CMPT 412 3D Reconstruction

Huiyi Zou 301355563

3.1.1 Implement the eight point algorithm

The *eightpoint* function:

```
function F = eightpoint(pts1, pts2, M)
    pts1 = pts1/M;
    pts2 = pts2/M;
    s = [1/M, 0, 0; 0, 1/M, 0; 0, 0, 1];
    A = [];
    for i = 1:size(pts1,1)
        temp = [pts2(i,1)*pts1(i,1), pts2(i,1)*pts1(i,2), pts2(i,1), pts2(i,2)*pts1(i,1),...
                pts2(i,2)*pts1(i,2), pts2(i,2), pts1(i,1), pts1(i,2), 1];
        A = [A; temp];
    end
    [~,~,V] = svd(A);
    F = reshape(V(:,end), 3, 3).';
    [U,S,V] = svd(F);
    new_S = S;
    new_S(3,3) = 0;
    new_F = U * new_S * V.';
    refined_F = refineF(new_F, pts1, pts2);
    F = s.' * refined_F * s;
end
```

The recovered F

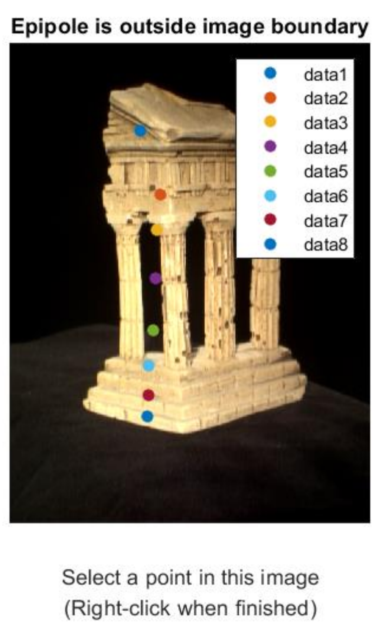
F

3x3 double

	1	2	3
1	1.7518e-09	-1.8667e-08	-8.5202e-06
2	-6.4567e-08	-4.0214e-10	4.9568e-04
3	1.6635e-05	-4.7610e-04	-0.0021

```
F =

    0.0000   -0.0000   -0.0000
   -0.0000   -0.0000    0.0005
    0.0000   -0.0005   -0.0021
```

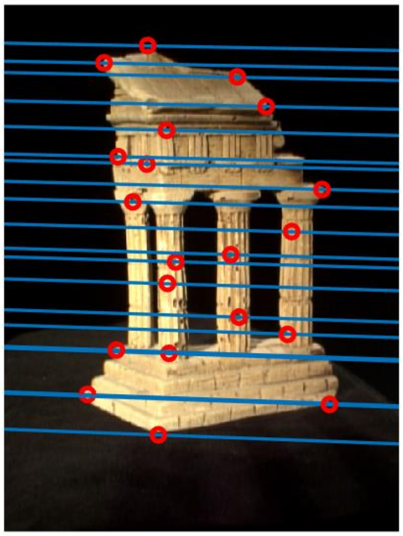


3.1.2 Find epipolar correspondences

The *epipolarCorrespondence* function

```
function [pts2] = epipolarCorrespondence(im1, im2, F, pts1)
    pad = 8;
    padded_im1 = padarray(im1, [pad pad], 0, 'both');
    padded_im2 = padarray(im2, [pad pad], 0, 'both');
    pts2 = [];
    for i = 1:size(pts1, 1)
        x1 = pts1(i,1);
        y1 = pts1(i,2);
        line = F * [x1; y1; 1];
        window_im1 = double(padded_im1(y1:y1+2*pad,x1:x1+2*pad));
        min_distance = Inf;
        for j = 1:size(im2, 2)
            y = round((-line(1)*j-line(3)) / line(2));
            window_im2 = double(padded_im2(y:y+2*pad,j:j+2*pad));
            distance = norm(window_im2 - window_im1);
            if distance < min_distance
                min_distance = distance;
                candidate = [j, y];
            end
        end
        pts2 = [pts2; candidate];
    end
end
```

After generated a set of candidate points in the second image, I set a window of size 17 around the point x and a window around each candidate point x' and calculated the Euclidean distance between the points in the window around x and the points in the window around x′ for similarity. The candidate points with minimum Euclidean distance would be the corresponding points in the second image.
The points around corners, dots, and unique shapes were correctly matched. But if points were around the part that had some similar patterns along with its corresponding epipolar line, there was a mistake. It might because those parts were very similar in the windows that had a low and close Euclidean distance.



