VISUAL PERCEPTION

Coursework 1: Calibrate a simulated camera
Related to lecture Camera Calibration

FREDERIC GARCIA (u1038431)

Part 1

- **Step 1**. Define the intrinsic parameters and extrinsic parameters with the following values.

```
au=557.0943; av=712.9824; u0=326.3819; v0=298.6679;
f=80 mm.;
Tx=100 mm.; Ty=0 mm.; Tz=1500 mm.;
Phix=0.8*pi/2; Phiy=-1.8*pi/2; Phix1=pi/5;      Euler XYX1
Image size:640x480
```

This initial step consists on declaration and initialization of the variables:

```
% Step 1. Define the instrinsic parameters and extrinsic parameters with the
% following values: (the units are in mm)
au=557.0943; av=712.9824;u0=326.3819;v0=298.6679; % Intrinsic params.
f=80; % Focal distance
Tx=100; Ty=0; Tz=1500; % Trans. vector
Phix=0.8*pi/2; Phiy=-1.8*pi/2; Phix1=pi/5;
```

MATLAB

- **Step 2**. Get the intrinsic and extrinsic transformation matrix.

To obtain the intrinsic transformation matrix is only necessary to order the initial values (au, av, u0, v0) as follows:

```
iMatrix=[au 0 u0 0;0 av v0 0;0 0 1 0]; % Intrinsic matrix
```

MATLAB

```
iMatrix =
  557.0943         0  326.3819         0
         0  712.9824  298.6679         0
         0         0    1.0000         0
```

According to the extrinsic transformation matrix, it is necessary to calculate the rotation matrix and the translation matrix. The former is obtained by using Phix, Phiy and Phiz values which is an Euler XYX1 (rotX, rot Y and rotX1), while the latter is given by Tx, Ty and Tz values. The extrinsic matrix is composed as follows:

```
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)];
R=rotX*rotY*rotX1; % Rotation matrix (Euler XYX1)
t=[Tx Ty Tz]'; % Translation matrix
eMatrix = [R t;0 0 0 1]; % Extrinsic matrix
```

MATLAB

```
eMatrix =
  1.0e+003 *
   -0.0010   -0.0002   -0.0003    0.1000
   -0.0003    0.0008    0.0006         0
    0.0001    0.0006   -0.0008    1.5000
         0         0         0    0.0010
```

- **Step 3**. Define a set of 3D points in the rang [0:1500;0:1500;0:1500]. Note the points should be non-linear and non-coplanar. At least you need to define a set of 6 points.

To define the 3D points, the *rand* Matlab function has been used. It returns a random value between 0 and 1, then it is necessary to multiply this value by 1500 to be between the given rang. Using this function, we are sure that the returned values are non-linear and non-coplanar.

In addition, nPoints is a declared variable which contains the number of 3D points. In this case is 6 (the minimum necessary to solve the 11 equations of the system) but can be modified to solve the next steps.

The Matlab code is:

```
nPoints=6; % At list we need to define 6 points
for i=1:nPoints
    p3D{i}=[1500*rand(1) 1500*rand(1) 1500*rand(1)];
end
```

- **Step 4**. Compute the projection on the image plane by using the camera transformation matrix.

To compute the projection on the image plane it is necessary to multiply the 3D point (point in the world coordinate system) by the transformation matrix (intrinsic transformation matrix multiply extrinsic transformation matrix) and then normalize the result by the third component (it has to be 1, because in the image plane only are used the 'x' and 'y' values, 2D.). This will return floating values (subpixel accuracy), however, using the *round* Matlab function, the result will be in pixel accuracy.

```
for i=1:nPoints
    p2D{i}=iMatrix*eMatrix*[p3D{i} 1]';
    % With subpixel accuracy
    p2D{i}(1)=p2D{i}(1)/p2D{i}(3); % Normalization
    p2D{i}(2)=p2D{i}(2)/p2D{i}(3);
    % Without subpixel accuracy
%     p2D{i}(1)=round(p2D{i}(1)/p2D{i}(3)); % Normalization
%     p2D{i}(2)=round(p2D{i}(2)/p2D{i}(3));
    p2D{i}(3)=1;
end
```

