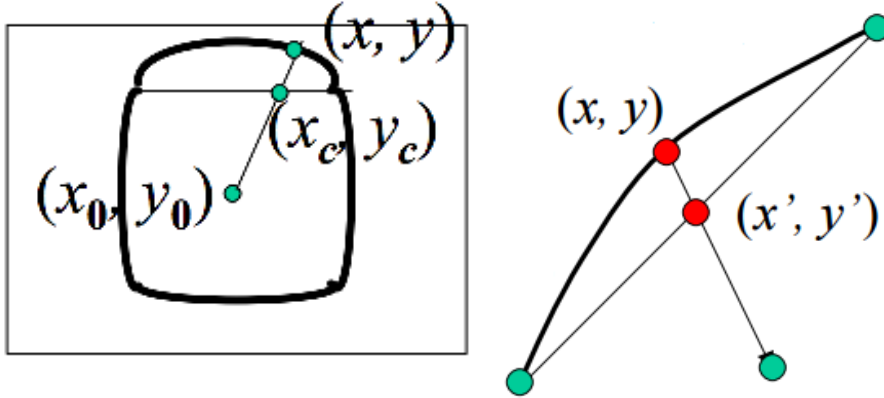


Step 1 – Remove distortion

The checkerboard calibration pattern is used to calibrate lens distortion and pattern shape is shown below with size 10x8 corners in both height and width. The method I used for lens distortion is to straighten the lines which are distorted but supposed to be straight lines.



First, I can choose the image center as the distortion center (x_c, y_c) , then for each pixel (x, y) that is not on the straight line, but actual it should be on the straight line. The radial distortion formula I used 4th order even polynomial, and higher order is omitted because r is getting very large and the picture I took does not show much distortion, so I used is:

$$L(r) = 1 + k_1 r + k_2 r^2 + k_3 r^3$$

I can first calculate the radius parameter and estimate the new position with initial estimation of k_1 and k_2

$$\begin{aligned} r^2 &= (x - x_0)^2 + (y - y_0)^2 \\ x_c &= x_0 + L(r)(x - x_0) = x + k_1 r^2 + k_2 r^4 \\ y_c &= y_0 + L(r)(y - y_0) = y + k_1 r^2 + k_2 r^4 \end{aligned}$$

Then I calculate the radial projection of point (x, y) on the horizontal line, here, I assume the start and end point of the known line is (x_s, y_s) , (x_e, y_e) , then the line equation I is easily calculated and the intersection point between radial line and the known line can be calculated as (x', y')

$$\begin{aligned} l &= (x_s, y_s, 1) \times (x_e, y_e, 1) \\ (x', y', z') &= l \times ((x, y, 1) \times (x_0, y_0, 1)) \\ x' &= x'/z' \\ y' &= y'/z' \end{aligned}$$

Now, I can minimize the error between estimated point (x_c, y_c) and projected point (x', y') for all the corner points in the image, which can be written in mathematically as

$$error = \sum_{i=1}^n (x_c - x')^2 + (y_c - y')^2$$

For the image I took with only one plane, I used the corner function provided in Matlab to obtain the coordinate of all corners and then reorder to the corners in the order of layer by layer, which can be seen below. Then I calculate the line equations with start point and end point for edge line. In this way, I am able to obtain the line equation for total 4 lines, 2 out of which is for horizontal lines and 2 for vertical lines.