3.1.3 Write a function to compute the essential matrix

The *essentialMatrix* function

```
function E = essentialMatrix(F, K1, K2)
% essentialMatrix computes the essential matrix
%   Args:
%       F:  Fundamental Matrix
%       K1: Camera Matrix 1
%       K2: Camera Matrix 2
%
%   Returns:
%       E:  Essential Matrix
    E = K2.' * F * K1;
end
```

The estimated E

E

3x3 double

	1	2	3
1	0.0040	-0.0433	-0.0192
2	-0.1498	-9.3633e-04	0.7264
3	0.0019	-0.7352	-8.4658e-04

```
E =

    0.0040   -0.0433   -0.0192
   -0.1498   -0.0009    0.7264
    0.0019   -0.7352   -0.0008
```

3.1.4 Implement triangulation

The *triangulate* function

```
function pts3d = triangulate(P1, pts1, P2, pts2 )
    pts3d = [];
    for i = 1:size(pts1,1)
        A = [pts1(i,2)*P1(3,:) - P1(2,:);
             P1(1,:) - pts1(i,1)*P1(3,:);
             pts2(i,2)*P2(3,:) - P2(2,:);
             P2(1,:) - pts2(i,1)*P2(3,:)];
        [~,~,V] = svd(A);
        curr_3d = V(1:3,end).'/V(end);
        pts3d = [pts3d; curr_3d];
    end
end
```

```
P1 = [eye(3), zeros(3,1)];
P2_candidates = camera2(E);
max_positive = 0;
correct_P2 = 0;
for n = 1:4
    P2 = P2_candidates(:,:,n);
    temp_pts3d = triangulate(K.K1*P1, S.pts1, K.K2*P2, S.pts2);
    err1 = 0;
    err2 = 0;
    positive = 0;
    for i = 1:size(temp_pts3d,1)
        pt1 = K.K1 * P1 * [temp_pts3d(i,:),1].';
        if pt1(3) > 0
            positive = positive + 1;
        end
        pt1 = pt1(1:2)'/pt1(3);
        err1 = err1 + norm(pt1-S.pts1(i,:));
        pt2 = K.K2 * P2 * [temp_pts3d(i,:),1].';
        if pt2(3) > 0
            positive = positive + 1;
        end
        pt2 = pt2(1:2).'/pt2(3);
        err2 = err2 + norm(pt2-S.pts2(i,:));
    end
    if positive >= max_positive
        max_positive = positive;
        correct_P2 = P2;
        pts3d = temp_pts3d;
        mean_err1 = err1 / size(temp_pts3d,1);
        mean_err2 = err2 / size(temp_pts3d,1);
    end
end
```

To find the correct extrinsic matrix, I counted the number of positive z of re-projection results with each candidate matrix. The matrix with the most number of positive z was the correct one.

