takes place.