Homework 4 - SOLUTIONS

Due Monday, March 4, 2013 (by 8:00 pm)

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. **(25 pts) A set of 2D points $\mathbf{P}_A = \{\mathbf{p}_A^{(1)},\ldots,\mathbf{p}_A^{(N)}\} = \{(x_A^{(1)}, y_A^{(1)}), \ldots, (x_A^{(N)}, y_A^{(N)})\}$ is extracted from image A. Corresponding points $\mathbf{P}_B = \{\mathbf{p}_B^{(1)},\ldots,\mathbf{p}_B^{(N)}\}$ are extracted from image B. You want to find a scaled similarity transform[1] that aligns the two, using a least squares fit. The transformation can be written as a nonlinear function $\mathbf{P}_B = \mathbf{f}(\mathbf{P}_A, \mathbf{x})$, where $\mathbf{x} = (s, \theta, t_x, t_y)^T$ is the vector of unknown transformation parameters.**
   a. **Write the symbolic form of the Jacobian matrix of f.**
   b. **Write code to solve for the transformation, using the point correspondences given below (for an initial guess, use *s*=1, and zeros for the other parameters). What is the final residual error (in terms of the sum of the squared errors)?**

Image points from image A (x;y):

126	**34**	**170**	**103**
60	**107**	**77**	**163**

Corresponding points from image B (x;y):

173	**95**	**196**	**121**
49	**47**	**77**	**107**

Solution: The scaled similarity transform is

$$\begin{pmatrix} x_B^{(i)} \\ y_B^{(i)} \\ 1 \end{pmatrix} = \begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_A^{(i)} \\ y_A^{(i)} \\ 1 \end{pmatrix}$$

or

$$x_B^{(i)} = s(\cos\theta)x_A^{(i)} - s(\sin\theta)y_A^{(i)} + st_x$$
$$y_B^{(i)} = s(\sin\theta)x_A^{(i)} + s(\cos\theta)y_A^{(i)} + st_y$$

(a) The Jacobian is

[1] See lecture notes on 2D-2D image transforms, slide 6.

$$\mathbf{J} = \left[\frac{\partial f_i}{\partial x_j}\right] = \begin{pmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \partial f_1/\partial x_3 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \partial f_2/\partial x_3 \\ \vdots & \vdots & \vdots \\ \partial f_{2N}/\partial x_1 & \partial f_{2N}/\partial x_2 & \partial f_{2N}/\partial x_3 \end{pmatrix}_{2Nx3}$$

$$= \begin{pmatrix} \partial f_1/\partial s & \partial f_1/\partial \theta & \partial f_1/\partial t_x & \partial f_1/\partial t_y \\ \partial f_2/\partial s & \partial f_2/\partial \theta & \partial f_2/\partial t_x & \partial f_2/\partial t_y \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

or

$$\mathbf{J} = \begin{pmatrix} x_A^{(1)}\cos\theta - y_A^{(1)}\sin\theta + t_x & -s\,x_A^{(1)}\sin\theta - s y_A^{(1)}\cos\theta & s & 0 \\ x_A^{(1)}\sin\theta + y_A^{(1)}\cos\theta + t_y & s x_A^{(1)}\cos\theta - s\,y_A^{(1)}\sin\theta & 0 & s \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

(b) Here is the Matlab code:

```
% HW4 problem 1
clear all, close all


% Here are the corresponding points
pA = [
   126    34   170   103;
    60   107    77   163;
     1     1     1     1];
pB = [
   173    95   196   121;
    49    47    77   107];
N = size(pA,2);

s = 1.0;
theta = 0;
tx = 0.0;
ty = 0;
x = [s; theta; tx; ty];    % initial guess

while true
    disp('Parameters (s; theta; tx; ty):'), disp(x);

    y = fTransform(x, pA); % Call function to compute expected measurements

    dy = reshape(pB,[],1) - y;      % new residual
    fprintf('Residual (sum of squared error): %f\n', sum(dy.^2));

    J = zeros(2*N,4);

    % Fill in values of J
    s = x(1);  theta = x(2);
    tx = x(3); ty = x(4);
    for i=1:N
        xA = pA(1,i);   yA = pA(2,i);
        J( 2*(i-1)+1, :) = [
            xA*cos(theta)-yA*sin(theta)+tx, -s*xA*sin(theta)-s*yA*cos(theta), s, 0 ];
        J( 2*(i-1)+2, :) = [
            xA*sin(theta)+yA*cos(theta)+ty,  s*xA*cos(theta)-s*yA*sin(theta), 0, s ];

    end

    dx = pinv(J)*dy;

    % Stop if parameters are no longer changing
    if abs( norm(dx)/norm(x) ) < 1e-6
        break;
    end

    x = x + dx;    % add correction
end
```

And the function to transform a set of points:

```
function y = fTransform(x, pIn)
% Do 2D scaled rigid transform

% Get params
s = x(1);
theta = x(2);
tx = x(3);
ty = x(4);

H = [s 0 0; 0 s 0; 0 0 1]*...
    [ cos(theta)  -sin(theta) tx;
      sin(theta)   cos(theta) ty;
      0  0  1];

pOut = H*pIn;   % Transform points, result is a 3xN matrix

% Reshape to be a 2Nx1 vector in the form (x1;y1;x2;y2; ... yN)
y = reshape(pOut(1:2, :), [], 1);

return
```

