Visual Perception
Lab 2 Report: Camera Calibration

Mohammad Rami Koujan
M.Sc. VIBOT
University of Girona

March 26, 2016

1 Introduction and Problem Definition

The objective of this lab practice is to be familiar with camera calibration problem by calibrating a simulated camera using more than one method, comparing those methods, and checking the accuracy of the results against noise. Camera Calibration is, in fact, the process of finding the various physical parameters that relate the world and camera coordinate frames. It has many applications, e.g. dense reconstruction, visual inspection, object localization, camera localization, and so on.

2 Analysis of the Camera Calibration steps

This section presents the different steps that were followed in this assignment in details. In the first step it was necessary to define the intrinsic and extrinsic parameters that are used in the following steps to compute the camera matrix. Those parameters are defined as follow:

- au=557.0943; av=712.9824; u0=326.3819; v0=298.6679;
- f=80;Tx=100; Ty=0; Tz=1500;
- Phix=0.8*pi/2; Phiy=-1.8*pi/2; Phix1=pi/5;

In the second step, the extrinsic matrix was computed from the rotation matrix R and translation vector T, where the rotation matrix was calculated from the Euler XYX matrices with rotation angles equal to Phix, Phiy, and Phix1 respectively and the translation vector from the Tx, Ty, Tz parameters defined in the previous step. However, the intrinsic matrix is defined by the internal geometry parameters of the camera, αu, αv, $u0$, $v0$.

- Rotx=[1 0 0;0 cos(Phix) -sin(Phix);0 sin(Phix) cos(Phix)];
- Roty=[cos(Phiy) 0 sin(Phiy) ;0 1 0;-sin(Phiy) 0 cos(Phiy)];
- Rotx1=[1 0 0;0 cos(Phix1) -sin(Phix1);0 sin(Phix1) cos(Phix1)];
- T=[Tx;Ty;Tz];
- R=Rotx*Roty*Rotx1;
- ext=[R(1,:) T(1);R(2,:) T(2);R(3,:) T(3);0 0 0 1];
- Int=[au 0 u0 0; 0 av v0 0;0 0 1 0];
- % output

$$Int = \begin{bmatrix} 557.09 & 0 & 326.38 & 0 \\ 0 & 712.98 & 298.67 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$ext = \begin{bmatrix} -0.95106 & -0.18164 & -0.25 & 100 \\ -0.29389 & 0.78166 & 0.55013 & 0 \\ 0.095492 & 0.59668 & -0.79678 & 1500 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

to make sure that the extrinsic matrix is calculated correctly, a multiplication by its inverse should give the identity matrix.

$$ext * inv(ext) = \begin{bmatrix} 1 & 0 & 2.7756e-17 & -2.8422e-14 \\ 0 & 1 & 5.5511e-17 & -1.1369e-13 \\ 2.7756e-17 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

which is almost the same as the identity.

Then, in the third step the idea is to generate 3D points randomly, at least six points which is explained later, in the interval -480:480. In order to do so, the matlab function "randi" was used, where it generates integers from a uniform discrete distribution on a given interval and for specific matrix size.

- Num_points=6; % Number of 3D points
- points=randi([-480,480],3,Num_points); % ouput matrix, points, size is 3*Num_points
- % output

$$points = \begin{bmatrix} -327 & 293 & -135 & -234 & 5 & 373 \\ 269 & 60 & -123 & 454 & -290 & -132 \\ -310 & -29 & -113 & 154 & -380 & -44 \end{bmatrix}$$

Each column of "points" matrix represents one point and the three rows are the x,y, and z coordinates of those points.

In the fourth step, the projection of the previously generated 3D points on the image plane was computed by using the camera transformation matrix. This matrix was generated from the multiplication of the intrinsic by extrinsic matrices found before. Hence, multiplying the transformation matrix by the randomly generated 3D points gives the projected 2D points.