The re-projection errors were:

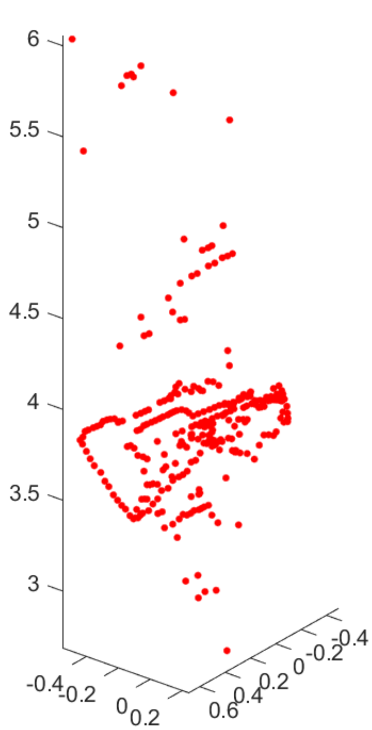
```
>> test_part3
Re-pojection error for pts1 is 0.6764
Re-pojection error for pts2 is 0.6812
```

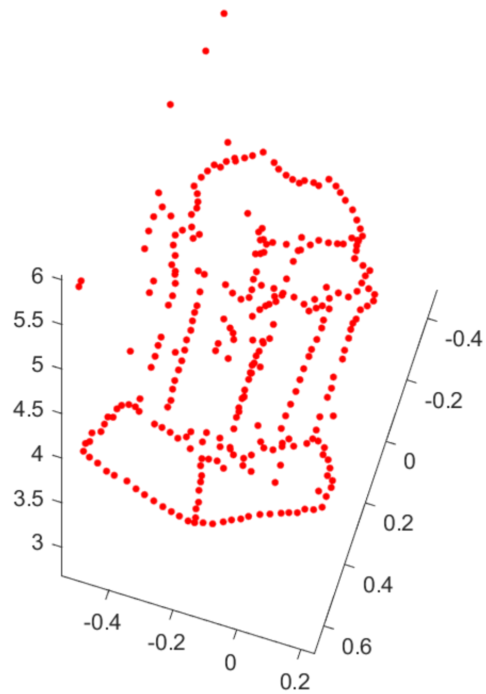
3.1.5 Write a test script that uses testTempleCoords

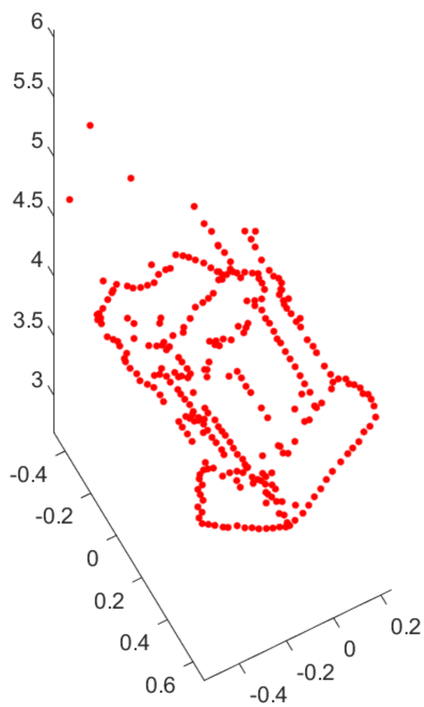
```
%% load data
im1 = imread('../data/im1.png');
im2 = imread('../data/im2.png');
points = load('../data/someCorresp.mat');
%% run eightpoint
F = eightpoint(points.pts1, points.pts2, points.M);
%displayEpipolarF(im1, im2, F);
%% get corrsponding points
temple = load('../data/templeCoords.mat');
temple_pts2 = epipolarCorrespondence(im1, im2, F, temple.pts1);
%[coordsIM1, coordsIM2] = epipolarMatchGUI(im1, im2, F);
%% essential matrix
intrinsics = load('../data/intrinsics.mat');
E = essentialMatrix(F, intrinsics.K1, intrinsics.K2);
%% project matrix
P1 = [eye(3), zeros(3,1)];
P2_candidates = camera2(E);
%% get correct P2
max_positive = 0;
correct_P2 = 0;
for n = 1:4
    P2 = P2_candidates(:,:,n);
    curr_pts3d = triangulate(intrinsics.K1*P1, temple.pts1, intrinsics.K2*P2, temple_pts2);
    positive = 0;
    for i = 1:size(curr_pts3d,1)
        x1 = intrinsics.K1 * P1 * [curr_pts3d(i,:),1].';
        if x1(3) > 0
            positive = positive + 1;
        end

        x2 = intrinsics.K2 * P2 * [curr_pts3d(i,:),1].';
        if x2(3) > 0
            positive = positive + 1;
        end
    end
    if positive >= max_positive
        max_positive = positive;
        correct_P2 = P2;
        pts3d = curr_pts3d;
    end
end
%% plot 3d points
plot3(pts3d(:,1), pts3d(:,2), pts3d(:,3), 'r.', 'MarkerSize', 10);
axis equal;
%% save extrinsic parameters
R1 = P1(:,1:3);
R2 = correct_P2(:,1:3);
t1 = P1(:,4);
t2 = correct_P2(:,4);
save('../data/extrinsics.mat', 'R1', 't1', 'R2', 't2');
```

