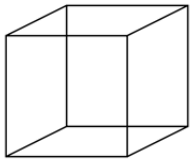
Homework 2 - SOLUTIONS

Due Monday, February 4, 2013

Notes: Please email me your solutions for these problems (in order) as a single Word or PDF document. If you do a problem on paper by hand, please scan it in and paste it into the document (although I would prefer it typed!).

1. **(20 pts) A cube has vertices in world coordinates: (0,0,0), (1,0,0), (1,1,0), (0,1,0), (0,0,1), (1,0,1), (1,1,1), (0,1,1). A camera is located at (X,Y,Z) = (3,-4,2) in world coordinates. The camera points directly at the origin and there is no roll about the axis (i.e., the +Z axis of the world points up in the image). Generate an image of a wireframe model of the cube as if were seen by the camera, as shown in the figure. Assume a pinhole camera model, with focal length = 600 pixels, where the image size is 640 pixels wide by 480 pixels high. Hints:**



   a. **You can draw a line in the image using Matlab's line function.**
   b. **The second vertex, the one with world coordinates (X,Y,Z) = (1,0,0), projects to pixel location (x,y) = (419, 268), rounded to the nearest pixel. Of course, the first vertex should be in the exact middle of the image.**
   c. **A good way to create the rotation matrix ${}^{W}_{C}R$ is to use the fact that its columns are the coordinate axes of the camera, expressed in world coordinates (see Lecture 4 slide 5). For example, the 3rd column is the z axis of the camera, expressed in world coordinates. This is just the direction in which the camera is pointing.**

Solution:
We use the fact that the +Z axis of the camera is the direction in which it points. We can get this by computing the vector from the camera's location, to the point at which it is looking (the world origin): $\hat{\mathbf{z}} = -\mathbf{t}/|\mathbf{t}|$, where $\mathbf{t}$ is the location of the camera in the world.

Next, we know that there is no roll about the axis (i.e., the +Z axis of the world points up in the image). This is equivalent to saying that the +X axis of the camera lies in the XY plane of the world. We can get this by doing a cross product of the camera's Z axis with world Z axis.

Finally, the Y axis of the camera is given by the cross product of the Z axis with the X axis.