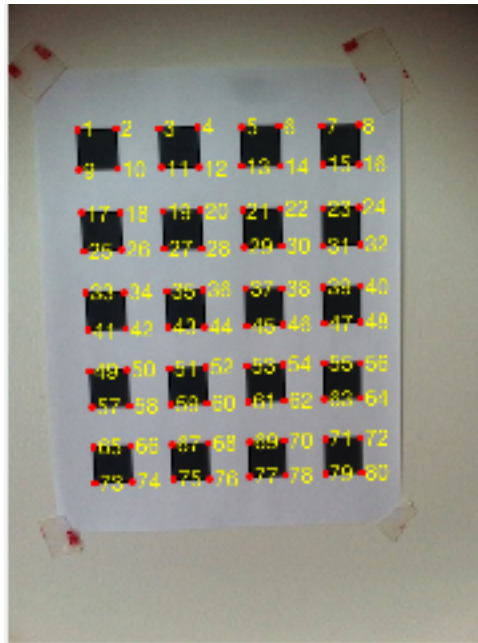


Figure 1 Lens distortion corner label

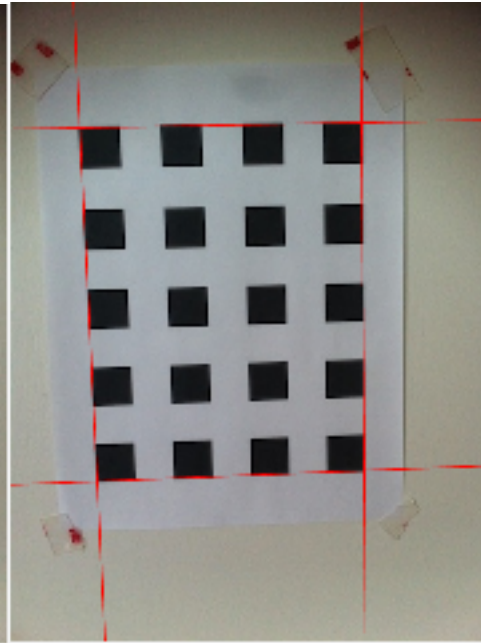
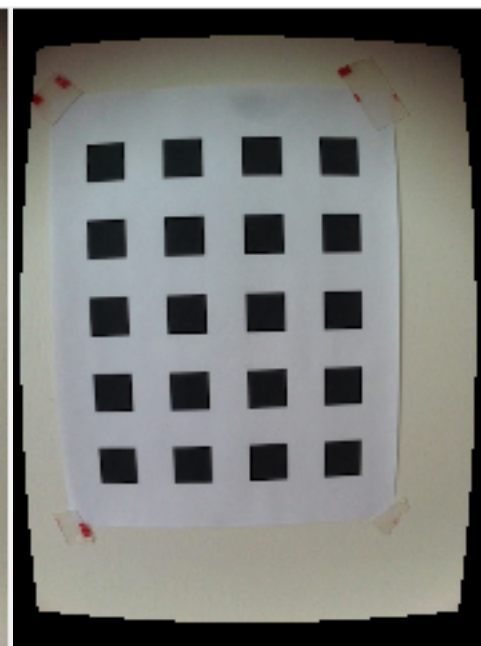
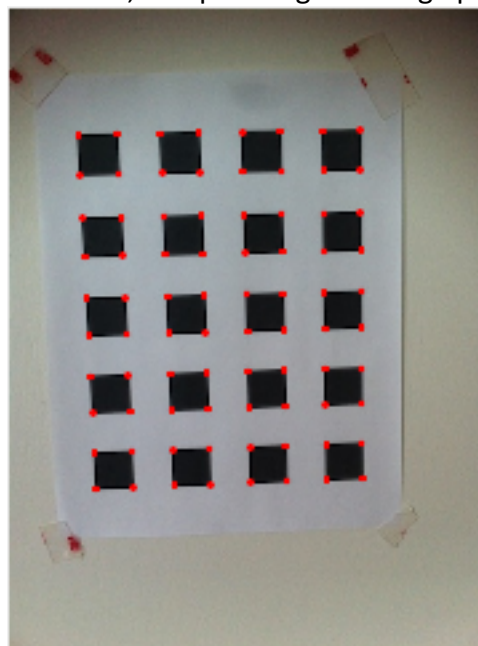


Figure 2 Lens distortion line

With this work done, I loop through the edge points to minimize the error for these 28 points.



After that, I used the ***undistortimage*** function provided in Matlab to correct the image points and it is shown above.

k1 = 3.7630e-05, k2 = -4.2000e-07, k3 = 5.4569e-10

Discussion: This lens distortion method is straightforward and we need to specify lines to be rectified and find all points on that lines, then we can try to minimize the distance between the distance between points. I chose polynomial order 3 for the distortion equation because the distortion is not too much for my picture.

Step2: Calibrate Camera with Gold Standard Algorithm

1. Remove distortion for new image

With the distortion coefficient obtained from question 1, I can transform all the points in the distorted image to the ideal image with straight line being straight. So for all points (x,y) in the image, I do the below steps:

$$\begin{aligned} r^2 &= (x - x_0)^2 + (y - y_0)^2 \\ x' &= x_0 + L(r)(x - x_0) = x + k_1 r^2 + k_2 r^4 \\ y' &= y_0 + L(r)(y - y_0) = y + k_1 r^2 + k_2 r^4 \end{aligned}$$

For the calibration step, the pattern I used is checkboard erected in two orthogonal plane as shown below, and I define the coordinate system on the plane as X-Y plane on one wall, as X going right and Y going down, and then Y-Z plane on the other wall, with Z pointing out of the paper. If I select the left corner point as origin starting point with coordinate (0,0,0), then the rest of the coordinate can be calculated accordingly. Now we can build the mapping between world points and image points through the homograph we learned in class.

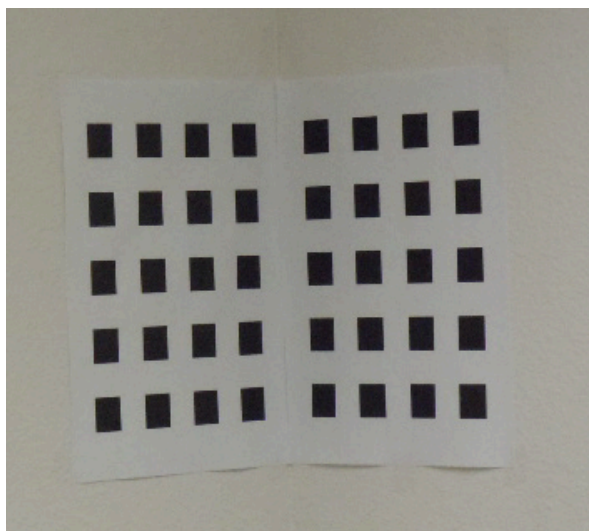
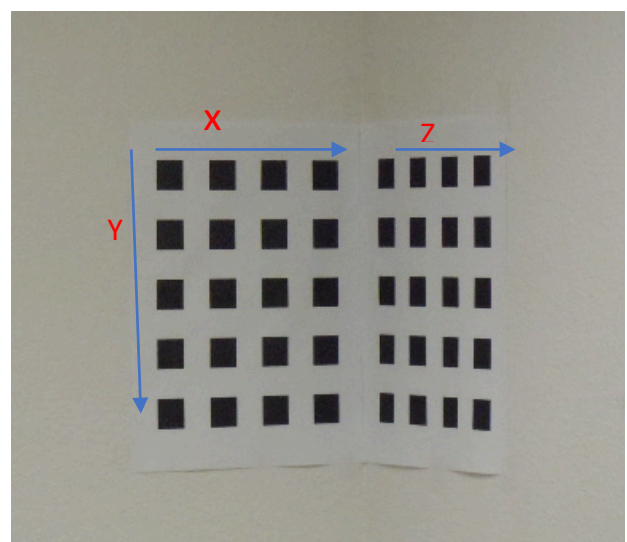
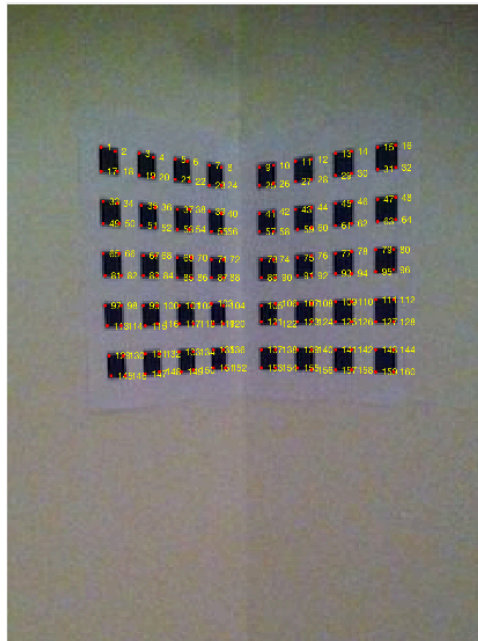