- points=[points;ones(1,Num_points)]; % homogeneous coordinates of the points
- camera_mat=Int*ext;
- proj=camera_mat*points;
- % output

$$camera\_mat = \begin{bmatrix} -498.66 & 93.556 & -399.33 & 5.4528e+05 \\ -181.02 & 735.52 & 154.26 & 4.48e+05 \\ 0.095492 & 0.59668 & -0.79678 & 1500 \end{bmatrix}$$

$$proj = \begin{bmatrix} 456.92 & 262.38 & 429.74 & 395.45 & 409.39 & 244.32 \\ 350.28 & 273.88 & 242.42 & 521.6 & 107.46 & 185.4 \\ 1876.3 & 1586.9 & 1503.8 & 1625.8 & 1630.2 & 1491.9 \end{bmatrix}$$

In the fifth step, those projected points were normalized by s and then plotted in a 2D window in Matlab.

- for i=1:size(proj,2)
- proj(1:2,i)=proj(1:2,i)/proj(3,i);
- end
- scatter(proj(1,:),proj(2,:));
- hold on;
- title('Projected 2D points');

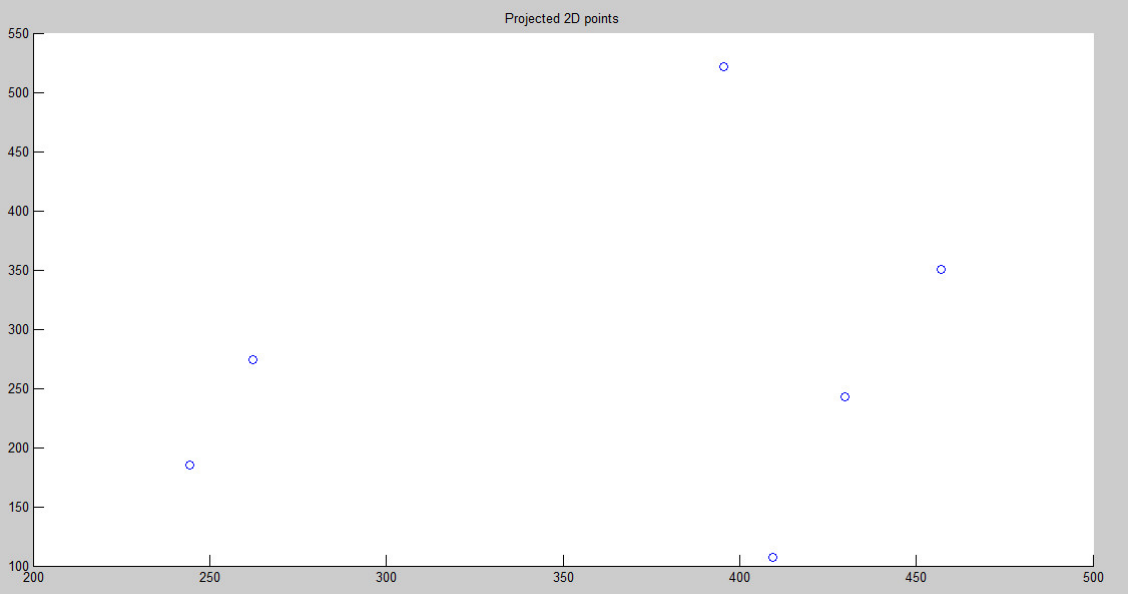


Figure 1: Projected 2D points

As it is shown in figure 1, the points are quite spread in the image plane. In fact, the accuracy of computations are not affected by the distribution of this points, except if they are collinear.

computing the transformation matrix according to hall method was performed in this step based on the 3D and 2D points obtained in the preceding steps.

- Q=zeros(size(points,2)*2,11); % every point gives two equations
- B=zeros(size(points,2)*2,1);
- for i=1:size(proj,2)
- Q(2*i-1,:)= [points(1,i),points(2,i), points(3,i),1,0,0,0,0,-proj(1,i)*points(1,i),.....
- -proj(1,i)*points(2,i),-proj(1,i)*points(3,i)];
- Q(2*i,:)=[0,0,0,0,points(1,i),points(2,i), points(3,i),1,-proj(2,i)*points(1,i),.....
- -proj(2,i)*points(2,i),-proj(2,i)*points(3,i)];
- B(2*i-1)=proj(1,i);
- B(2*i)=proj(2,i);
- end
- Hall_mat=Q\B; % vector of size 11*1
- Hall_mat=[Hall_mat; 1];
- Hall_mat=[Hall_mat(1:4)';Hall_mat(5:8)';Hall_mat(9:12)'];
- % output

$$Hall\_mat = \begin{bmatrix} -0.33244 & 0.062371 & -0.26622 & 363.52 \\ -0.12068 & 0.49034 & 0.10284 & 298.67 \\ 6.3661e-05 & 0.00039778 & -0.00053119 & 1 \end{bmatrix}$$