The final result:

```
Parameters (s; theta; tx; ty):
    0.7479
    0.5068
  150.2917
  -47.9060
Residual (sum of squared error): 3.145506
```

2. **(25 pts) Your task in this problem is to do some experiments with the SIFT code developed in class. This code extracted SIFT features from two images, matched the features, and found a set of correspondences that had the same scale, rotation, and location. The code can be used to recognize an object in a cluttered scene. Download**

the image dataset from the website http://www.computing.dundee.ac.uk/staff/jessehoey/teaching/vision/project1.html.

a. **Take the object images "book1.pgm" and "book2.pgm". Look at each of the images named "Img0[i].pgm", where [i]=1...10., and determine whether each object is present in the scene (note: each object is present in 5 of the images). Now use the SIFT code to match each object to each image, and determine the number of points that were matched.**

b. **Using your results from above, decide upon a threshold on the number of feature matches that can classify whether the object is present in an image. Ideally, if you get that number of matches or more, the object is actually present in the image; otherwise, if you get fewer matches, the object is actually not present. You may not be able to find a threshold that gives perfect classification results, but choose the value that the minimizes the number of misses (number of images that contained the object that were classified as not containing the object) and the number of false positives (number of images that do not contain the object that were classified as containing the object).**

c. **Now match the two objects to the test images named "TestImg0[i].pgm", where [i]=1...10, and use the same threshold to indicate the presence of the object. Compute the number of misses and the number of false positives.**

Solution:

I ran the code that follows at the end of this problem. Here are the parameters I used (I didn't try optimizing any of them):

Parameter	*Value*	*Description*
Peak_thresh, Edge_thresh	Peak_thresh = 0 (0 = default) Edge_thresh = 10 (10 = default)	Limits the number of features detected by `vl_sift`
matching threshold for `vl_ubcmatch`	Threshold = 2.0 (default is 1.5)	Descriptor D1 is matched to a descriptor D2 only if the distance d(D1,D2) multiplied by thresh is not greater than the distance of D1 to all other descriptors
Resolution of Hough parameter array	x,y is divided into 5 bins theta is divided into 9 bins scale is divided	This is the resolution of the Hough array that accumulates votes for a particular (x,y,theta,scale)

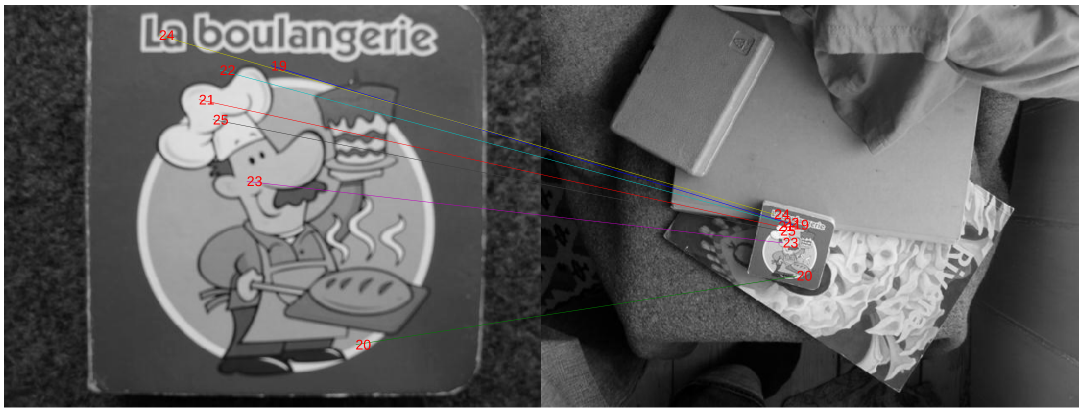
	into 5 bins	

The results of matching two objects (book1 and book2) to each of the training images:

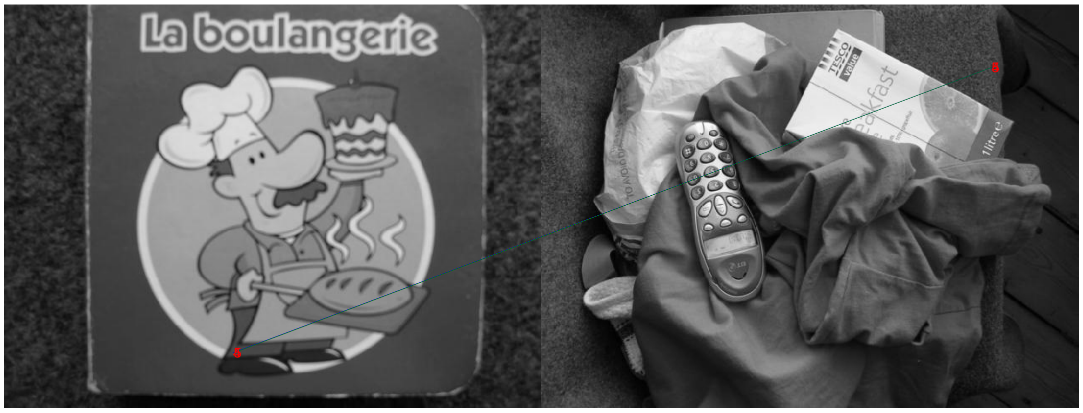
Training image	book1			book2		
	Actually in image (y/n)?	# points matched	Classification result (y/n)	Actually in image (y/n)?	# points matched	Classification result (y/n)
Img01.pgm	y	7	y	y	75	y
Img02.pgm	y	15	y	y	40	y
Img03.pgm	y	3	n	y	137	y
Img04.pgm	y	8	y	n	1	n
Img05.pgm	y	13	y	n	2	n
Img06.pgm	n	2	n	n	1	n
Img07.pgm	n	2	n	y	8	y
Img08.pgm	n	4	y	y	9	y
Img09.pgm	n	2	n	n	3	n
Img010.pgm	n	1	n	n	2	n