Figure 3 Pattern for Image Calibration



In the above calibration pattern, I can order all the coordinate in a way below and use MATLAB corner function to retrieve all the corners and coordinates. In this case, I have 160 points.



2. Normalization on 2-d data:

$$T_1 = \begin{bmatrix} s & 0 & t_x \\ 0 & s & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

The transformation matrix consists of translation and scaling and takes point X_i to a new set of points X'_i such that the centroid of the new points has the coordinate $(0, 0)^T$, and their average distance from the origin is $\sqrt{2}$. Suppose $X_i = (x_i, y_i, 1)^T$, we have

$$\hat{X}_i = T_1 \vec{X} = \begin{bmatrix} sx_i + t_x \\ sy_i + t_y \\ 1 \end{bmatrix} = \begin{bmatrix} \hat{x} \\ \hat{y} \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} s\bar{x}_i + t_x \\ s\bar{y}_i + t_y \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\begin{aligned}
& \frac{1}{n} \sum_i^m \sqrt{\hat{x}_i^2 + \hat{y}_i^2} \\
&= \frac{1}{n} \sum_i^m \sqrt{(sx_i - s\bar{x})^2 + (sy_i - s\bar{y})^2} \\
&= \frac{s}{n} \sum_i^m \sqrt{(x_i - \bar{x})^2 + (y_i - \bar{y})^2} \\
\\
& S = \frac{\sqrt{2}}{\frac{1}{m} \sum_i^m \sqrt{(x_i - \bar{x})^2 + (y_i - \bar{y})^2}}
\end{aligned}$$

3. Normalization on 3D data

$$\begin{aligned}
\hat{X}_i = T_1 X &= \begin{bmatrix} sX_i + t_x \\ sY_i + t_y \\ sZ_i + t_z \\ 1 \end{bmatrix} = \begin{bmatrix} \hat{X} \\ \hat{Y} \\ \hat{Z} \\ 1 \end{bmatrix} \\
&\begin{bmatrix} sX_i + t_x \\ sY_i + t_y \\ sZ_i + t_z \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \\
&\frac{1}{n} \sum_1^m \sqrt{\hat{x}_i^2 + \hat{y}_i^2 + \hat{z}_i^2} \\
&= \frac{1}{n} \sum_1^m \sqrt{(sX_i - s\bar{X})^2 + (sY_i - s\bar{Y})^2 + (sZ_i - s\bar{Z})^2} \\
&\quad \sqrt{3} \\
&= \frac{1}{m} \sum_1^m \sqrt{(X - \bar{X})^2 + (Y - \bar{Y})^2 + (Z - \bar{Z})^2}
\end{aligned}$$

The basic equation used in the finding homograph between the normalized world coordinates and the normalized image coordinates is:

$$x = PX$$

in which x is the homogenous 3-dimension coordinate as (x, y,1) and X is the homogenous world coordinate as X = (x_w, y_w, Z_w, 1)

$$\begin{bmatrix} \mathbf{0}^\top & -w_i \mathbf{X}_i^\top & y_i \mathbf{X}_i^\top \\ w_i \mathbf{X}_i^\top & \mathbf{0}^\top & -x_i \mathbf{X}_i^\top \\ -y_i \mathbf{X}_i^\top & x_i \mathbf{X}_i^\top & \mathbf{0}^\top \end{bmatrix} \begin{pmatrix} \mathbf{P}^1 \\ \mathbf{P}^2 \\ \mathbf{P}^3 \end{pmatrix} = \mathbf{0}$$

$$\begin{bmatrix}
 0 & 0 & 0 & 0 & -X_i & -Y_i & -Z_i & -1 & y_i X_i & y_i Y_i & y_i Z_i & y_i \\
 X_i & Y_i & Z_i & 1 & 0 & 0 & 0 & 0 & -x_i X_i & -x_i Y_i & -x_i Z_i & -x_i \\
 \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\
 \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\
 0 & 0 & 0 & 0 & -X_n & -Y_n & -Z_n & -1 & y_n X_n & y_n Y_n & y_n Z_n & y_n \\
 X_i & Y_i & Z_i & 1 & 0 & 0 & 0 & 0 & -x_n X_n & -x_n Y_n & -x_n Z_n & -x_n
 \end{bmatrix}
 \begin{bmatrix}
 p_1 \\
 p_2 \\
 p_3 \\
 p_4 \\
 p_5 \\
 p_6 \\
 p_7 \\
 p_8 \\
 p_9 \\
 p_{10} \\
 p_{11} \\
 p_{12}
 \end{bmatrix}
 = 0$$