Finding the transformation matrix by method of Hall is equivalent to solve the system

$$A = (Q^t * Q)^{-1} * Q^t * B$$

Therefore, The idea is to calculate in a for loop the Q matrix, which has a size of twice the number of 3D points, in this case 12, by 11, based on the 2D projected points and the 3D points. This explains why we need at least six points since we have 11 unknowns and every point gives us two equations. B was also calculated in the same loop using the 2D points coordinates. Then, using the pseudo-inverse (left matrix division in Matlab), the Hall matrix was obtained after reshaping it and adding the normalized 12th element.

Comparing Hall matrix with the camera matrix computed in step 2 shows that they are rather the same.

- camera_mat=camera_mat./camera_mat(3,4);
- diff=Hall_mat-camera_mat;
- ave=sum(diff(:))/12; % mean
- SD=sqrt(var(diff(:))); % standard deviation
- % output

$$diff = \begin{bmatrix} 3.1641e-15 & 3.0323e-15 & -3.8858e-15 & -2.8422e-13 \\ 2.3453e-15 & 2.387e-15 & -2.609e-15 & -5.6843e-14 \\ 1.0178e-17 & 7.5894e-18 & -8.4568e-18 & 0 \end{bmatrix}$$

$$ave = -2.8051e-14$$

$$SD = 8.2363e-14$$

Step 8 discusses the effect of adding noise to the 2D points on the accuracy of the computed Hall transformation matrix. First, Gaussian noise was generated by using the "randn" Matlab function, with size equivalent to the 2D points, which generates Gaussian points with zero mean and unit standard deviation. Since it is required to add Gaussian points with 95% in the range [-1,1] and 95% of the generated points is between -2σ and 2σ, the function is multiplied by 0.5.

- Noise=0.5*randn(size(proj));
- proj_n=proj+Noise;

Second, the Hall matrix was computed again in the same previously described way but with the noisy 2D points instead.

- Q_n=zeros(size(points,2)*2,11);
- B_n=zeros(size(points,2)*2,1);
- for i=1:size(proj,2)
- Q_n(2*i-1,:)= [points(1,i),points(2,i), points(3,i),1,0,0,0,0,-proj_n(1,i)*points(1,i),.....
- -proj_n(1,i)*points(2,i),-proj_n(1,i)*points(3,i)];
- Q_n(2*i,:)=[0,0,0,0,points(1,i),points(2,i), points(3,i),1,-proj_n(2,i)*points(1,i),.....
- -proj_n(2,i)*points(2,i),-proj_n(2,i)*points(3,i)];
- B_n(2*i-1)=proj_n(1,i);
- B_n(2*i)=proj_n(2,i);
- end
- Hall_mat_n=Q_n\B_n;
- Hall_mat_n=[Hall_mat_n; 1];
- Hall_mat_n=[Hall_mat_n(1:4)';Hall_mat_n(5:8)';Hall_mat_n(9:12)'];
- diff_n=Hall_mat_n-Hall_mat;
- ave_n=sum(diff_n(:))/12;

- SD_n=sqrt(var(diff_n(:)));
- % output

$$diff\_n = \begin{bmatrix} 0.010746 & 0.021113 & -0.018897 & -0.48889 \\ 0.0065543 & 0.016305 & -0.010987 & 1.0304 \\ 3.1592e-05 & 5.4629e-05 & -4.879e-05 & 0 \end{bmatrix}$$

$$ave\_n = 0.047197$$

$$SD\_n = 0.3405$$

The diff_n, ave_n, and SD_n parameters show that there is a difference between the two transformation matrices before and after adding noise. This difference, of course, depends on the range of the added noise. The last part of this step is to compute again the 2D points from the noisy matrix and compare them to the original ones.