Three images of final reconstruction of the templeCoords points



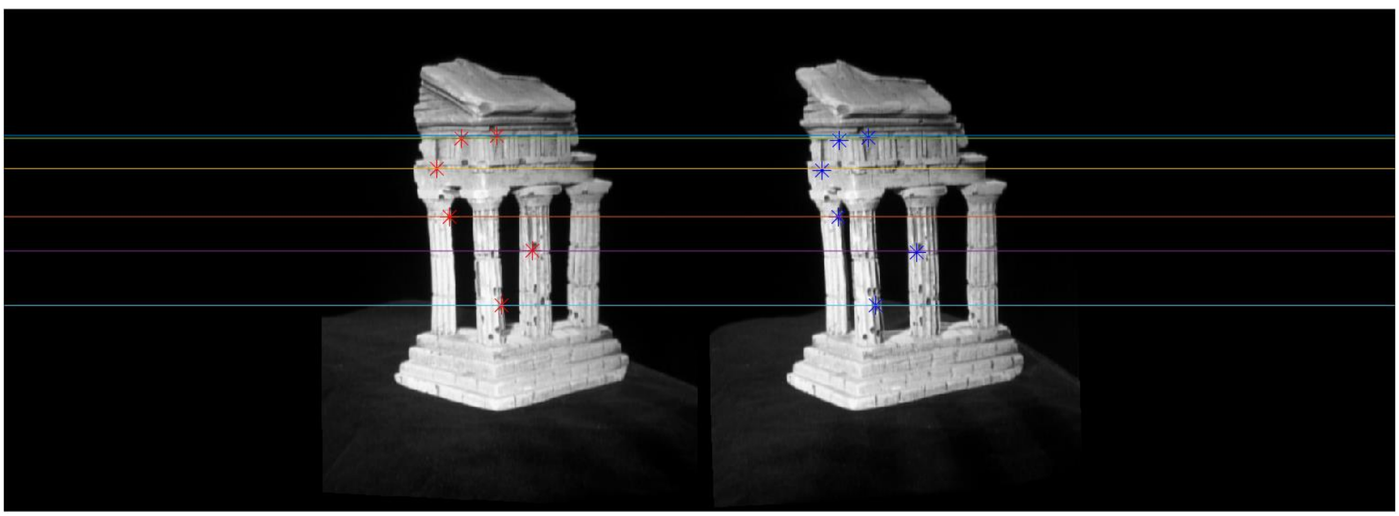




3.2.1 Image rectification

The $rectify\_pair$ function

```
function [M1, M2, K1n, K2n, R1n, R2n, t1n, t2n] = ...
                        rectify_pair(K1, K2, R1, R2, t1, t2)
    % center
    c1 = -inv(K1*R1) * (K1*t1);
    c2 = -inv(K2*R2) * (K2*t2);
    % rotation
    r1 = (c1-c2) / norm(c1-c2);
    r2 = cross(R1(3,:).', r1);
    r3 = cross(r2, r1);
    R = [r1, r2, r3].';
    R1n = R;
    R2n = R;
    % new K
    K = K2;
    K1n = K;
    K2n = K;
    % new translation
    t1n = -R * c1;
    t2n = -R * c2;
    M1 = (K*R) / (K1*R1);
    M2 = (K*R) / (K2*R2);
end
```