$Ap = 0$

To solve this over-determined equation subject to $|p| = 1$, we can use SVD decomposition provided in MATLAB to find p as the result DLT result, here the result I got is:

$p =$
 $[-0.3634 \quad -0.0085 \quad -0.4182 \quad -0.0021 \quad -0.0162 \quad -0.5413 \quad 0.0274 \quad -0.0068 \quad -0.0269 \quad -0.0063 \quad 0.0245 \quad -0.6305]$

This is the just the result obtained with DLT, now we have to further optimize it with Levenberg–Marquardt algorithm with the cost function to calculate re-projection error:

$$\sum_{i=1}^n d(x_i, PX_i)^2$$

```

opt = optimoptions('lsqcurvefit','Algorithm','levenberg-marquardt');
[q,res] = lsqcurvefit(@fun,p,xWn,coors,[],[],opt); % Refined parameter

```

$p =$
 $[-0.3633 \quad -0.0085 \quad -0.4179 \quad -0.0019 \quad -0.0163 \quad -0.5414 \quad 0.0274 \quad -0.0068 \quad -0.0271 \quad -0.0063 \quad 0.0248 \quad -0.6307]$

To make it in a matrix form

```

K>> P
P =
-0.3633    -0.0085    -0.4179    -0.0019
-0.0163    -0.5414     0.0274    -0.0068
-0.0271    -0.0063     0.0248    -0.6307

```

De-normalization has to be applied on the result of P to obtain on the real world

$$P = (T_2)^{-1} P T_1$$

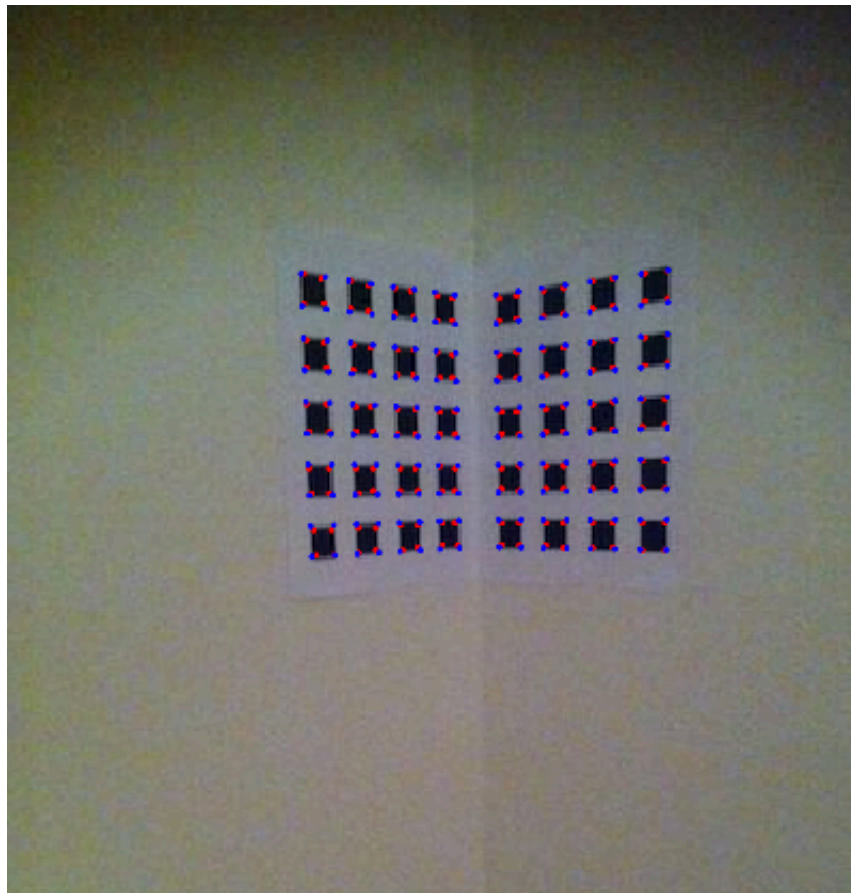
Then I obtained the below result for P

P =
 $1.0e+05 \times$
 $\begin{bmatrix} -0.0051 & -0.0004 & -0.0029 & -1.2809 \\ -0.0014 & -0.0057 & 0.0014 & -1.1218 \\ -0.0000 & -0.0000 & 0.0000 & -0.0101 \end{bmatrix}$

We can use RQ decomposition to find the result of KR, $KR = P(1:3,1:3)$

K =
 $\begin{bmatrix} 558.0086 & 1.0835 & 187.4397 \\ 0 & 532.7424 & 291.7412 \\ 0 & 0 & 1.0000 \end{bmatrix}$

The final selected point and re-projected points is marked below:



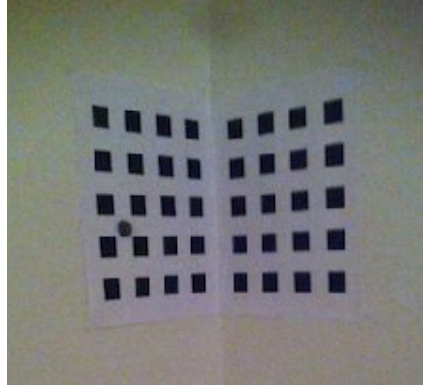
Discussion:

I first used all the corner points and the real world corresponding points, normalize them and used DLT to find the calculate initial p value, then I used Levenberg–Marquardt algorithm to

further optimize it. There are not much difference between the initial value and final value, I guess the initial calculation is already very close to the final value.

Step3: Verification

In this step, I am going to make the verification with a coin and fix the coin at some known place in the world coordinate system, then shift the coin location, take picture, apply P on the coin location to calculate its position in the image world coordinate system.