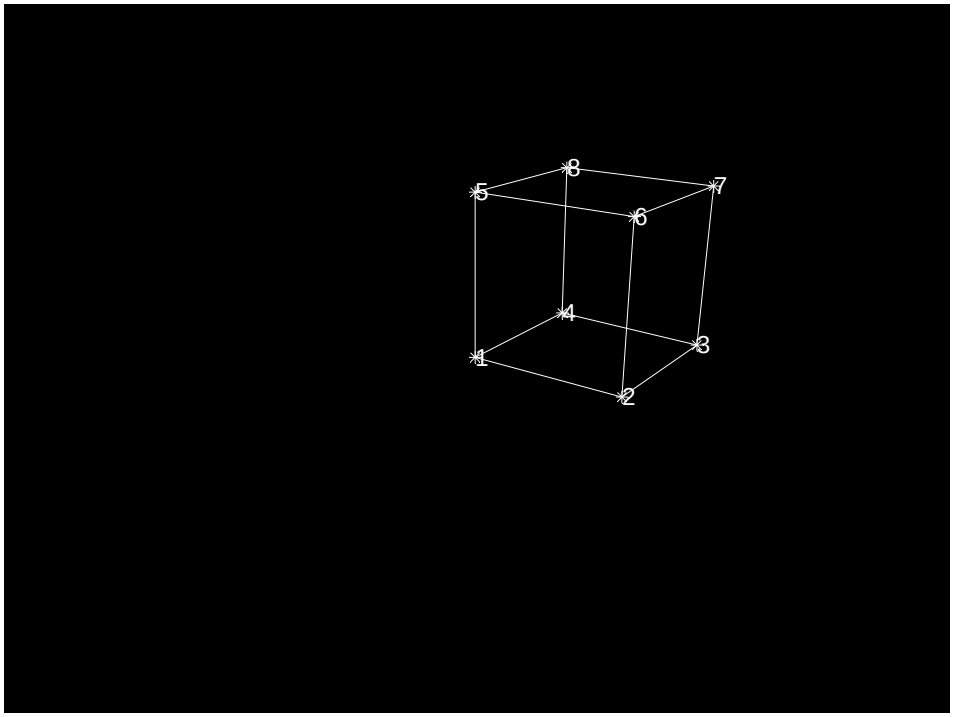
The code:

```
clear all
close all

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Create camera pose

tc_w = [3; -4; 2];          % location of camera origin in world

% The z axis of camera is the unit vector from camera to the world origin
uz_w = -tc_w/norm(tc_w);
```

```
% The x axis of camera is in xy plane of world.
ux_w = cross(uz_w, [0; 0; 1]);
ux_w = ux_w/norm(ux_w);

uy_w = cross(uz_w,ux_w);    % finally, form the y axis

R_c_w = [ ux_w uy_w uz_w ] ;
H_c_w = [R_c_w tc_w; 0 0 0 1];
H_w_c = inv(H_c_w);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Create camera projection matrices

% Here are the given parameters of the camera:
H = 480;        % height of image in pixels
W = 640;        % width of image in pixels
f = 600;        % focal length in pixels
cx = W/2;       % optical center
cy = H/2;

K = [ f  0  cx  ;   % Intrinsic camera parameter matrix
    0  f  cy  ;
    0  0   1  ];

% Extrinsic camera parameter matrix
Mext = H_w_c(1:3,:);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Define model in world coords

% The complete set of vertices is
P_w = [
    0   1   1   0   0   1   1   0;
    0   0   1   1   0   0   1   1;
    0   0   0   0   1   1   1   1;
    1   1   1   1   1   1   1   1];

% Define the lines to be drawn (indices of starting and ending points)
Lines = [
    1,  2;      % lower 4 points form a square
    2,  3;
    3,  4;
    4,  1;
    5,  6;      % upper 4 points form a square
    6,  7;
    7,  8;
    8,  5;
    1,  5;      % connect lower square to upper square
    2,  6;
    3,  7;
    4,  8 ];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Do perspective projection
p_img = K * Mext * P_w;
p_img(1,:) = p_img(1,:) ./ p_img(3,:);
p_img(2,:) = p_img(2,:) ./ p_img(3,:);
p_img(3,:) = p_img(3,:) ./ p_img(3,:);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Create image
I = zeros(H,W);
imshow(I, []);
hold on
for i=1:size(p_img,2)
    plot(p_img(1,i), p_img(2,i), 'w*');   % Mark points
    text(p_img(1,i), p_img(2,i), sprintf('%d', i), 'Color', 'w');
end

for i=1:size(Lines,1)
    i1 = Lines(i,1);    % index of starting point
    i2 = Lines(i,2);    % index of ending point
    line([p_img(1,i1) p_img(1,i2)], [p_img(2,i1) p_img(2,i2)], ...
        'Color', 'w');
end
```

The image:



Note that you can also construct the rotation matrix by performing rotations about the world axes. You first rotate about the world X axis, then about the world Z axis. The following code gives the same rotation matrix as the one computed above:

```
% Note - you can also compute the rotation matrix using a combination of
% rotations about the various axes:
ax = -pi/2-atan(2/5); ay = 0*pi/180; az = atan(3/4); % radians
Rx = [ 1 0 0; 0 cos(ax) -sin(ax); 0 sin(ax) cos(ax) ];
Ry = [ cos(ay) 0 sin(ay); 0 1 0; -sin(ay) 0 cos(ay) ];
Rz = [ cos(az) -sin(az) 0; sin(az) cos(az) 0; 0 0 1 ];
R_c_w = Rz * Ry * Rx;
disp('R_c_w:'), disp(R_c_w);
```

2. (15 pts) Repeat the previous problem, but this time generate an image of the cube assuming a weak perspective projection model (Lecture 5, slide 8).