- proj_n_r=Hall_mat_n*points;
- for i=1:size(proj,2)
- proj_n_r(1:2,i)=proj_n_r(1:2,i)/proj_n_r(3,i); % normalization by s
- end
- dis_norm=zeros(1,Num_points);
- for i=1:size(proj,2)
- dis_norm(i)=norm([(proj(1,i)-proj_n_r(1,i)),(proj(2,i)-proj_n_r(2,i))]);
- end
- ave_n2=mean(dis_norm);
- SD_n2=sqrt(var(dis_norm));
- % output

$$ave\_n2 = 0.66313$$

$$SD\_n2 = 0.29549$$

To compute those points, simply the noisy Hall matrix was multiplied by the 3D points. A "for" loop was used to normalize the obtained 2D points by s and then the distance between the noisy and non-noisy points was computed by using the norm function after passing the difference between them to it. Finally, to compare between the two sets of 2D points, the mean and standard deviation were calculated. The results demonstrate that the 2D points are not the same after adding the noise. Furthermore, increasing the number of 3D points has the effect of getting more accurate matrix. The following are the mean and standard deviation values for 10 and 50 points respectively.

$$ave\_10pts = 0.3595$$

$$SD\_10pts = 0.2762$$

$$ave\_50pts = 0.26999$$

$$SD\_50pts = 0.13724$$

Vector X of the method of Faugeras can be defined, as it is required in step 10 (part 2), from the following equation

$$X = (Q^t * Q)^{-1} * Q^t * B$$

where Q and B can be found in a "for" loop from the 2D and 3D points obtained before as follow.

- Q_faug=zeros(size(points,2)*2,11);
- B_faug=zeros(size(points,2)*2,1);
- for i=1:size(proj,2)

- Q_faug(2*i-1,:)= [points(1,i),points(2,i), points(3,i),-proj(1,i)*points(1,i),.....
- -proj(1,i)*points(2,i), -proj(1,i)*points(3,i),0,0,0,1,0];
- Q_faug(2*i,:)=[0,0,0,-proj(2,i)*points(1,i),-proj(2,i)*points(2,i),-proj(2,i)*points(3,i),.....
- points(1,i), points(2,i), points(3,i),0,1];
- B_faug(2*i-1)=proj(1,i);
- B_faug(2*i)=proj(2,i);
- end
- X=Q_faug\B_faug;
- % output

$$X = \begin{bmatrix} -0.33244 \\ 0.062371 \\ -0.26622 \\ 6.3661e-05 \\ 0.00039778 \\ -0.00053119 \\ -0.12068 \\ 0.49034 \\ 0.10284 \\ 363.52 \\ 298.67 \end{bmatrix}$$

here X was calculated by using the pseudo-inverse operator of Matlab. Thereafter, the camera parameters can be extracted by applying the same theoretical equations that depend mainly on calculating the dot and cross product between T's and C's parameters ,that constitute the X vector, as shown below.

- T1=X(1:3);
- T2=X(4:6);
- T3=X(7:9);
- C1=X(10);
- C2=X(11);
- $u01 = dot(T1, T2)/(norm(T2)^2)$;
- $v01 = dot(T2, T3)/(norm(T2)^2)$;
- $au1 = norm(cross(T1, T2))/(norm(T2)^2)$;
- $av1 = norm(cross(T2, T3))/(norm(T2)^2)$;
- $r1 = (norm(T2)/norm(cross(T1, T2))) * (T1 - ((dot(T1, T2)/(norm(T2)^2)) * T2))$;
- $r2 = (norm(T2)/norm(cross(T2, T3))) * (T3 - ((dot(T2, T3)/(norm(T2)^2)) * T2))$;
- $r3 = T2/norm(T2)$;
- $tx = (norm(T2)/norm(cross(T1, T2))) * (C1 - (dot(T1, T2)/(norm(T2)^2)))$;
- $ty = (norm(T2)/norm(cross(T2, T3))) * (C2 - (dot(T2, T3)/(norm(T2)^2)))$;
- $tz = 1/norm(T2)$;
- faug_mat=[T1'*tz C1*tz;T3'*tz C2*tz;T2'*tz tz];
- % output

$$faug\_mat = \begin{bmatrix} -498.66 & 93.556 & -399.33 & 5.4528e+05 \\ -181.02 & 735.52 & 154.26 & 4.48e+05 \\ 0.095492 & 0.59668 & -0.79678 & 1500 \end{bmatrix}$$