Then I can apply the homograph transformation on each picture with P obtained from step 4:

$$x = PX$$

More specifically, the x coordinate is obtained from the *getinput* command from MATLAB to pick the coordinate at the center of the coin. Then the coordinate in the world coordinate system is calculated with relative coordinate to the fixed (0,0,0) as used in the step 3.

In this way, I selected 11 locations with coordinates

```
xy = [129.1971 184.4416 1; 147.4161 183.0401 1; 239.9124 181.6387 1;
      129.1971 162.0182 1; 225.8978 143.7993 1; 151.6204 159.2153 1;
      129.1971 209.6679 1; 144.6131 206.8650 1; 197.8686 162.0182 1;
      136.2044 131.1861 1; 147.4161 136.7920 1];
```

And its corresponding world locations are in the images.

```
XYZ = [36 132 0 1; 84 132 0 1; 192 132 156 1; 36 84 0 1;
       192 36 84 1; 84 84 0 1; 36 180 0 1; 84 180 0 1;
       192 84 60 1; 36 36 0 1; 84 36 0 1];
```

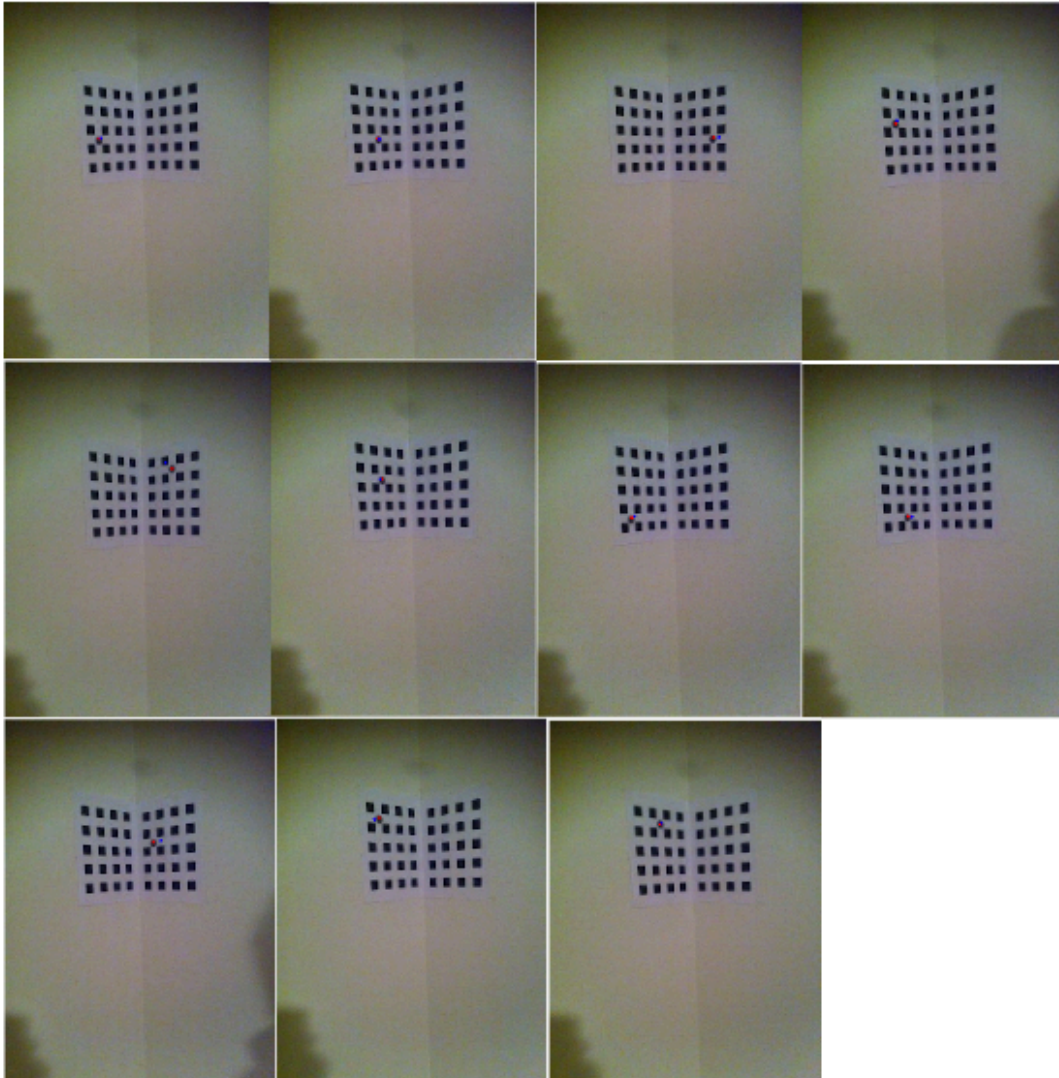
After I obtain the new location according to $x = PX$, I can calculate the mean error in x, y

```
error = xy0-xy_p
```



```
avg = mean(error) = [-4.3415    1.4211];  
variance = variance(error) = [30.9581    7.7958];
```

Also, the variance can be calculated above. Below is the images and location of coins for reference.