Solution:

(b) To project the cube vertices onto the image using weak perspective projection, we use $\mathbf{p} = \mathbf{K}\,\mathbf{M}_{\text{ext}}\,{}^{W}\mathbf{P}$ where

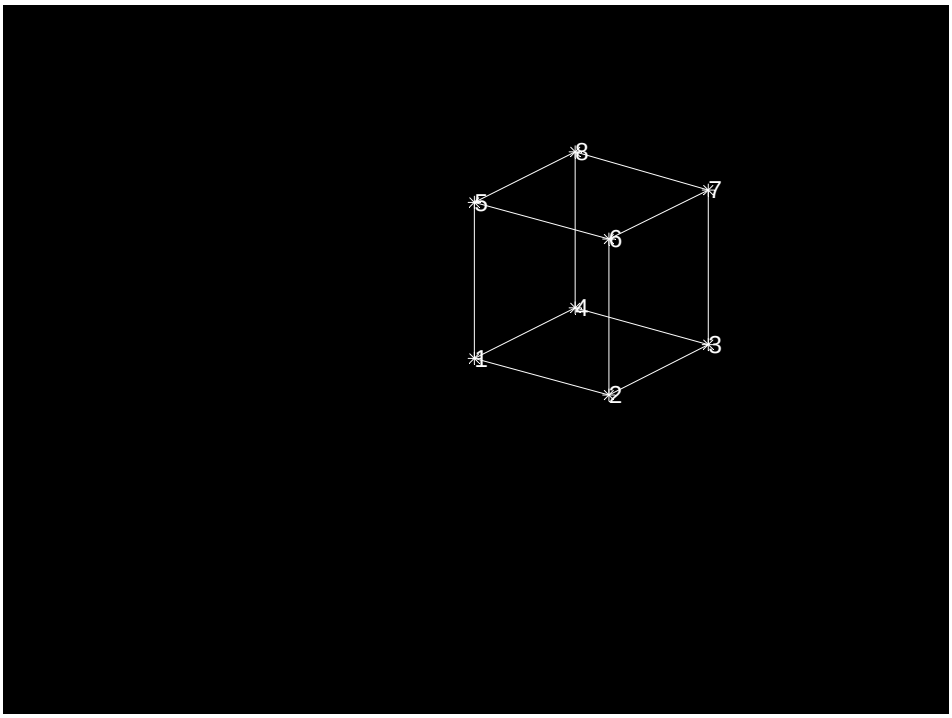
$$\mathbf{M}_{\text{ext}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & Z_{avg} \end{pmatrix} \begin{pmatrix} {}^{C}_{W}\mathbf{R} & {}^{C}\mathbf{t}_{Worg} \\ \mathbf{0} & 1 \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_X \\ r_{21} & r_{22} & r_{23} & t_Y \\ 0 & 0 & 0 & Z_{avg} \end{pmatrix}$$

Namely, we just make the last row of the matrix equal to (0,0,0, Zavg).

Here is the additional code and the image:

```
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Weak perspective projection

% We need the average Z distance from the camera to the points.
P_c = H_w_c * [0.5; 0.5; 0.5; 1];   % Get average point in camera coords
Zavg = P_c(3);
Mext(3,:) = [ 0 0 0 Zavg ];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Do weak projection
p_img = K * Mext * P_w;
p_img(1,:) = p_img(1,:) ./ p_img(3,:);
p_img(2,:) = p_img(2,:) ./ p_img(3,:);
p_img(3,:) = p_img(3,:) ./ p_img(3,:);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Create image
I = zeros(H,W);
figure, imshow(I, []);
hold on
for i=1:size(p_img,2)
    plot(p_img(1,i), p_img(2,i), 'w*');   % Mark points
    text(p_img(1,i), p_img(2,i), sprintf('%d', i), 'Color', 'w');
end

for i=1:size(Lines,1)
    i1 = Lines(i,1);    % index of starting point
    i2 = Lines(i,2);    % index of ending point
    line([p_img(1,i1) p_img(1,i2)], [p_img(2,i1) p_img(2,i2)], ...
        'Color', 'w');
end
```