So from these results, it looks like that if we simply set a threshold of 4 or more points matched, then we would have only one false positive and one miss out of these 20 runs. The false positive (shown in green above) would be incorrectly detecting book1 in Img08, because it really is not in the scene. The miss (shown in red above) would be incorrectly saying that book1 is not in Img03, because it really is in the scene.

Two example runs are shown below. This is the result of matching book1.pgm to Img01.pgm (the object is in the scene, and 7 points are matched):



Here is the result of matching book1.pgm to Img06.pgm (the object is not in the scene, and 2 points are matched):



Here are the results of matching book1 and book2 to the test data:

Test image	book1			book2		
	Actually in image (y/n)?	# points matched	Classification result present (y/n)?	Actually in image (y/n)?	# points matched	Classification result present (y/n)?
TestImg01.pgm	y	6	y	y	7	y
TestImg02.pgm	y	16	y	n	2	n
TestImg03.pgm	n	2	n	n	3	n
TestImg04.pgm	y	4	y	n	1	n
TestImg05.pgm	n	1	n	y	136	y
TestImg06.pgm	n	1	n	y	19	y
TestImg07.pgm	n	1	n	y	7	y
TestImg08.pgm	n	3	n	n	2	n
TestImg09.pgm	y	2	n	n	1	n
TestImg010.pgm	y	3	n	y	127	y

If we use the threshold of 4 or points needed to signal a detection, then we have two false negatives (shown in red in the table above) and no false positives.

Here is the complete code (other than the functions from the vl_feat library):