The last line shows the Faugeras matrix which can be defined using directly T's, C's and tz or from combining the extracted parameters in intrinsic and extrinsic matrices and multiply them together. In order to compare the extracted parameters with the one in step 1, the differences were calculated below.

- R_diff=R-[r1';r2';r3'];
- T_diff=[Tx;Ty;Tz]-[tx;ty;tz];
- u0_diff=u01-u0;
- v0_diff=v01-v0;
- au_diff=au1-au;
- av_diff=av1-av;
- % output

$$R\_diff = \begin{bmatrix} 0 & -8.3267e-17 & 3.8858e-16 \\ 3.3307e-16 & -4.4409e-16 & 8.8818e-16 \\ -5.8287e-16 & 7.7716e-16 & 4.4409e-16 \end{bmatrix}$$

$$T\_diff = \begin{bmatrix} -1.5206e-12 \\ -7.1754e-13 \\ -4.5475e-12 \end{bmatrix}$$

$$u0\_diff = \begin{bmatrix} -4.5475e-13 \end{bmatrix}$$

$$v0\_diff = \begin{bmatrix} -3.979e-13 \end{bmatrix}$$

$$au\_diff = \begin{bmatrix} 1.4779e-12 \end{bmatrix}$$

$$av\_diff = \begin{bmatrix} 2.3874e-12 \end{bmatrix}$$

The results reveal that they are pretty much the same. In the next step, 11, a Gaussian noise was added to the 2D points, with 95 % of the noisy points in the ranges [-1,1],[-2,2],[-3,3] respectively, and the vector X was computed again for each of these ranges. For the sake of comparison between Faugeras and Hall method with such noise, the accuracy from 2D points discrepancy was calculated for each range for both methods and compared. Hence, the implementation depends on one noise matrix for both methods and each time it is adapted to the stated ranges by multiplying it by 0.5,1, and 1.5 respectively. This means that the algorithm were run three times to get the results.

- proj_nn1=proj_n; % proj_n is defined in step 8
- Q_n1=zeros(size(points,2)*2,11);
- B_n1=zeros(size(points,2)*2,1);
- for i=1:size(proj,2)
- Q_n1(2*i-1,:)= [points(1,i),points(2,i), points(3,i),-proj_nn1(1,i)*points(1,i),
- -proj_nn1(1,i)*points(2,i),-proj_nn1(1,i)*points(3,i),0,0,0,1,0];
- Q_n1(2*i,:)=[0,0,0,-proj_nn1(2,i)*points(1,i),-proj_nn1(2,i)*points(2,i),-proj_nn1(2,i)*points(3,i),
- points(1,i),points(2,i), points(3,i),0,1];
- B_n1(2*i-1)=proj_nn1(1,i);
- B_n1(2*i)=proj_nn1(2,i);
- end
- X_n1=Q_n1\B_n1;

- T1=X_n1(1:3);
- T2=X_n1(4:6);
- T3=X_n1(7:9);
- C1=X_n1(10);
- C2=X_n1(11);
- tz=1/norm(T2);
- faug_mat_n1=[T1' C1;T3' C2;T2' 1]; % normalized by tz to compare it with Hall's
- proj_nn1_r=faug_mat_n1*points;
- for i=1:size(proj_nn1_r,2)
- proj_nn1_r(1:2,i)=proj_nn1_r(1:2,i)/proj_nn1_r(3,i); % normalization by s
- end
- dis_norm_nn1=zeros(1,Num_points);
- for i=1:size(proj,2)
- dis_norm_nn1(i)=norm([(proj(1,i)-proj_nn1_r(1,i)),(proj(2,i)-proj_nn1_r(2,i))]);
- end
- ave_nn1=mean(dis_norm_nn1); % mean of distances
- SD_nn1=sqrt(var(dis_norm_nn1)); % standard deviation of distances
- % output (for 1'st range)