The image:



3. (10 pts) Compute the following quantities by hand for the 5x5 image below. Assume as usual that +x is to the right, and +y is down.

- **a. What is the gradient at the pixel in at the center? Use the "central difference" operators; *i.e.*, df/dx ≈ [-0.5 0 +0.5], df/dy ≈ $[-0.5 \;\; 0 \;\; +0.5]^T$.**
- **b. What is the magnitude of the gradient at that point?**
- **c. What is the direction of the gradient at that point?**

6	7	8	9	10
5	6	7	8	9
4	5	6	7	8
3	4	5	6	7
2	3	4	5	6

Solution:

(a) $\nabla f = \left(\partial f/\partial x \quad \partial f/\partial y\right)^T = \left(1 \quad -1\right)^T$

(b) $|\nabla f| = \sqrt{\left(\partial f/\partial x\right)^2 + \left(\partial f/\partial y\right)^2} = \sqrt{2}$

(c) The direction of the gradient, expressed as a unit vector, is $\hat{\mathbf{n}} = \left(\sqrt{2} \quad -\sqrt{2}\right)^T$. Equivalently, you could specify it as an angle with respect to the x axis: $\theta = \tan^{-1}\left(\dfrac{\partial f/\partial x}{\partial f/\partial y}\right)$ = -45 degrees. It points up and to the right.

4. (15 pts) Implement a program to correlate a box filter consisting of a 15x15 array of 1's with the "pout.tif" image (Lecture 6, slide 10). For this problem, do not use the Matlab function `imfilter`, or any related functions such as `convn` in your implementation. Estimate by hand the total number of addition operations that are needed (note – no multiplications are needed).

Solution:

```
clear all
close all

I = double(imread('pout.tif'));

m = 7;      % Half size of filter; filter size is (2m+1)x(2m+1)

% Do a correlation of a 15x15 box filter
G = zeros(size(I));     % Allocate output image (not necessary, but faster)
for r=m+1:size(I,1)-m
    for c=m+1:size(I,2)-m

        % Do the inner summation over the filter coefficients
        sum = 0;
        for rr=r-m:r+m
            for cc=c-m:c+m
                sum = sum + I(rr, cc);
            end
        end
        G(r,c) = sum;

%         % Or, you can simply use the following instruction:
%         G(r,c) = sum(sum(I(r-m:r+m, c-m:c+m)));
    end
end

figure, imshow(G, []);     % This is the filtered image

% Compare to result using imfilter
G2 = imfilter(I, ones(15,15));
D = G2-G;
figure, imshow(D, []), impixelinfo
```

The filtered image:

The image is the same as that obtained directly via "imfilter" except for the border area.

To estimate the number of additions: This image is 291x240. We do the inner summation over the filter, at every pixel except those within 7 pixels of the border. So there are (291-14)*(240-14) = 62602 locations for which we do the inner summation.

At each of the above locations, we do 15*15 = 225 additions. So the total is 62602*225 = 14085450 for the whole image.

Note that there are other ways you could handle pixels near the border. When I did the above, I just set pixels near the border to zero, since the filter doesn't fit completely inside the image at those pixels. Another way to handle pixels near the border is to assume that the image contains zeros outside the boundary of the image. Then you can compute the filter result at pixels near the border. I believe "imfilter" uses this method by default. If you did the problem using this method, that is ok. In that case you would get a larger number of additions.