3.2.2 Dense window matching to find per pixel density

The $get\_disparity$ function

```
function dispM = get_disparity(im1, im2, maxDisp, windowSize)
% GET_DISPARITY creates a disparity map from a pair of rectified images im1 and
%   im2, given the maximum disparity MAXDISP and the window size WINDOWSIZE
    all_dispM(:,:,1) = conv2((double(im1)-double(im2)).^2, ones(windowSize), 'same');
    for d = 1:maxDisp
        temp = circshift(im2, d, 2);
        temp(:,1:d) = Inf;
        all_dispM(:,:,d+1) = conv2((double(im1)-double(temp)).^2, ones(windowSize), 'same');
    end
    [~, index] = min(all_dispM, [], 3);
    dispM = index - 1;
end
```

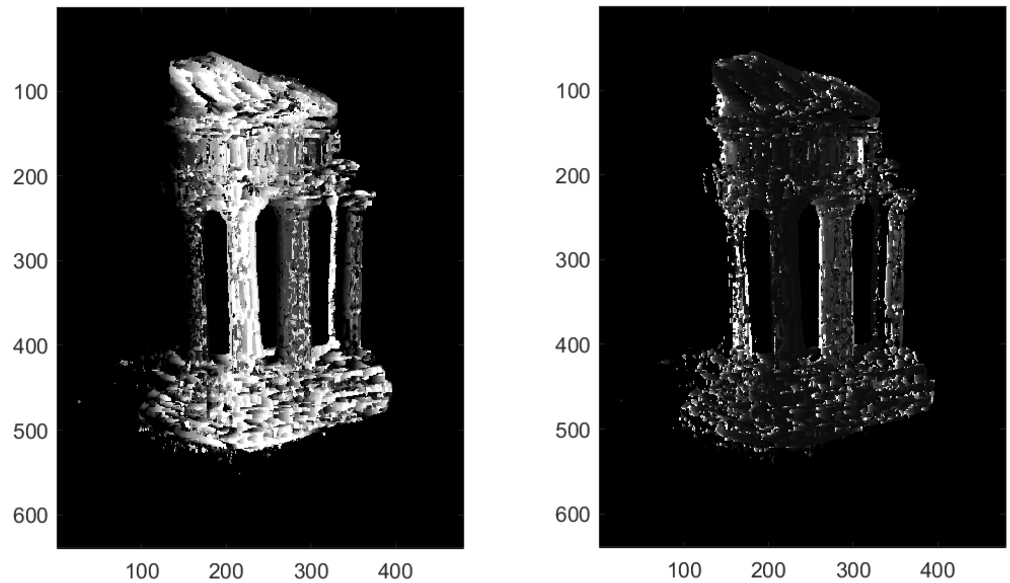
I used the $conv2$ function to compute their distance. For each disparity d, each row of im2 was circularly shifted d position to the right by $circshift$ function. After the circular shift, I set the values of first d columns to be infinite, which would not affect any minimum distance.