- % Faugeras discrepancy

$$ave\_nn1 = 0.2074$$

$$SD\_nn1 = 0.11846$$

- % Hall discrepancy

$$ave\_n2 = 0.2074$$

$$SD\_n2 = 0.11846$$

- % output (for 2'nd range)

- % Faugeras discrepancy

$$ave\_nn1 = 0.35224$$

$$SD\_nn1 = 0.19024$$

- % Hall discrepancy

$$ave\_n2 = 0..35224$$

$$SD\_n2 = 0.19024$$

- % output (for 3'rd range)

- % Faugeras discrepancy

$$ave\_nn1 = 0.51256$$

$$SD\_nn1 = 0.3935$$

- % Hall discrepancy

$$ave\_n2 = 0.51256$$

$$SD\_n2 = 0.3935$$

The presented results disclose that the discrepancy goes up for each method with the increase in the noise range. However, both methods exhibit similar results for the studied ranges.

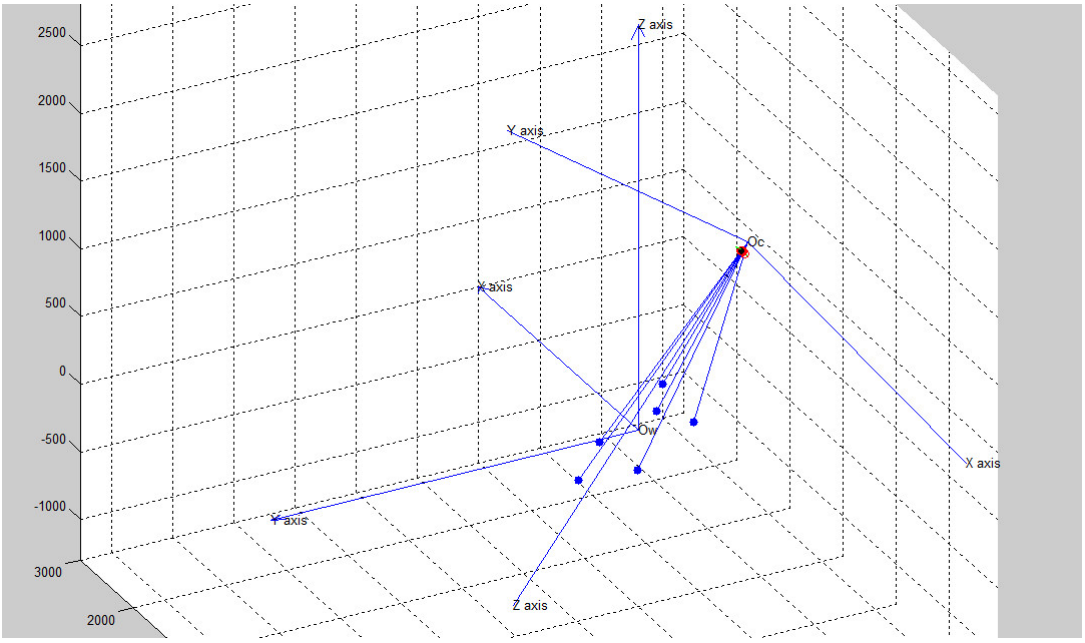
In the last step, part 3, a Matlab figure was opened and the entire simulated environment was drawn. It was assumed while drawing that the coordinate system of the Matlab figure is the same as the world coordinate system and everything was referenced to it before plotting. Thus, the world coordinate system was plotted by just defining the standard unit vectors lines, with an arbitrary length chosen be 3000 to adapt to the environment, and the 3D points were scattered directly on the same figure. The camera coordinate system was drawn next in the same way but after transforming it to the world coordinate system. This transformation was performed by multiplying it by the inverse extrinsic matrix. The focal point is assumed, during the theoretical analysis of camera calibration, to be at the camera coordinate system origin and image plane is far from this origin by the focal length on the positive z axis. Therefore, a point with [0,0,f] coordinates was drawn after transforming it to the world coordinate system. Next, image plane was plotted by first defining its four corners and then transforming them to the world coordinate system. Those corners, in fact, were computed with respect to the principal point (u0,v0) defined in step 1. It is noteworthy that the principal point is measured with respect to computer image buffer. Hence, to be drawn in the current real environment, it should be scaled back by multiplying u0 and vo by $\frac{1}{ku}$ ($\frac{f}{au}$) and $\frac{1}{kv}$ ($\frac{f}{av}$) respectively. Finally, in order to draw the 2D points, there are two possible methods. The first one, which was implemented in this report, is to follow step one and two (affine and perspective transformations) of camera calibration steps studied in the lecture to get the 2D points on the image plane and then transform them back to the current figure/world coordinate system to be plotted correctly. The second way is to go backward by subtracting the principal point from the projected 2D points, scale them back by dividing the x and y coordinates by -ku and -kv, and then transform them to the Matlab/world coordinate system. In addition, those 2D points were connected to their corresponding 3D points by lines. The following code shows the previously described steps of plotting with the output figure.