Point:

129.1971 184.4416
147.4161 183.0401
239.9124 181.6387
129.1971 162.0182
225.8978 143.7993
151.6204 159.2153
129.1971 209.6679
144.6131 206.8650
197.8686 162.0182
136.2044 131.1861
147.4161 136.7920

Estimated Point

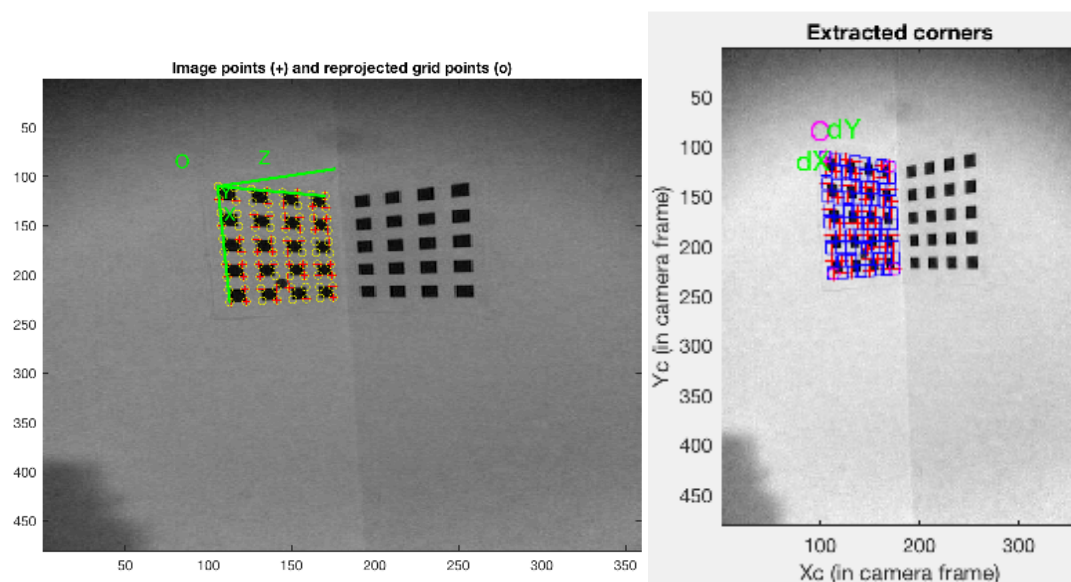
131.6163 183.2553
149.4836 183.5719
247.0673 181.4960
130.7903 158.4305
217.0525 135.3900
148.8169 159.5475
132.4299 207.7044
150.1404 207.2441
207.8084 160.0265
129.9514 133.2213
148.1403 135.1632

1.0000

Discussion: As it can be seen from the superimposed pictures, the location for most coins can be predicted well, but there are also some deviations, and the average error in x direction is - 4.3415 , means there is a shift in x direction, and y 1.4211] also. The variance [30.9581 7.7958] in x direction and y direction, means variance in x direction is larger than y direction. The possible cause might be camera is not fixed well when taking multiple images, probably hand shaking. But overall, it gives good prediction on the results.

Step 4: Re-run with Matlab toolbox

Rerun the program in MATLAB camera calibration toolbox with the same two orthogonal plane, to make it general and I used 7 pictures out of these 10 pictures, and select one single plane in each picture to calculate the camera intrinsic parameter first, and then I run the program again to calculate the extrinsic parameters for particular pictures.



Here below is my result after calculation for both camera intrinsic parameters and extrinsic parameters. Then we are able to calculate the camera matrix P as

$$P = K[R|t]$$

Focal Length: $fc = [487.72707 \ 467.60019] \text{ +/- } [60.27514 \ 28.11562]$
Principal point: $cc = [179.00000 \ 239.50000] \text{ +/- } [0.00000 \ 0.00000]$
Skew: $\alpha_c = [0.00000] \text{ +/- } [0.00000] \Rightarrow \text{angle of pixel axes} = 90.00000 \text{ +/- } 0.00000 \text{ degrees}$
Distortion: $kc = [-0.04826 \ 0.22394 \ -0.01703 \ 0.00148 \ 0.00000] \text{ +/- } [0.11814 \ 0.50369 \ 0.01208 \ 0.00467 \ 0.00000]$
Pixel error: $err = [0.16985 \ 0.12528]$
Rotation Matrix R = $\begin{bmatrix} 0.0460 & 0.6607 & 0.7492 \\ 0.9963 & -0.0853 & 0.0140 \\ 0.0731 & 0.7458 & -0.6622 \end{bmatrix}$
Translation vector t = $[-131.576717 \ -238.621905 \ 866.868603]^T$
Camera Matrix P = $1.0e+04 * \begin{bmatrix} 0.0004 & 0.0057 & 0.0031 & 1.1374 \\ 0.0060 & 0.0017 & -0.0019 & 1.2004 \\ 0.0000 & 0.0000 & -0.0000 & 0.0108 \end{bmatrix}$

In Comparison:

In step 2, we calculated the

$$f_{cx} = 558.0086 \quad f_{cy} = 532.7424, skew = 1.0835, p_x = 187.4397, p_y = 291.7412$$

P =

$$1.0e+05 * \begin{bmatrix} 0.0051 & 0.0004 & 0.0029 & 1.2809 \\ 0.0014 & 0.0057 & -0.0014 & 1.1218 \\ 0.0000 & 0.0000 & -0.0000 & 0.0101 \end{bmatrix}$$

Discussion:

In MATLAB program, the range for f_{cx} is : $427.45193 \leq f_{cx} \leq 548.00221$,
 $439.48457 < f_{cy} < 495.71581$, $p_x = 179.00000$, $p_y = 239.50000$

It can be seen that , the focal length I calculated $f_{cx} = 558.0086$ $f_{cy} = 532.7424$ is larger, and the same for p_x and p_y . The reason is that I only used one image with two orthogonal plane to calibrate, but the toolbox requires several images on one plane, so the calibration method is totally different. Of course, more pictures give better calibration accuracy. However, if we have a close look at the camera matrix P, we would find the two camera matrix is also very similar to

each other because when calculate the matrix P, it only requires one image to perform the calculation.