I.e.

```
K>> p3D{1}
ans =
  1.0e+003 *
    1.0642    0.6433    0.4569
K>> p2D{1}
ans =
  -66.4063
  492.7967
    1.0000
```

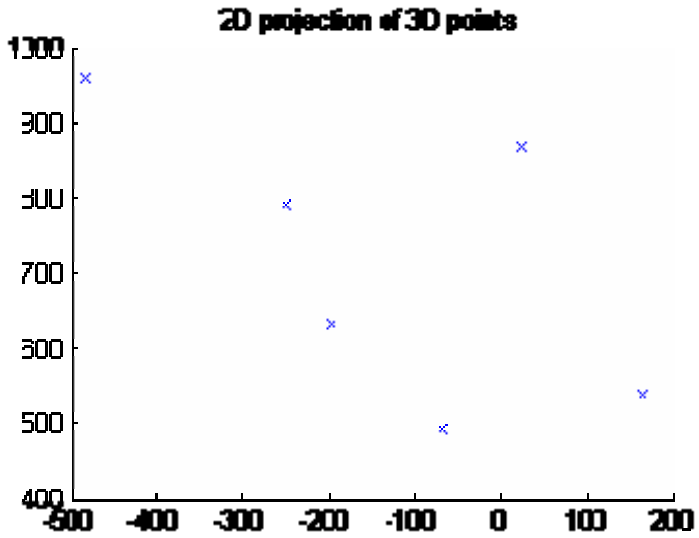
- **Step 5**. Open a window in Matlab which will be used as the image plane and draw the 2D points. Are the points well spread in the image plane? Will the distribution of points in the image affect the accuracy in the computation?

  The Matlab code to draw the points is:

```
figure;
hold on;
for i=1:nPoints
    plot(p2D{i}(1),p2D{i}(2),'bx');
end
title('2D projection of 3D points');
```

MATLAB

And the result for the 3D points given is:



The points are well spread in the image plane as can be see in the previous figure because they are randomly computed. According to the distribution of points in the image and how they can affect the accuracy in the computation, if the points are not correctly distributed or they are in the same plane, the result of the computation can be neglected because it is non useful.

- **Step 6**. By using the points of Step 3 and their projection obtained in Step 5, compute the 3x4 transformation matrix by using the method of Hall.

The 3x4 transformation matrix to compute ('A') is presented in the theory as:

$$\begin{pmatrix} s\,{}^{I}X_u \\ s\,{}^{I}Y_u \\ s \end{pmatrix} = \underbrace{\begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & 1 \end{pmatrix}}_{A} \begin{pmatrix} {}^{W}X_w \\ {}^{W}Y_w \\ {}^{W}Z_w \\ 1 \end{pmatrix}$$

Where A matrix is obtained by the formula:

$$A = \left(Q^t Q\right)^{-1} Q^t B$$

And,

$$Q_{2i-1} = \begin{pmatrix} {}^{W}X_{wi} & {}^{W}Y_{wi} & {}^{W}Z_{wi} & 1 & 0 & 0 & 0 & 0 & -{}^{I}X_{ui}\,{}^{W}X_{wi} & -{}^{I}X_{ui}\,{}^{W}Y_{wi} & -{}^{I}X_{ui}\,{}^{W}Z_{wi} \end{pmatrix}$$

$$Q_{2i} = \begin{pmatrix} 0 & 0 & 0 & 0 & {}^{W}X_{wi} & {}^{W}Y_{wi} & {}^{W}Z_{wi} & 1 & -{}^{I}Y_{ui}\,{}^{W}X_{wi} & -{}^{I}Y_{ui}\,{}^{W}Y_{wi} & -{}^{I}Y_{ui}\,{}^{W}Z_{wi} \end{pmatrix}$$