5. (10 pts) A fast method to implement box filters of various sizes is to create an "integral image". Create a "summed area table" or "integral image" for the 5x5 image shown below, as illustrated in Figure 3.17 in the textbook. Find the sum of the center 3x3 pixel region directly, and show that you get the same result using equation 3.32.

3	4	7	8	7
6	8	3	5	3
6	3	5	2	7
2	5	6	2	2
1	2	8	3	8

Solution:

```
% Compute summed area table
 S = zeros(size(I));
 for i=1:5
     for j=1:5
         S(i,j) = sum(sum(I(1:i,1:j)));
     end
 end
 disp(S);

 % Find the sum of the center 3x3 pixels
 s = S(4,4) - S(1,4) - S(4,1) + S(1,1);
 disp(s);

 % Compare to direct sum
 s = sum(sum(I(2:4,2:4)));
 disp(s);
```

Integral image is

3	7	14	22	29
9	21	31	44	54
15	30	45	60	77
17	37	58	75	94
18	40	69	89	116

The center sum is 39, computed both ways.

6. (10 pts) The following is a binary image.

	1	1			1			1	
1	1				1			1	
1		1	1	1	1	1	1		1
1						1	1		
1		1	1	1					
1		1		1		1		1	
1		1		1		1	1	1	
1		1	1	1		1	1	1	
1							1		
1	1	1	1	1	1	1	1		

(a) Label (by hand) the connected components assuming 4-neighbor connectivity. Compare your result to that obtained by Matlab's `bwlabel`.

(b) Label (by hand) the connected components assuming 8-neighbor connectivity. Compare your result to that obtained by Matlab's `bwlabel`.

Labeling the image by hand using 4-neighbor connectivity, you get 5 components:

	1	1			2			3	
1	1				2			3	
1		2	2	2	2	2	2		4
1						2	2		
1		5	5	5					
1		5		5		1		1	
1		5		5		1	1	1	
1		5	5	5		1	1	1	
1							1		
1	1	1	1	1	1	1	1		

Labeling the image by hand using 8-neighbor connectivity, you get 2 components:

	1	1			1			1	
1	1				1			1	
1		1	1	1	1	1	1		1
1						1	1		
1		2	2	2					
1		2		2		1		1	
1		2		2		1	1	1	
1		2	2	2		1	1	1	
1							1		
1	1	1	1	1	1	1	1		

We use the following Matlab code to do the labeling:

```
% HW2 problem 5
clear all
close all

% Here is the binary image
B = [
0   1   1   0   0   1   0   0   1   0;
1   1   0   0   0   1   0   0   1   0;
1   0   1   1   1   1   1   1   0   1;
1   0   0   0   0   0   1   1   0   0;
1   0   1   1   1   0   0   0   0   0;
1   0   1   0   1   0   1   0   1   0;
```

```
1   0   1   0   1   0   1   1   1   0;
1   0   1   1   1   0   1   1   1   0;
1   0   0   0   0   0   0   1   0   0;
1   1   1   1   1   1   1   1   0   0];

[L4,n4] = bwlabel(B,4);     % 4-connected
fprintf('Number of 4-connected components = %d\n', n4);
disp(L4);

[L8,n8] = bwlabel(B,8);     % 8-connected
fprintf('Number of 8-connected components = %d\n', n8);
disp(L8);
```

Here is the output. The results are the same as with the hand-labeling, except for the actual labels, which are arbitrary.