- **Discussion about what I have learn from the process**

The homograph method in this assignment requires two orthogonal image plane in the world coordinate system and select points accordingly. In this process, I learned how to calibrate cameras from images step by step with my own code, in the mean time I learned how to use MATLAB functions to calibrate cameras. The final result shows good match, so the method implemented by mine is good. For beginners, it is time-consuming to get all details correct.

- **Challenge**

The weak projective camera model is a camera at infinity for which the scale factors in the two axial image directions are not equal. Such a camera has a projection matrix of the form

$$P = \begin{bmatrix} \alpha_x & 0 & 0 \\ 0 & \alpha_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{r}^{1T} & t_1 \\ \mathbf{r}^{2T} & t_2 \\ \mathbf{0}^T & 1 \end{bmatrix}$$

It has seven degree of freedom, now I can write it in just one matrix

$$P = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$P = [M \mid t]$$

subject to the constraint $m_{11}m_{21} + m_{12}m_{22} + m_{13}m_{23} = 0$. It means the first two rows of P is orthogonal with each other.

Now, I can use these special properties to simplify the general form of 3D -2D homograph:

$$\begin{bmatrix} \mathbf{0}^\top & -w_i \mathbf{X}_i^\top & y_i \mathbf{X}_i^\top \\ w_i \mathbf{X}_i^\top & \mathbf{0}^\top & -x_i \mathbf{X}_i^\top \\ -y_i \mathbf{X}_i^\top & x_i \mathbf{X}_i^\top & \mathbf{0}^\top \end{bmatrix} \begin{pmatrix} \mathbf{P}^1 \\ \mathbf{P}^2 \\ \mathbf{P}^3 \end{pmatrix} = \mathbf{0}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & -X_i & -Y_i & -Z_i & -1 & y_i X_i & y_i Y_i & y_i Z_i & y_i \\ X_i & Y_i & Z_i & 1 & 0 & 0 & 0 & 0 & -x_i X_i & -x_i Y_i & -x_i Z_i & -x_i \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & -X_n & -Y_n & -Z_n & -1 & y_n X_n & y_n Y_n & y_n Z_n & y_n \\ X_i & Y_i & Z_i & 1 & 0 & 0 & 0 & 0 & -x_n X_n & -x_n Y_n & -x_n Z_n & -x_n \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = 0$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & -X_i & -Y_i & -Z_i & -1 & y_i \\ X_i & Y_i & Z_i & 1 & 0 & 0 & 0 & 0 & -x_i \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & -X_n & -Y_n & -Z_n & -1 & y_n \\ X_i & Y_i & Z_i & 1 & 0 & 0 & 0 & 0 & -x_n \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \\ 1 \end{bmatrix} = 0$$

Or it can be written as

$$\begin{bmatrix} 0 & 0 & 0 & 0 & X_i & Y_i & Z_i & 1 \\ X_i & Y_i & Z_i & 1 & 0 & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & X_n & Y_n & Z_n & 1 \\ X_i & Y_i & Z_i & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \end{bmatrix} = \begin{bmatrix} y_1 \\ x_1 \\ \dots \\ \dots \\ y_n \\ x_n \end{bmatrix}$$

Or in a more compact form

$$Ap = x$$

$$p = A^+ x = (A^T A)^{-1} A^T x$$

The above solution is the solution obtained from DLT equations, now it can be further optimized with constraint $p_1 p_5 + p_2 p_6 + p_3 p_7 + p_4 p_8 = 0$ to ensure the orthogonality of the

two rows, then the degree of freedom for matrix form P is just 7. Now we can look at the different errors with different cost functions:

a) *algebraic error*

In this case, the cost function to be optimize is $\|AP\|^2$ with

$$\begin{bmatrix} \mathbf{0}^\top & -w_i \mathbf{X}_i^\top & y_i \mathbf{X}_i^\top \\ w_i \mathbf{X}_i^\top & \mathbf{0}^\top & -x_i \mathbf{X}_i^\top \\ -y_i \mathbf{X}_i^\top & x_i \mathbf{X}_i^\top & \mathbf{0}^\top \end{bmatrix} \begin{pmatrix} P^1 \\ P^2 \\ P^3 \end{pmatrix} = 0$$

However, we have already showed that the above equation can be written in another form after simplification, namely,

$$\begin{bmatrix} 0 & 0 & 0 & 0 & X_i & Y_i & Z_i & 1 \\ X_i & Y_i & Z_i & 1 & 0 & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & X_n & Y_n & Z_n & 1 \\ X_i & Y_i & Z_i & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \end{bmatrix} = \begin{bmatrix} y_1 \\ x_1 \\ \dots \\ y_n \\ x_n \end{bmatrix}$$

So the cost function

$$\|AP\|^2 = \sum_i^n (x_i - P^{1T} X_i)^2 + (y_i - P^{2T} X_i)^2 = \sum_i^n d(x_i - PX)^2$$

The parameters of P is 7 degree of freedom due to the orthogonality, so I can add the constraint to the cost function and the new cost function is:

$$F = \sum_i^n d(x_i - PX)^2 + (p_1 p_5 + p_2 p_6 + p_3 p_7)$$

```
opt = optimoptions('lsqcurvefit','Algorithm','levenberg-marquardt')
[q,res] = lsqcurvefit(@fun,p,xWn,obj,[],[],opt); % Refined parameter

function [ F ] = fun(p,X)
% This function describes the camera projection
% from the world plane to the image plane.
% This function is used for the Levenberg-Marquardt method.
P = [p(1) p(2) p(3) p(4); p(5) p(6) p(7) p(8); 0 0 0 1]; % Intrinsic parameter matrix K
[M,~] = size(X);
F = zeros(3*M+1,1);
for k = 1:M
    Y = P * X(k,:);
    F(3*k-2:3*k,:) = Y'/Y(3);
end
% error = sum(sum((coors - F).^2))/320
F(3*M+1,1) = [p(1) p(2) p(3)]*[p(5) p(6) p(7)]';
end
```