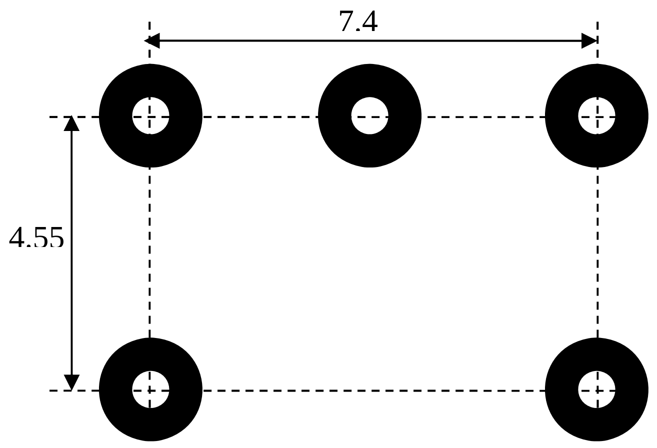
```
clear all
close all

% Need to do this to set up the pathname:
%run('C:\Users\whoff\Documents\Research\SIFT\vlfeat-0.9.16\toolbox\vl_setup');

I1 = imread('images/book2.pgm');
I1 = single(I1);  % Convert to single precision floating point

% These parameters limit the number of features detected
peak_thresh = 0;     % increase to limit; default is 0
edge_thresh = 10;    % decrease to limit; default is 10
```

```
[f1,d1] = vl_sift(I1, ...
    'PeakThresh', peak_thresh, ...
    'edgethresh', edge_thresh );
fprintf('Number of frames (features) detected: %d\n', size(f1,2));


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Second image
I2 = single( imread('images/TestImg010.pgm') );

% These parameters limit the number of features detected
peak_thresh = 0;    % increase to limit; default is 0
edge_thresh = 10;   % decrease to limit; default is 10

[f2,d2] = vl_sift(I2, ...
    'PeakThresh', peak_thresh, ...
    'edgethresh', edge_thresh );
fprintf('Number of frames (features) detected: %d\n', size(f2,2));


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Threshold for matching
% Descriptor D1 is matched to a descriptor D2 only if the distance d(D1,D2)
% multiplied by THRESH is not greater than the distance of D1 to all other
% descriptors
thresh = 2.0;   % default = 1.5; increase to limit matches
[matches, scores] = vl_ubcmatch(d1, d2, thresh);
fprintf('Number of matching frames (features): %d\n', size(matches,2));

indices1 = matches(1,:);        % Get matching features
f1match = f1(:,indices1);
d1match = d1(:,indices1);

indices2 = matches(2,:);
f2match = f2(:,indices2);
d2match = d2(:,indices2);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Between all pairs of matching features, compute
% orientation difference, scale ratio, and center offset
allScales = zeros(1,size(matches,2));   % Store computed values
allAngs = zeros(1,size(matches,2));
allX = zeros(1,size(matches,2));
allY = zeros(1,size(matches,2));

for i=1:size(matches, 2)
    scaleRatio = f1match(3,i)/f2match(3,i);
    dTheta = f1match(4,i) - f2match(4,i);

    % Force dTheta to be between -pi and +pi
    while dTheta > pi   dTheta = dTheta - 2*pi;     end
    while dTheta < -pi   dTheta = dTheta + 2*pi;     end

    allScales(i) = scaleRatio;
    allAngs(i) = dTheta;

    x1 = f1match(1,i);      % the feature in image 1
    y1 = f1match(2,i);
    x2 = f2match(1,i);      % the feature in image 2
    y2 = f2match(2,i);
    % The "center" of the object in image 1 is located at an offset of
    % (-x1,-y1) relative to the detected feature.  We need to scale and rotate
    % this offset and apply it to the image 2 location.
    offset = [-x1; -y1];
    offset = offset / scaleRatio;   % Scale to match image 2 scale
    offset = [cos(dTheta) +sin(dTheta); -sin(dTheta) cos(dTheta)]*offset;

    allX(i) = x2 + offset(1);
    allY(i) = y2 + offset(2);
end

% Use a coarse Hough space.
% Dimensions are [angle, scale, x, y]
% Define bin centers
aBin = -pi:(pi/4):pi;
sBin = 0.5:(2):10;
xBin = 1:(size(I2,2)/5):size(I2,2);
yBin = 1:(size(I2,1)/5):size(I2,1);

H = zeros(length(aBin), length(sBin), length(xBin), length(yBin));
for i=1:size(matches, 2)
    a = allAngs(i);
    s = allScales(i);
    x = allX(i);
    y = allY(i);

    % Find bin that is closest to a,s,x,y
    [~, ia] = min(abs(a-aBin));
    [~, is] = min(abs(s-sBin));
    [~, ix] = min(abs(x-xBin));
    [~, iy] = min(abs(y-yBin));

    H(ia,is,ix,iy) = H(ia,is,ix,iy) + 1;   % Inc accumulator array
end

% Find all bins with 3 or more features
[ap,sp,xp,yp] = ind2sub(size(H), find(H>=3));


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get the features corresponding to the largest bin
nFeatures = max(H(:));      % Number of features in largest bin
fprintf('Largest bin contains %d features\n', nFeatures);
[ap,sp,xp,yp] = ind2sub(size(H), find(H == nFeatures));
indices = [];   % Make a list of indices
for i=1:size(matches, 2)
    a = allAngs(i);
    s = allScales(i);
    x = allX(i);
    y = allY(i);

    % Find bin that is closest to a,s,x,y
    [~, ia] = min(abs(a-aBin));
    [~, is] = min(abs(s-sBin));
    [~, ix] = min(abs(x-xBin));
    [~, iy] = min(abs(y-yBin));

    if ia==ap(1) && is==sp(1) && ix==xp(1) && iy==yp(1)
        indices = [indices i];
    end
end
```

```
% Show matches to features in largest bin as line segments
figure, imshow([I1,I2],[]);
o = size(I1,2) ;
line([f1match(1,indices);f2match(1,indices)+o], ...
    [f1match(2,indices);f2match(2,indices)]) ;
for i=1:length(indices)
    x = f1match(1,indices(i));
    y = f1match(2,indices(i));
    text(x,y,sprintf('%d',indices(i)), 'Color', 'r');
end
for i=1:length(indices)
    x = f2match(1,indices(i));
    y = f2match(2,indices(i));
    text(x+o,y,sprintf('%d',indices(i)), 'Color', 'r');
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Fit an affine transformation to those features.
% We use affine transformation because the image of a planar surface
% undergoing a small out-of-plane rotation can be approximated by an
% affine transformation.

% Create lists of corresponding points pA and pB.
pA = [f1match(1,indices); f1match(2,indices)];
pB = [f2match(1,indices); f2match(2,indices)];
N = size(pA,2);

% Calculate the transformation T from I1 to I2; ie p2 = T p1.
A = zeros(2*N,6);
for i=1:N
    A( 2*(i-1)+1, :) = [ pA(1,i)  pA(2,i)  0       0       1  0];
    A( 2*(i-1)+2, :) = [ 0        0        pA(1,i) pA(2,i) 0  1];
end
b = reshape(pB, [], 1);
x = A\b;

T = [ x(1)  x(2)    x(5);
      x(3)  x(4)    x(6);
      0     0       1];
fprintf('Derived affine transformation:\n');
disp(T);

r = A*x-b;          % Residual error
ssr = sum(r.^2);    % Sum of squared residuals

% Estimate the error for each image point measurement.
% For N image points, we get two measurements from each, so there are 2N
% quantities in the sum.  However, we have 6 degrees of freedom in the result.
sigmaImg = sqrt(ssr/(2*N-6));       % Estimated image std deviation
fprintf('#pts = %d, estimated image error = %f pixels\n', N, sigmaImg);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Ok, apply the transformation to image 1 to align it to image 2.
% We'll use Matlab's imtransform function.
tform = maketform('affine', T');
I3 = imtransform(I1,tform, ...
    'XData', [1 size(I1,2)], 'YData', [1 size(I1,1)] );


% Overlay the images
```

```
RGB(:,:,1) = (I2+I3)/2;
RGB(:,:,2) = (I2+I3)/2;
RGB(:,:,3) = I2/2;

RGB = uint8(RGB);
figure, imshow(RGB);
```