3.2.3 Depth map

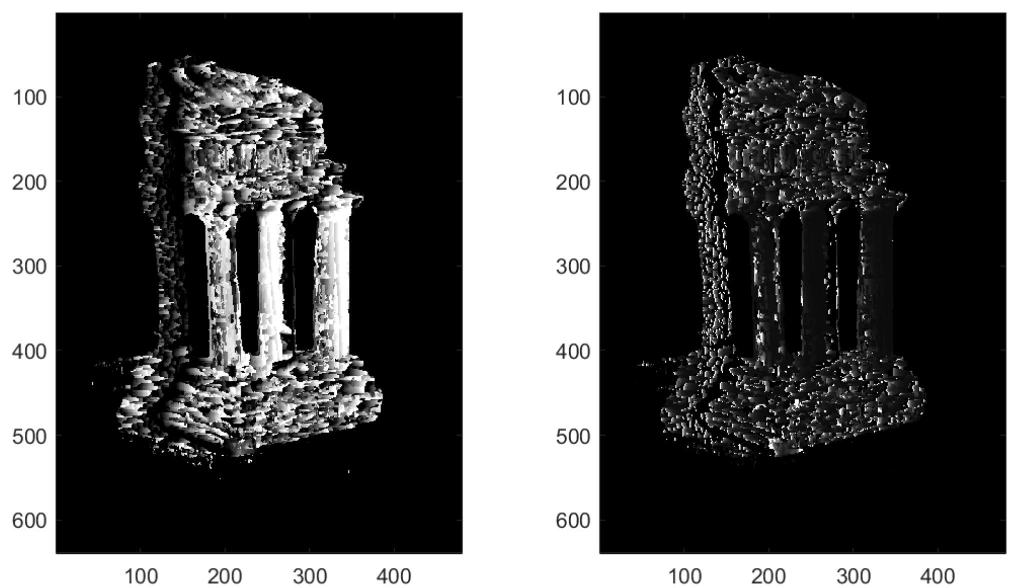
The get_depth function

```
function depthM = get_depth(dispM, K1, K2, R1, R2, t1, t2)
% GET_DEPTH creates a depth map from a disparity map (DISPM).
    c1 = -inv(K1*R1) * (K1*t1);
    c2 = -inv(K2*R2) * (K2*t2);
    b = norm(c1-c2);
    f = K1(1,1);
    depthM = b * f ./ dispM;
    depthM(dispM==0) = 0;
end
```

The disparity map (left) and the depth map (right) before rectification



The disparity map (left) and the depth map (right) after rectification



3.3.1 Estimate camera matrix P

The *estimate_pose* function

```
function P = estimate_pose(x, X)
% ESTIMATE_POSE computes the pose matrix (camera matrix) P given 2D and 3D
% points.
%   Args:
%       x: 2D points with shape [2, N]
%       X: 3D points with shape [3, N]
A=[];
for i = 1:size(x,2)
    trans_X = [X(:,i); 1].';
    temp = [trans_X, zeros(1,4), -x(1,i)*trans_X;
            zeros(1,4), trans_X, -x(2,i)*trans_X];
    A = [A;temp];
end
[~, ~, V] = svd(A);
P = reshape(V(:, end), 4, 3).';
end
```

The output of the script testPose

```
>> testPose
Reprojected Error with clean 2D points is 0.0000
Pose Error with clean 2D points is 0.0000
------------------------------
Reprojected Error with noisy 2D points is 2.8988
Pose Error with noisy 2D points is 0.0085
```

3.3.2 Estimate intrinsic/extrinsic parameters

The $estimate\_params$ function

```
function [K, R, t] = estimate_params(P)
% ESTIMATE_PARAMS computes the intrinsic K, rotation R and translation t from
% given camera matrix P.
    [~, ~, V] = svd(P);
    c = V(1:3, end) / V(end);
    A = flip(eye(3));
    P = A * P(:,1:3);
    [Q, R] = qr(P.');
    K = A * R.' * A;
    R = A * Q.';

    D = diag(sign(diag(K)));
    K = K * D;
    R = D * R;

    if round(det(R))==-1
        R = -R;
    end
    t = -R * c;
end
```

For this part, I referred to those pages:
https://math.stackexchange.com/questions/1640695/rq-decomposition
https://ksimek.github.io/2012/08/14/decompose/

There was no unique result from RQ-decomposition. I got the signs of diagonal of K and enforced the positive diagonal for a unique solution, which might make matrix R had a determinant of -1 instead of 1. Thus, if the determinant of R is -1, then negate R. Here, I used the $round$ function because $det(R)$ is floating-point number that may not be exactly equal to an integer.