- W=3000;
- scatter3(points(1,:),points(2,:),points(3,:),'fill');hold on;
  % world x axis
- line([W 0],[0 0],[0 0]);
- line([W W-100],[0 50],[0 0]); % first part of the arrow
- line([W W-100],[0 -50],[0 0]);% second part of the arrow
- text(W,0,0,'X axis');
  % world y axis
- line([0 0],[0 W],[0 0]);
- line([0 50],[W W-100],[0 0]);% first part of the arrow
- line([0 -50],[W W-100],[0 0]);% second part of the arrow
- text(0,W,0,'Y axis');
  % world z axis
- line([0 0],[0 0],[0 W]);
- line([0 0],[0 -50],[W W-100]);% first part of the arrow
- line([0 0],[0 +50],[W W-100]);% second part of the arrow
- text(0,0,W,'Z axis');
- text(0,0,0,'Ow');
  % calculating the inverse transformation matrix
- inv_ext=[R',-R'*T];
- inv_ext=[inv_ext;0 0 0 1]; % inv_ext*ext: should be equal to identity
  % computing the origin and standard unit vectors of the camersa coordinate system
- Oc=inv_ext*[0;0;0;1];

```
- Xc=inv_ext*[W;0;0;1];
- Yc=inv_ext*[0;W;0;1];
- Zc=inv_ext*[0;0;W;1];
  % camera x axis
- line([Oc(1) Xc(1)],[Oc(2) Xc(2)],[Oc(3) Xc(3)]);hold on;
- text(Xc(1),Xc(2),Xc(3),'X axis');
  % camera y axis
- line([Oc(1) Yc(1)],[Oc(2) Yc(2)],[Oc(3) Yc(3)]);
- text(Yc(1),Yc(2),Yc(3),'Y axis');
  % camera z axis
- line([Oc(1) Zc(1)],[Oc(2) Zc(2)],[Oc(3) Zc(3)]);
- text(Zc(1),Zc(2),Zc(3),'Z axis');
- text(Oc(1),Oc(2),Oc(3),'Oc');
  % checking that camera unit vectors are perpendicular
  % dot([Xc(1:3)-Oc(1:3)],[Yc(1:3)-Oc(1:3)])
  %dot([Zc(1:3)-Oc(1:3)],[Yc(1:3)-Oc(1:3)])
  %dot([Zc(1:3)-Oc(1:3)],[Xc(1:3)-Oc(1:3)])

  % focal point
- foc=inv_ext*[0;0;f;1];
- scatter3(foc(1),foc(2),foc(3),'k','fill');
  % image plane
- Ix=u0*f/au;
- Iy=v0*f/av;
- im_cor1=[-Ix,-Iy,f];im_cor1=inv_ext*[im_cor1,1]';
- im_cor2=[Ix,-Iy,f];im_cor2=inv_ext*[im_cor2,1]';
- im_cor3=[Ix,Iy,f];im_cor3=inv_ext*[im_cor3,1]';
- im_cor4=[-Ix,Iy,f];im_cor4=inv_ext*[im_cor4,1]';
- line([im_cor1(1) im_cor2(1)],[im_cor1(2) im_cor2(2)],[im_cor1(3) im_cor2(3)],'Color','g');
- line([im_cor2(1) im_cor3(1)],[im_cor2(2) im_cor3(2)],[im_cor2(3) im_cor3(3)],'Color','g');
- line([im_cor3(1) im_cor4(1)],[im_cor3(2) im_cor4(2)],[im_cor3(3) im_cor4(3)],'Color','g');
- line([im_cor4(1) im_cor1(1)],[im_cor4(2) im_cor1(2)],[im_cor4(3) im_cor1(3)],'Color','g');
  % 2d points
- points_2d=ext*points;
- points_2d(1,:)=f*points_2d(1,:)./points_2d(3,:);
- points_2d(2,:)=f*points_2d(2,:)./points_2d(3,:);
- points_2d(3,:)=f;
- points_2d=inv_ext*points_2d;
- scatter3(points_2d(1,:),points_2d(2,:),points_2d(3,:));
  %lines between the 3d points and their projections on the image plane
- for i=1:size(points,2)
-         line([Oc(1,1) points(1,i)],[Oc(2,1) points(2,i)],[Oc(3,1) points(3,i)]);
- end
  % checking step to make sure that each 3d point with its projection,2d point , and the origin of the camera coordinate system % are collinear. The result of the cross prodcut should be theoritically zero:
- for i=1:size(points,2)
-         cross(points_2d(1:3,i)-Oc(1:3),points(1:3,i)-Oc(1:3));
```