3. **(25 pts) The objective of this exercise is to compute the pose of a known object with respect to a camera, from a set of correspondences between object points and observed image points. In Lecture 16 on "pose estimation", I gave an example of how to compute the pose using nonlinear least squares. Using that method, find the pose of the 5-point concentric circle target with respect to the camera. Use the images "robot1_rect.jpg", "robot2_rect.jpg", and "robot3_rect.jpg" on the course website[2]. The measurements of the model (in inches) are indicated in the figure. The top middle target feature is midway between the top left and the top right feature. Use the top left feature as the origin of the target's coordinate system, with its x-axis pointing to the right, its y-axis pointing down, and its z-axis pointing into the page.**



(a) For each image, give the pose of the target with respect to the camera (in terms of XYZ angles and XYZ translation) in each image.
(b) Use that pose to draw the XYZ coordinate axes of the target as an overlay on each image.

Note:

- **Your program should automatically detect and identify the CCC targets in the images, using the methods developed in HW2.**
- **If you need to, it is ok to use a different initial guess for the pose in each image, in order to get your algorithm to converge to the correct solution (you will know**

[2] Note – these images have been "rectified" to remove lens distortion. We will cover how to do this a little later in the course. We will also cover how to compute the camera intrinsic parameters

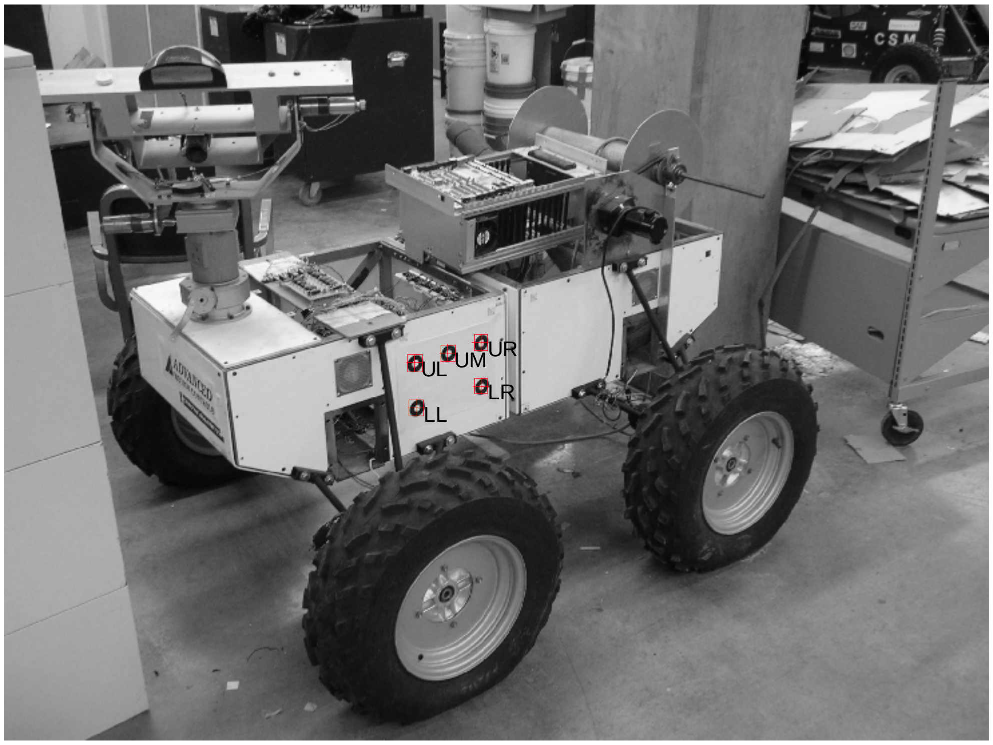
that the solution is correct because the graphical overlays will look right). The model is about 95 inches from the camera in the first image.

- **Use the following values for the camera intrinsic parameters: focal length = 2443 (in pixels), image center (*x,y* in pixels) = (1124, 831).**