The output of the script testKRt

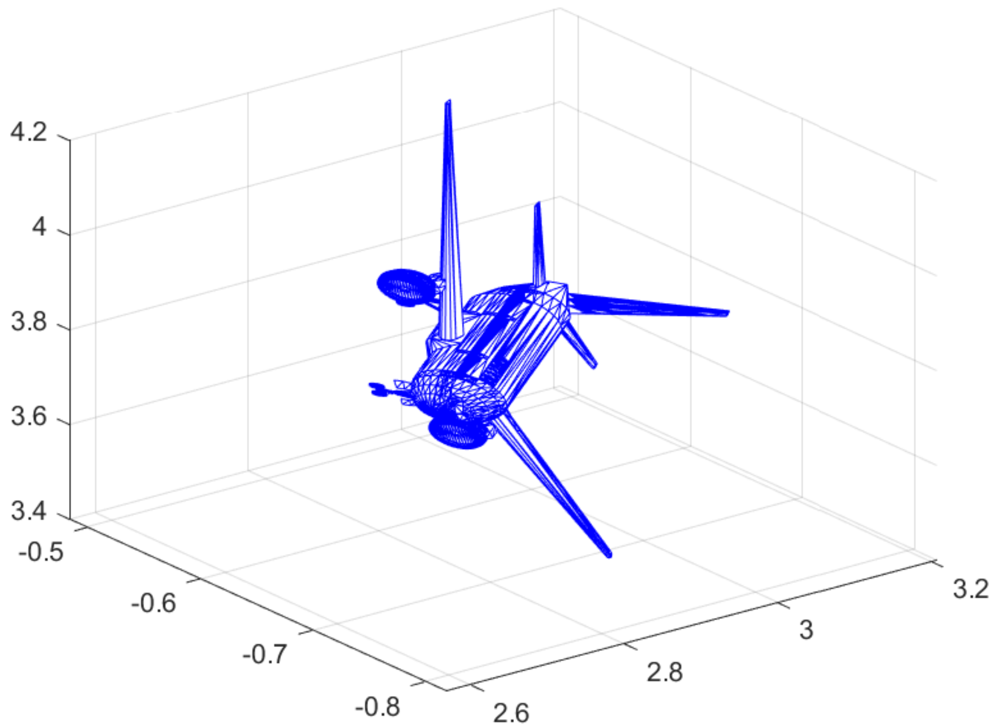
```
>> testKRt
Intrinsic Error with clean 2D points is 0.0000
Rotation Error with clean 2D points is 0.0000
Translation Error with clean 2D points is 0.0000
-------------------------------
Intrinsic Error with noise 2D points is 0.8276
Rotation Error with noise 2D points is 0.0722
Translation Error with noise 2D points is 0.1382
```

3.3.3 Project a CAD model to the image

```
% load image
load('../data/PnP.mat');
% estimate matrices
P = estimate_pose(x, X);
[K, R, t] = estimate_params(P);
% project X onto image
projected_X = P * [X; ones(1, size(X,2))];
projected_X = projected_X(1:2,:) ./ projected_X(3,:);
% plot points
figure;
imshow(image); hold on;
plot(x(1,:), x(2,:), 'greeno', 'MarkerSize', 15);
plot(projected_X(1,:), projected_X(2,:), 'black.', 'MarkerSize', 10);
% rotate and translate CAD
rotated_vertices = R * cad.vertices.' + t;
figure;
trimesh(cad.faces, rotated_vertices(1,:), rotated_vertices(2,:), rotated_vertices(3,:), 'edgecolor', 'blue');
% project CAD model
projected_model = P * [cad.vertices'; ones(1, size(cad.vertices,1))];
projected_model = projected_model(1:2,:) ./ projected_model(3,:);
figure;
imshow(image); hold on;
patch('Faces', cad.faces, 'Vertices', projected_model.', 'FaceColor', 'red', 'FaceAlpha', .2, 'EdgeColor', 'None');
```

The image annotated with given 2D points (green circle) and projected 3D points (black points)

The CAD model rotated by R and translated by t



The image overlapping with projected CAD model