Number of 4-connected components = 5

0	1	1	0	0	2	0	0	4	0
1	1	0	0	0	2	0	0	4	0
1	0	2	2	2	2	2	2	0	5
1	0	0	0	0	0	2	2	0	0
1	0	3	3	3	0	0	0	0	0
1	0	3	0	3	0	1	0	1	0
1	0	3	0	3	0	1	1	1	0
1	0	3	3	3	0	1	1	1	0
1	0	0	0	0	0	0	1	0	0
1	1	1	1	1	1	1	1	0	0

Number of 8-connected components = 2

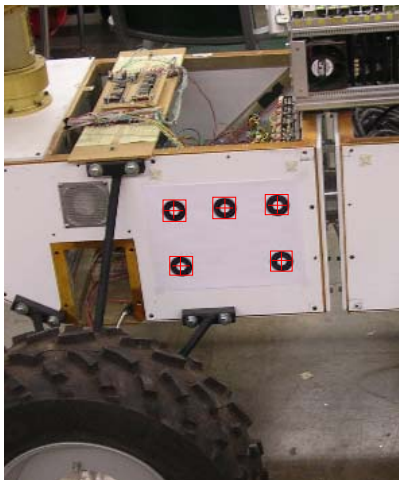
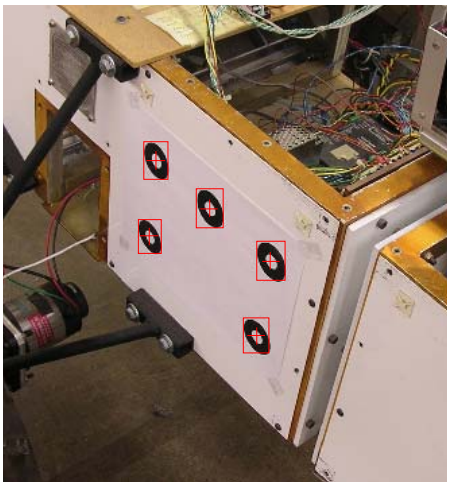
0	1	1	0	0	1	0	0	1	0
1	1	0	0	0	1	0	0	1	0
1	0	1	1	1	1	1	1	0	1
1	0	0	0	0	0	1	1	0	0
1	0	2	2	2	0	0	0	0	0
1	0	2	0	2	0	1	0	1	0
1	0	2	0	2	0	1	1	1	0
1	0	2	2	2	0	1	1	1	0
1	0	0	0	0	0	0	1	0	0
1	1	1	1	1	1	1	1	0	0

7. (20 pts) The images "robot1.jpg", "robot2.jpg", and "robot3.jpg" are on the course website. Using the same program (i.e., you can't change any thresholds or parameters other than the name of the image file), find the concentric circle targets in all three

images and draw crosshairs on top of the targets, on the original images. Your program should find all the valid fiducial targets, and reject false targets (you may have to use additional tests to reject false targets). Give your program as well as the images with the overlays. Extra credit: Automatically identify the targets and label them with "UL", "UM", "UR", "LL", or "LR" for upper left, upper right, lower left, etc.

Solution: Using the method we developed in class, the approach is to threshold the image automatically and look for white blobs, then complement it and look for black blobs. We do a morphological "opening" to remove small (noise) regions. Then we look for white blobs and black blobs whose centroids are located at almost the same point (they won't necessarily be exactly the same point due to image noise). I used a threshold (empirically chosen) of 3 pixels for this test.

This finds the targets, but also finds a couple of other false targets. You need some additional tests. One easy one is that the black area should be greater than the white area (or you could test whether the black bounding box encloses the white bounding box). With this additional test, my code finds all targets reliably, and no false targets.

```
clear all
close all

S=strel('disk', 5);     % structuring element

I=imread('robot2.jpg');     % Edit name to read in different images
W=im2bw(I,graythresh(I));
W=imopen(W,S);
[LW,nw]=bwlabel(W);
statsWhite = regionprops(LW);
figure, imshow(LW), impixelinfo;

B=~W;                       % complement image, to find black blobs
[LB,nb]=bwlabel(B);
```

```
statsBlack = regionprops(LB);
figure, imshow(LB), impixelinfo;

D = 3;                          % threshold for how close centroids must be
figure, imshow(I,[]);
n = 0;  % number of targets found
for i=1:nw
    for j=1:nb
        if norm( statsWhite(i).Centroid - statsBlack(j).Centroid ) < D
            if statsWhite(i).Area < statsBlack(j).Area
                x0 = (statsWhite(i).Centroid(1) + ...
                    statsBlack(j).Centroid(1))/2;
                y0 = (statsWhite(i).Centroid(2) + ...
                    statsBlack(j).Centroid(2))/2;

                % Draw crosshair on image
                line( [x0-20 x0+20], [y0 y0], 'Color', 'r');
                line( [x0 x0], [y0-20 y0+20], 'Color', 'r');

                % Draw bounding box around outer (black) blob
                rectangle('Position', statsBlack(j).BoundingBox, ...
                    'EdgeColor', 'r');

                % Save target location
                n = n + 1;
                target(:, n) = [x0; y0];
            end
        end
    end
end
```