Solution: The first part of the code is directly from Homework 2:

```
% Homework 4 problem 4.
% Find the pose of a 5-ccc target.

clear all
close all

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

I=imread('robot1_rect.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

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

The result of running this on the first image:



Next, we use the iterative least squares pose estimation code developed in class:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Find the pose from the point correspondences
imshow(I, [])

% First specify the points in the model's coordinate system (inches), in
% the same order as the detected points; ie UL,UM,UR,LL,LR.  Each column is
% a point in the form X;Y;Z;1.
P_M = [  0     3.7     7.4      0      7.4;
         0      0       0      4.55    4.55;
         0      0       0       0       0;
         1      1       1       1       1 ];

% Define camera parameters
f = 2443;      % focal length in pixels
cx = 1124;
cy = 831;

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

% Create the vector of measurements, as a single column vector in the form
% x1;y1;x2;y2; ... yN
```

```
y0 = [];
for i=1:5
    p = target(:, idTargets(i));
    y0 = [y0; p(1); p(2)];
end

% Make an initial guess of the pose [ax ay az tx ty tz].
% We know the target is about 80 inches away (tz).
% Leave the other parameters at zero.
% (For image robot3_rect.jpg, I had to use ay=0.5, az=0.5).
x = [0; 0.0; 0.0; 0; 0; 80];

% Get predicted image points by substituting in the current pose
y = fProject(x, P_M, K);

for i=1:2:length(y)
    rectangle('Position', [y(i)-8 y(i+1)-8 16 16], 'FaceColor', 'r');
end

pause

for i=1:10
    fprintf('\nIteration %d\nCurrent pose:\n', i);
    disp(x);

    % Get predicted image points
    y = fProject(x, P_M, K);

    imshow(I, [])
    for i=1:2:length(y)
        rectangle('Position', [y(i)-8 y(i+1)-8 16 16], ...
            'FaceColor', 'r');
    end
    pause;

    % Estimate Jacobian
    e = 0.00001;    % a tiny number
    J(:,1) = ( fProject(x+[e;0;0;0;0;0],P_M,K) - y )/e;
    J(:,2) = ( fProject(x+[0;e;0;0;0;0],P_M,K) - y )/e;
    J(:,3) = ( fProject(x+[0;0;e;0;0;0],P_M,K) - y )/e;
    J(:,4) = ( fProject(x+[0;0;0;e;0;0],P_M,K) - y )/e;
    J(:,5) = ( fProject(x+[0;0;0;0;e;0],P_M,K) - y )/e;
    J(:,6) = ( fProject(x+[0;0;0;0;0;e],P_M,K) - y )/e;


    % Error is observed image points - predicted image points
    dy = y0 - y;
    fprintf('Residual error: %f\n', norm(dy));

    % Ok, now we have a system of linear equations   dy = J dx
    % Solve for dx using the pseudo inverse
    dx = pinv(J) * dy;

    % Stop if parameters are no longer changing
    if abs( norm(dx)/norm(x) ) < 1e-6
        break;
    end

    x = x + dx;   % Update pose estimate
end

u0 = fProject(x, [0;0;0;1], K);  % origin
uX = fProject(x, [1;0;0;1], K);  % unit X vector
```

```
uY = fProject(x, [0;1;0;1], K);  % unit Y vector
uZ = fProject(x, [0;0;1;1], K);  % unit Z vector

line([u0(1) uX(1)], [u0(2) uX(2)], 'Color', 'r', 'LineWidth', 3);
line([u0(1) uY(1)], [u0(2) uY(2)], 'Color', 'g', 'LineWidth', 3);
line([u0(1) uZ(1)], [u0(2) uZ(2)], 'Color', 'b', 'LineWidth', 3);
```

We also need the function “fProject.m”:

```
function p = fProject(x, P_M, K)
% Project 3D points onto image

% Get pose params
ax = x(1); ay = x(2); az = x(3);
tx = x(4); ty = x(5); tz = x(6);

% Rotation matrix, model to camera
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 = Rz * Ry * Rx;

% Extrinsic camera matrix
Mext = [ R  [tx;ty;tz] ];

% Project points
ph = K*Mext*P_M;

% Divide through 3rd element of each column
ph(1,:) = ph(1,:)./ph(3,:);
ph(2,:) = ph(2,:)./ph(3,:);
ph = ph(1:2,:); % Get rid of 3rd row

p = reshape(ph, [], 1); % reshape into 2Nx1 vector
return
```

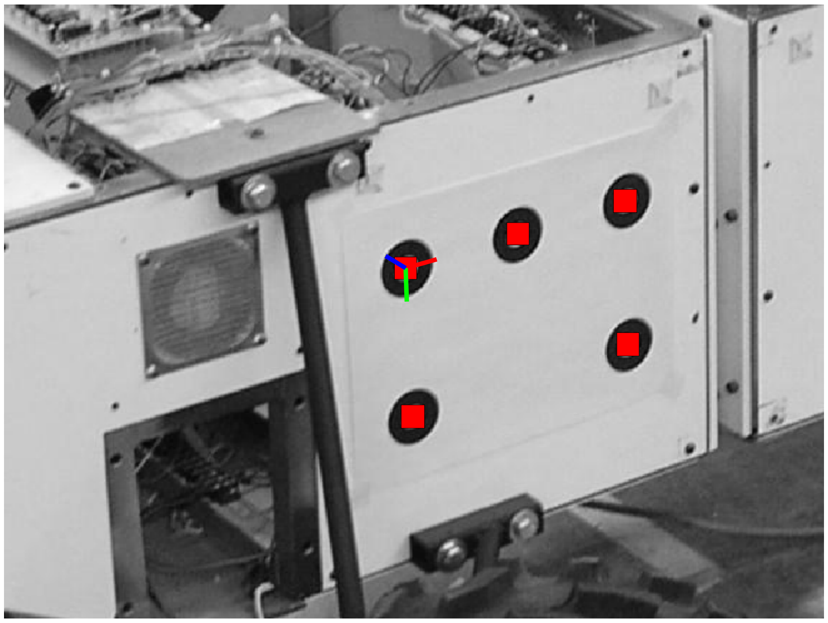
I used the same code for all images, with the initial guess of the pose being (ax,ay,az,tx,ty,tz) = [0; 0.0; 0.0; 0; 0; 80]. However, for image robot3_rect.jpg, I had to change ay=0.5, az=0.5 in order to get it to converge.