$$B_{2i-1} = \left({}^{I}X_{ui}\right)$$

$$B_{2i} = \left({}^{I}Y_{ui}\right)$$

Then, the implementation consists on taking the 3D points coordinates and built the equations to be solved:

```
Q=[];B=[];
for i=1:nPoints
    Q=[Q;p3D{i}(1) p3D{i}(2) p3D{i}(3) 1 0 0 0 0 -p2D{i}(1)*p3D{i}(1) -p2D{i}(1)*p3D{i}(2)
-p2D{i}(1)*p3D{i}(3)];
    Q=[Q;0 0 0 0 p3D{i}(1) p3D{i}(2) p3D{i}(3) 1 -p2D{i}(2)*p3D{i}(1) -p2D{i}(2)*p3D{i}(2)
-p2D{i}(2)*p3D{i}(3)];
    B=[B;p2D{i}(1)];
    B=[B;p2D{i}(2)];
end
A=inv(Q'*Q)*Q'*B; % Pseudo inverse due to Q is not a regular matrix
A=[A(1) A(2) A(3) A(4);
   A(5) A(6) A(7) A(8);
   A(9) A(10) A(11) 1];
```

MATLAB

The matrix result in this example is:

```
A =
   -0.3324    0.0624   -0.2662  363.5215
   -0.1207    0.4903    0.1028  298.6679
    0.0001    0.0004   -0.0005    1.0000
```

- **Step 7**. Compare the matrix obtained in Step 6 to the one defined in Step 2.

The matrix obtained in the Step 2 is A1, it is obtained by multiplying the Intrinsic transformation matrix by the Extrinsic transformation matrix, while A is the matrix obtained in the previous step using the method of Hall.

The Matlab code is:

```
A1=iMatrix*eMatrix; % Original one
A1=A1./A1(3,4);
display('Matrix obtained in Step 6:');
A
display('Matrix obtained in Step 2:');
A1
```

And as can be observed, the resulting matrixes are the same:

```
Matrix obtained in Step 6:
A =
   -0.3324    0.0624   -0.2662  363.5215
   -0.1207    0.4903    0.1028  298.6679
    0.0001    0.0004   -0.0005    1.0000
Matrix obtained in Step 2:
A1 =
   -0.3324    0.0624   -0.2662  363.5215
   -0.1207    0.4903    0.1028  298.6679
    0.0001    0.0004   -0.0005    1.0000
```

- **Step 8**. Add some Gaussian noise to the 2D points producing discrepancies between the range [-1,+1] pixels for the 95% of points. Again repeat step 6 with the noisy 2D points and the ones defined in step 3. Compare the matrix obtained.

In this step, it is calculated which points must be distorted (95%) and a random noise is added. Then Step 6 and Step 7 are recomputed using the distorted points.

The Matlab code is:

```
nP2AddNoise=round(95/100*nPoints); % Number of 2D points to add noise (95%)
% Add Gaussian noise btw [-1,+1]
p2DwNoise=p2D;
for i=1:nP2AddNoise
    p2DwNoise{i}(1)=p2D{i}(1)+round(rand(1)*2-1);
    p2DwNoise{i}(2)=p2D{i}(2)+round(rand(1)*2-1);
end
Q_Noise=[];B_Noise=[];
for i=1:nPoints
    Q_Noise=[Q_Noise;p3D{i}(1) p3D{i}(2) p3D{i}(3) 1 0 0 0 0 -p2DwNoise{i}(1)*p3D{i}(1) -
p2DwNoise{i}(1)*p3D{i}(2) -p2DwNoise{i}(1)*p3D{i}(3)];
    Q_Noise=[Q_Noise;0 0 0 0 p3D{i}(1) p3D{i}(2) p3D{i}(3) 1 -p2DwNoise{i}(2)*p3D{i}(1) -
p2DwNoise{i}(2)*p3D{i}(2) -p2DwNoise{i}(2)*p3D{i}(3)];
    B_Noise=[B_Noise;p2DwNoise{i}(1)];
    B_Noise=[B_Noise;p2DwNoise{i}(2)];
end
A_Noise=inv(Q_Noise'*Q_Noise)*Q_Noise'*B_Noise; % Pseudo inverse due to Q_Noise is not a
regular matrix
A_Noise=[A_Noise(1) A_Noise(2) A_Noise(3) A_Noise(4);
         A_Noise(5) A_Noise(6) A_Noise(7) A_Noise(8);
         A_Noise(9) A_Noise(10) A_Noise(11) 1];
```

```
% Comparing the matrices...
display('Matrix obtained in Step 8 with some Gaussian noise:');
A_Noise
display('Matrix obtained in Step 2:');
A1
```