– end



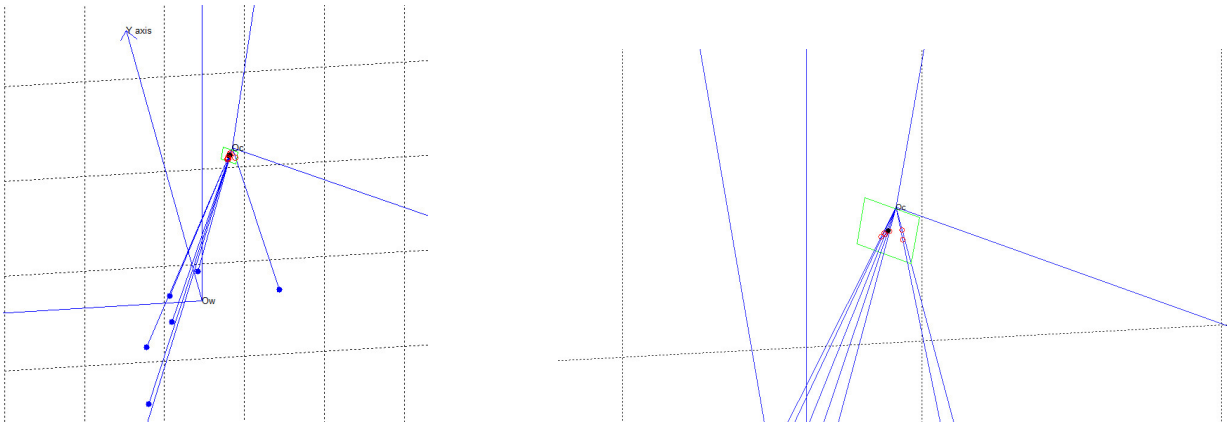


Figure 2: The entire simulated camera environment

As it is shown in the preceding three images, the rays pass through 3D points, 2D points, and the camera coordinate system origin. This was also checked by checking their collinearity, shown in the last step of the Matlab code. The image plane was, in fact, drawn in its real dimension but it cab be magnified by multiplying Ix and Iy in the code by scale factor.

3 Conclusion

In this lab assignment, a simulated camera was calibrated using more than one method. Moreover, the effect of the noise on these methods was studied by finding the accuracy of the 2D points before and after adding the Gaussian noise, and a number of comparisons were drawn.

Indeed, considerable attention should be paid to the different parameters names so that nothing will be overwritten. One interesting thing that could be done in this assignment is to calibrate a real camera and use 3D points from a real scene instead of generating them randomly. Subsequently, the results in this case can be reflected visually on the captured image which could make it easier to compare and understand the different effects.