Here is the pose of the target with respect to the camera (in terms of XYZ angles and XYZ translation) in each image:

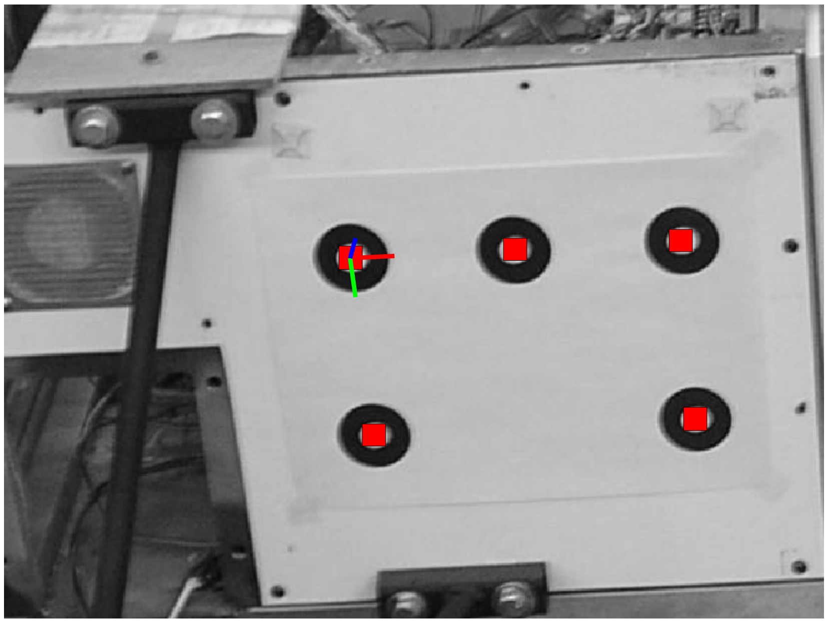
	robot1_rect.jpg	robot2_rect.jpg	robot3_rect.jpg
ax (radians)	0.5184	0.5151	0.8844
ay (radians)	-0.5588	0.0437	0.4503
az (radians)	-0.3099	-0.0494	0.7085
tx (inches)	-6.5826	-8.5849	-6.6759
ty (inches)	0.1988	-4.2004	-4.5470
tz (inches)	95.4803	78.0897	46.7477

Here is the overlay of the XYZ coordinate axes of the target on each image. (I have zoomed in so you can see the axes more clearly)

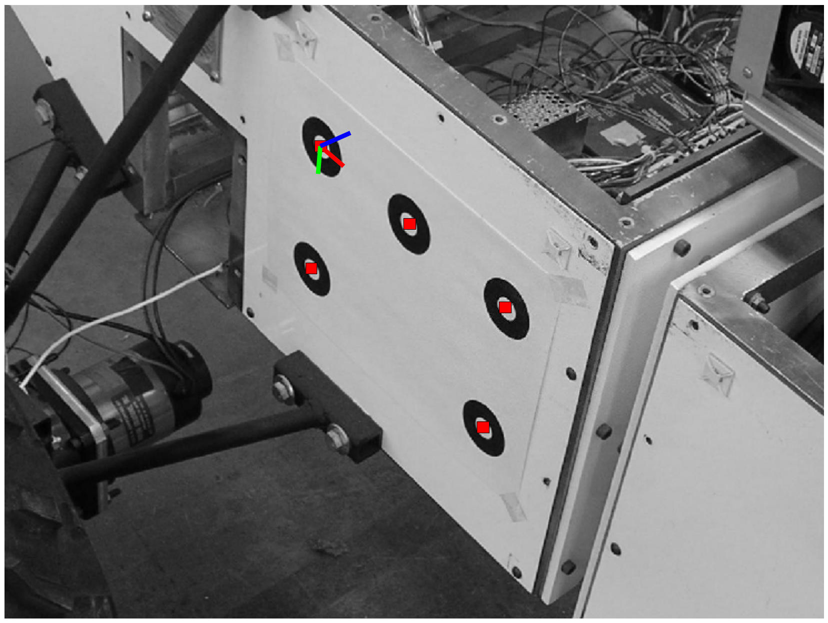
robot1_rect.jpg



robot2_rect.jpg



robot3_rect.jpg



4. **(25 pts) In the previous problem, you computed the Jacobian matrix numerically. It is also possible to compute the Jacobian symbolically (*i.e.*, by doing the derivatives by hand), although the expressions can be rather complicated. However, under certain conditions the Jacobian is easier to compute symbolically. In particular, if the rotation angles are close to zero[3], the rotation matrix can be simplified using the "small angle approximation", which is cos(θ) ≈ 1, sin(θ) ≈ θ, and $\theta * \theta \approx 0$.**

   (a) Write the 3x3 rotation matrix $\mathbf{R}(\theta_X, \theta_Y, \theta_Z)$, using the small angle approximation.

   (b) Write the equations for the function that projects a point $\mathbf{P} = (X, Y, Z)^T$ onto an image using both weak perspective and the small angle approximation.