MATLAB

And as can be observed, in this case appears some discrepancies in the floating part of the matrix values:

```
Matrix obtained in Step 8 with some Gaussian noise:
A_Noise =
   -0.3307    0.0606   -0.2670  364.2339
   -0.1188    0.4898    0.1008  298.4382
    0.0001    0.0004   -0.0005    1.0000
Matrix obtained in Step 2:
A1 =
   -0.3324    0.0624   -0.2662  363.5215
   -0.1207    0.4903    0.1028  298.6679
    0.0001    0.0004   -0.0005    1.0000
```

- **Step 9**. Increase the number of 3D points and their 2D projections and repeat Step 8. More the points we use more accurate is the matrix obtained.

To implement this step, it is necessary to modify the 'nPoints' value and recompute all the follow steps:

```
nPoints=50; % Step9: increase nPoints
…
```

MATLAB

As is said in the sheet, the more points we use, the more accurate is the matrix obtained:

nPoints = 100:

```
Matrix obtained in Step 8 with some Gaussian noise:
A_Noise =
   -0.3322    0.0624   -0.2659  363.1128
   -0.1211    0.4907    0.1029  298.5060
    0.0001    0.0004   -0.0005    1.0000
Matrix obtained in Step 2:
A1 =
   -0.3324    0.0624   -0.2662  363.5215
   -0.1207    0.4903    0.1028  298.6679
    0.0001    0.0004   -0.0005    1.0000
```

nPoints = 1000:

```
Matrix obtained in Step 8 with some Gaussian noise:
A_Noise =
   -0.3324    0.0623   -0.2662  363.4914
   -0.1207    0.4904    0.1027  298.7366
    0.0001    0.0004   -0.0005    1.0000
Matrix obtained in Step 2:
A1 =
   -0.3324    0.0624   -0.2662  363.5215
   -0.1207    0.4903    0.1028  298.6679
    0.0001    0.0004   -0.0005    1.0000
```

Part 2

- **Step 10**. Define the vector X of the method of Faugeras. Compute X by using both least-squares (LS) and Singular Value Decomposition (SVD) by using the points of Step 3 and Step 4, without noise. Extract the camera parameters from both computations. Compare the obtained parameters with the ones defined in Step 1.

  Using the LS method, the Matlab code to obtain X is:

```
Q_Faug=[];B_Faug=[];
for i=1:nPoints
    Q_Faug=[Q_Faug;p3D{i}(1) p3D{i}(2) p3D{i}(3) -p2D{i}(1)*p3D{i}(1) -p2D{i}(1)*p3D{i}(2)
-p2D{i}(1)*p3D{i}(3) 0 0 0 1 0];
    Q_Faug=[Q_Faug;0 0 0 -p2D{i}(2)*p3D{i}(1) -p2D{i}(2)*p3D{i}(2) -p2D{i}(2)*p3D{i}(3)
p3D{i}(1) p3D{i}(2) p3D{i}(3) 0 1];
    B_Faug=[B_Faug;p2D{i}(1)];
    B_Faug=[B_Faug;p2D{i}(2)];
end
% X = [T1 T2 T3 C1 C2]'
X=inv(Q_Faug'*Q_Faug)*Q_Faug'*B_Faug; % Pseudo inverse due to Q_Faug is not a regular
matrix
```