Identifying the targets is a bit harder. This is called the "correspondence" problem – trying to find the correspondence between image features and model features – and is a key part of object recognition. We will get into this in more detail later in the course. One approach is to look for features that are invariant with respect to pose. A simple invariant feature for this target is the co-linearity of the three top target circles. Regardless of pose, a set of three points that are collinear in 3D space will also be collinear in the image. So you could try all combinations of three detected features to find a set that is collinear. Identifying the order of the three features, and the two remaining features, is then pretty straightforward.

Here is one possible solution:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Optional part:  Do correspondence

% First find the UM point.  This is the point that is closest to the
% midpoint between two other points.
idMidpoint = zeros(5,5);
dMidpoint = Inf(5,5);
for i=1:5
```

```
    for j=i+1:5
        pMid = (target(:, i) + target(:, j))/2;     % ideal midpoint

        d = Inf(5,1);   % Distances of a 3rd point to the ideal midpoint
        for k=1:5
            if k==i || k==j  continue;  end
            d(k) = norm(pMid-target(:, k));
        end

        % Get the point that is closest to the ideal midpoint btwn i,j
        [dmin,k] = min(d);
        dMidpoint(i,j) = dmin;
        idMidpoint(i,j) = k;
    end
end
[i1,i3] = find(dMidpoint == min(dMidpoint(:)));
i2 = idMidpoint(i1,i3);

% Find the two other points, other than the triple we have already found
allids = 1:5;
ids = find( ~(allids==i1 | allids==i2 | allids==i3) );
i4 = ids(1);
i5 = ids(2);

% Signed area is the determinant of the 2x2 matrix [ p4-p1, p3-p1 ]
M = [ target(:,i4)-target(:,i1) target(:,i3)-target(:,i1) ];
if det(M) < 0
    idTargets(1) = i1;    % UL
    idTargets(2) = i2;    % UM
    idTargets(3) = i3;    % UR;
else
    idTargets(1) = i3;    % UL
    idTargets(2) = i2;    % UM
    idTargets(3) = i1;    % UR;
end

% LL is the closer point to UL
if norm( target(:,i4)-target(:,idTargets(1)) ) < norm( target(:,i5)-
target(:,idTargets(1)) )
    idTargets(4) = i4;      % LL
    idTargets(5) = i5;      % LR
else
    idTargets(4) = i5;      % LL
    idTargets(5) = i4;      % LR
end

% Label the targets in the image
p = target(:, idTargets(1));    text(p(1)+15, p(2)+15, 'UL');
p = target(:, idTargets(2));    text(p(1)+15, p(2)+15, 'UM');
p = target(:, idTargets(3));    text(p(1)+15, p(2)+15, 'UR');
p = target(:, idTargets(4));    text(p(1)+15, p(2)+15, 'LL');
p = target(:, idTargets(5));    text(p(1)+15, p(2)+15, 'LR');

fprintf('Target locations (UL,UM,UR,LL,LR):\n');
for i=1:5
```

```
    p = target(:, idTargets(i));
    fprintf('(x,y) = (%f,%f)\n', p(1), p(2));
end
```