   (c) Finally, write the equations for the Jacobian of that function with respect to the unknown parameters $\theta_X, \theta_Y, \theta_Z, t_x, t_y, Z_{avg}$; i.e., find the expressions in the matrix

$$\mathbf{J} = \begin{pmatrix} \partial x/\partial\theta_X & \partial x/\partial\theta_Y & \partial x/\partial\theta_Z & \partial x/\partial t_X & \partial x/\partial t_Y & \partial x/\partial Z_{avg} \\ \partial y/\partial\theta_X & \partial y/\partial\theta_Y & \partial y/\partial\theta_Z & \partial y/\partial t_X & \partial y/\partial t_Y & \partial y/\partial Z_{avg} \end{pmatrix}$$

Solution:

(a) Using the small angle approximation,

[3] For example, if you are tracking an object and are just computing the correction to its estimated pose.

$$\mathbf{R}_X(\theta_X) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_X & -\sin\theta_X \\ 0 & \sin\theta_X & \cos\theta_X \end{pmatrix} \approx \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -\theta_X \\ 0 & \theta_X & 1 \end{pmatrix}$$

$$\mathbf{R}_Y(\theta_Y) = \begin{pmatrix} \cos\theta_Y & 0 & \sin\theta_Y \\ 0 & 1 & 0 \\ -\sin\theta_Y & 0 & \cos\theta_Y \end{pmatrix} \approx \begin{pmatrix} 1 & 0 & \theta_Y \\ 0 & 1 & 0 \\ -\theta_Y & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_Z(\theta_Z) = \begin{pmatrix} \cos\theta_Z & -\sin\theta_Z & 0 \\ \sin\theta_Z & \cos\theta_Z & 0 \\ 0 & 0 & 1 \end{pmatrix} \approx \begin{pmatrix} 1 & -\theta_Z & 0 \\ \theta_Z & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}(\theta_X, \theta_Y, \theta_Z) = \mathbf{R}_Z(\theta_Z)\mathbf{R}_Y(\theta_Y)\mathbf{R}_X(\theta_X)$$

$$= \begin{pmatrix} 1 & -\theta_Z & 0 \\ \theta_Z & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & \theta_Y \\ 0 & 1 & 0 \\ -\theta_Y & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -\theta_X \\ 0 & \theta_X & 1 \end{pmatrix} = \begin{pmatrix} 1 & -\theta_Z & \theta_Y \\ \theta_Z & 1 & \theta_X \\ -\theta_Y & -\theta_X & 1 \end{pmatrix}$$

(b) To project a point onto an image using weak perspective projection, we use $\mathbf{p} = \mathbf{K}\,\mathbf{M}_{\mathbf{ext}}\,\mathbf{P}$, where

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

Using the small angle approximation, this is

$$\mathbf{M}_{\mathbf{ext}} = \begin{pmatrix} 1 & -\theta_Z & \theta_Y & t_X \\ \theta_Z & 1 & \theta_X & t_Y \\ 0 & 0 & 0 & Z_{avg} \end{pmatrix}$$

The standard camera intrinsic parameter matrix is

$$\mathbf{K} = \begin{pmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

So the projection is

$$\mathbf{p} = \mathbf{K}\,\mathbf{M}_{\mathbf{ext}}\,\mathbf{P} = \begin{pmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & -\theta_Z & \theta_Y & t_X \\ \theta_Z & 1 & \theta_X & t_Y \\ 0 & 0 & 0 & Z_{avg} \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \begin{pmatrix} f\left(X - \theta_Z Y + \theta_Y Z + t_X\right) + c_x Z_{avg} \\ f\left(\theta_Z X + Y - \theta_X Z + t_Y\right) + c_y Z_{avg} \\ Z_{avg} \end{pmatrix}$$

We still have to normalize the result by dividing through by the third element:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} f\,\dfrac{X - \theta_Z Y + \theta_Y Z + t_X}{Z_{avg}} + c_x \\ f\,\dfrac{\theta_Z X + Y - \theta_X Z + t_Y}{Z_{avg}} + c_y \end{pmatrix}$$

(c) Here is the Jacobian of a projected point (*x*,*y*) with respect to the unknown parameters $\theta_X, \theta_Y, \theta_Z, t_x, t_y, Z_{avg}$, assuming the world coordinates of the point (*X*,*Y*,*Z*) are known:

$$\mathbf{J} = \begin{pmatrix} \partial x/\partial\theta_X & \partial x/\partial\theta_Y & \partial x/\partial\theta_Z & \partial x/\partial t_X & \partial x/\partial t_Y & \partial x/\partial Z_{avg} \\ \partial y/\partial\theta_X & \partial y/\partial\theta_Y & \partial y/\partial\theta_Z & \partial y/\partial t_X & \partial y/\partial t_Y & \partial y/\partial Z_{avg} \end{pmatrix}$$

$$= \begin{pmatrix} 0 & f\,\dfrac{Z}{Z_{avg}} & f\,\dfrac{-Y}{Z_{avg}} & f\,\dfrac{1}{Z_{avg}} & 0 & -f\,\dfrac{X - \theta_Z Y + \theta_Y Z + t_X}{Z_{avg}^2} \\ f\,\dfrac{-Z}{Z_{avg}} & 0 & f\,\dfrac{X}{Z_{avg}} & 0 & f\,\dfrac{1}{Z_{avg}} & -f\,\dfrac{\theta_Z X + Y - \theta_X Z + t_Y}{Z_{avg}^2} \end{pmatrix}$$

$$= \frac{f}{Z_{avg}} \begin{pmatrix} 0 & Z & -Y & 1 & 0 & -\dfrac{X - \theta_Z Y + \theta_Y Z + t_X}{Z_{avg}} \\ -Z & 0 & X & 0 & 1 & -\dfrac{\theta_Z X + Y - \theta_X Z + t_Y}{Z_{avg}} \end{pmatrix}$$