  Using the SVD method, the Matlab code to obtain X is:

```
Q_Faug=[];B_Faug=[];
for i=1:nPoints
    Q_Faug=[Q_Faug;p3D{i}(1) p3D{i}(2) p3D{i}(3) -p2D{i}(1)*p3D{i}(1) -p2D{i}(1)*p3D{i}(2)
-p2D{i}(1)*p3D{i}(3) 0 0 0 1 0];
    Q_Faug=[Q_Faug;0 0 0 -p2D{i}(2)*p3D{i}(1) -p2D{i}(2)*p3D{i}(2) -p2D{i}(2)*p3D{i}(3)
p3D{i}(1) p3D{i}(2) p3D{i}(3) 0 1];
    B_Faug=[B_Faug;p2D{i}(1)];
    B_Faug=[B_Faug;p2D{i}(2)];
end
% Applying SVD:
[U,S,V]=svd(Q_Faug);
X=(V*[inv(S(1:size(S,1)-1,:)) zeros(11,1)]*U')*B_Faug;
```

And to extract the camera parameters, the Matlab code is:

```
T1=X(1:3); T2=X(4:6); T3=X(7:9); C1=X(10); C2=X(11);
T1=T1';T2=T2';T3=T3';
u0Faug=(T1*T2')/norm(T2)^2;
v0Faug=(T2*T3')/norm(T2)^2;
auFaug=(norm(cross(T1',T2')))/norm(T2)^2;
avFaug=(norm(cross(T2',T3')))/norm(T2)^2;
TxFaug=(norm(T2)/norm(cross(T1',T2')))*(C1-((T1*T2')/norm(T2)^2));
TyFaug=(norm(T2)/norm(cross(T2',T3')))*(C2-((T2*T3')/norm(T2)^2));
TzFaug=1/norm(T2);
r1Faug=(norm(T2)/norm(cross(T1',T2')))*(T1-((T1*T2')/norm(T2)^2)*T2);
r2Faug=(norm(T2)/norm(cross(T2',T3')))*(T3-((T2*T3')/norm(T2)^2)*T2);
r3Faug=T2/norm(T2);
iMatrixFaug=[auFaug 0 u0Faug 0;0 avFaug v0Faug 0;0 0 1 0]; % Intrinsec matrix (Faugeras-
Toscani)
eMatrixFaug=[r1Faug(1) r1Faug(2) r1Faug(3) TxFaug;
             r2Faug(1) r2Faug(2) r2Faug(3) TyFaug;
             r3Faug(1) r3Faug(2) r3Faug(3) TzFaug;
             0         0         0         1    ]; % Extrinsec matrix (Faugeras-Toscani)
AFaug=iMatrixFaug*eMatrixFaug;
AFaug=AFaug./AFaug(3,4);
```

MATLAB

This consists on the implementation of the formulas presented in the theory.

I.e.

- Using LS method:

```
Matrix obtained in Step 2:
A1 =
   -0.3324    0.0624   -0.2662  363.5215
   -0.1207    0.4903    0.1028  298.6679
    0.0001    0.0004   -0.0005    1.0000
Matrix obtained in Step 10 (LS):
AFaug =
   -0.3324    0.0624   -0.2662  363.5215
   -0.1207    0.4903    0.1028  298.6679
    0.0001    0.0004   -0.0005    1.0000
```

- Using SVD method:

```
Matrix obtained in Step 2:
A1 =
   -0.3324    0.0624   -0.2662  363.5215
   -0.1207    0.4903    0.1028  298.6679
    0.0001    0.0004   -0.0005    1.0000
Matrix obtained in Step 10 (SVD):
AFaug =
   -0.3324    0.0624   -0.2662  363.5215
   -0.1207    0.4903    0.1028  298.6679
    0.0001    0.0004   -0.0005    1.0000
```

As can be observed, the resulting matrixes are the same.

Note: To compute an inverse matrix quickly it is usually to use the SVD method as is indicate in the next formula:

$$A^{-1} = V \cdot S^{-1} \cdot U^{T}$$

However, in this specific case, the size of Q_Faug is 12x11. Then, after apply the SVD, the matrices U, S, V obtained are 12x12, 12x11, 11x11 respectively. S is the diagonal matrix and it is necessary to apply the inverse of it according with the previous formula. Due to its size, it is necessary to remove the last row which all values are zeros and after compute the inverse, it is necessary to add a new column which will allow the multiplication with the transpose of U.

The Matlab code is: `[inv(S(1:size(S,1)-1,:)) zeros(11,1)]`

- **Step 11**. Increase the noise in the 2D points (as done in Step 8) and compute again the vector X (repeat Step 10).

  This step will use the same 2D points which were generated in the Step 8.

- Which are the parameters more influenced by noise?

  A1 is the original transformation matrix, while AFaug is the transformation matrix obtained using six 3D points and its projections without noise added and with LS method. AFaugNoise is the same matrix as AFaug but using the 2D projection points with some added noise. The parameters which are more influenced are the parameters which correspond on the translation vector as can be seen in the next example:

```
Matrix obtained in Step 2:
A1 =
   -0.3324    0.0624   -0.2662  363.5215
   -0.1207    0.4903    0.1028  298.6679
    0.0001    0.0004   -0.0005    1.0000
Matrix obtained in Step 10 (LS):
AFaug =
   -0.3324    0.0624   -0.2662  363.5215
   -0.1207    0.4903    0.1028  298.6679
    0.0001    0.0004   -0.0005    1.0000
Matrix obtained in Step 11 (LS):
AFaugNoise =
   -0.3280    0.0601   -0.2654  359.8737
   -0.1250    0.4940    0.1060  297.5768
    0.0001    0.0004   -0.0005    1.0000
```

- Which method of computation is more accurate (LS or SVD)?

The previous example was obtained using the LS method, using the SVD method, the result will be:

```
Matrix obtained in Step 2:
A1 =
   -0.3324    0.0624   -0.2662  363.5215
   -0.1207    0.4903    0.1028  298.6679
    0.0001    0.0004   -0.0005    1.0000
Matrix obtained in Step 10 (SVD):
AFaug =
   -0.3324    0.0624   -0.2662  363.5215
   -0.1207    0.4903    0.1028  298.6679
    0.0001    0.0004   -0.0005    1.0000
Matrix obtained in Step 11 (SVD):
AFaugNoise =
   -0.3467    0.0629   -0.2747  379.5143
   -0.1218    0.5229    0.0940  308.1103
    0.0001    0.0004   -0.0006    1.0000
```

It is appreciable that LS method is more accurate.

- Which method is more accurate (Faugeras or Hall)?

The original transformation matrix obtained in the Step 2 is:

```
Matrix obtained in Step 2:
A1 =
   -0.3324    0.0624   -0.2662  363.5215
   -0.1207    0.4903    0.1028  298.6679
    0.0001    0.0004   -0.0005    1.0000
```

The transformation matrix obtained using the method of Hall is:

```
Matrix obtained in Step 8 with some Gaussian noise:
A_Noise =
   -0.3297    0.0591   -0.2640  362.5732
   -0.1438    0.5079    0.1076  299.1146
    0.0000    0.0004   -0.0005    1.0000
```

And the transformation matrix obtained using the method of Faugeras is:

```
Matrix obtained in Step 11 (LS):
AFaugNoise =
   -0.3280    0.0601   -0.2654  359.8737
   -0.1250    0.4940    0.1060  297.5768
    0.0001    0.0004   -0.0005    1.0000
```

After compute this example, can be said that the method of Faugeras is more accurate in the rotation values while the method of Hall is more accurate in the translations values.

Part 3

- **Step 12**. Open another window in Matlab and draw the world coordinate system, the camera coordinate system, the focal point, the image plane, and both 3D points and their corresponding projections in the image plane by using the parameters of Step 2 and points without noise.