After optimization, we can compare the P in the projective world and this weak projective form:

Projective: P =

0.5063	0.0418	0.2851	126.7492
0.1383	0.5681	-0.1391	111.0071
0.0007	0.0002	-0.0007	1.0000

Weak Projective: P =

0.3481	0.0079	0.3938	129.8975
0.0151	0.4997	-0.0256	114.9750
0	0	0	1.0000

Now we compare the errors in terms of geometric errors:

$$error = \frac{1}{n} \sum_i^n d(x_i - PX)^2$$

Projective pixel error: 1.212394e+00

Weak Projective pixel error: 2.772136e+00

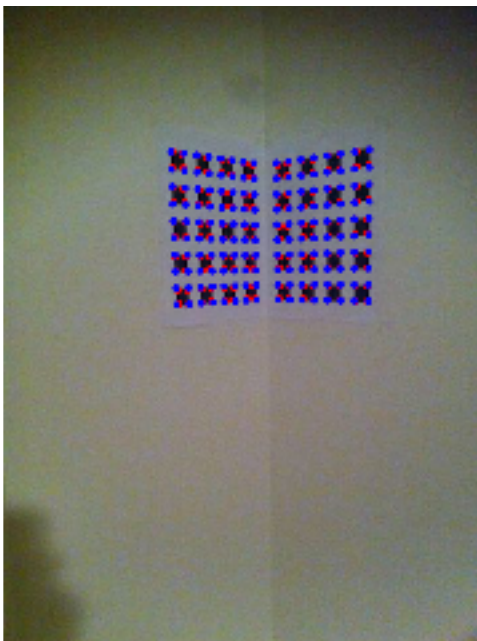


Figure 4 Projective Geometric Error

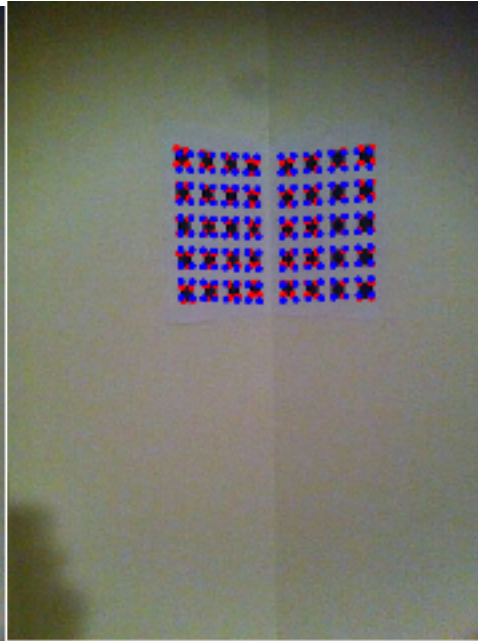


Figure 5 Weak Projective Geometric Error

In this case, the projective model is more accurate.

b) *reprojection error*

Now, we can look at the projection error, we will use the same cost function but make some necessary modification because the parameters we are going to optimize now becomes $3n + 7$ and n is the total number of 3D points selected in the world coordinate

system, thus we use $f(P)$ instead of PX in this expression because we include X as part of the parameter to be optimized.

$$F = \sum_i^n d(x_i - f(P))^2 + (p_1 p_5 + p_2 p_6 + p_3 p_7)$$

In code, the size of P is $8 + 3*n$

```
para = zeros(8+3*M2,1);
para(1:8,:) = p; para(9:end,:) = reshape(xWn(:,1:3)',[3*M2,1]);
opt = optimoptions('lsqcurvefit','Algorithm','levenberg-marquardt');
[q,res] = lsqcurvefit(@funprojection,para,xWn,obj,[],[],opt); % Refined parameter
P = [q(1) q(2) q(3) q(4); q(5) q(6) q(7) q(8);0 0 0 1];
xWn(1:end,1:3) = reshape(q(9:end),[3,M2])';
P = T2\P*T1;
xW = xWn*(T1')^(-1);

function [ F ] = funprojection(p,X)
% This function describes the camera projection
% from the world plane to the image plane.
% This function is used for the Levenberg-Marquardt method.
P = [p(1) p(2) p(3) p(4); p(5) p(6) p(7) p(8);0 0 0 1]; % Intrinsic parameter
matrix
len = length(p);
X = reshape(p(9:end),[3 (len-8)/3])';
[M,~] = size(X);
F = zeros(3*M+1,1);
for k = 1:M
    Y = P * [X(k,:) 1]';
    F(3*k-2:3*k,:) = Y'/Y(3);
end
% error = sum(sum((coors - F).^2))/320
F(3*M+1,1) = [p(1) p(2) p(3)]*[p(5) p(6) p(7)]';
end
```

After optimization, we can compare the P in the projective world and this weak projective form:

Projective: $P =$

0.5063	0.0418	0.2851	126.7492
0.1383	0.5681	-0.1391	111.0071
0.0007	0.0002	-0.0007	1.0000

Weak Projective: $P =$

0.3482	0.0090	0.3936	129.7733
0.0154	0.4997	-0.0250	114.8945
0	0	0	1.0000

Now we compare the errors in terms of geometric errors:

$$error = \frac{1}{n} \sum_i^n d(x_i - PX)^2$$

Projective pixel error: 1.212394e+00

Weak Projective pixel error: 6.753827e-26

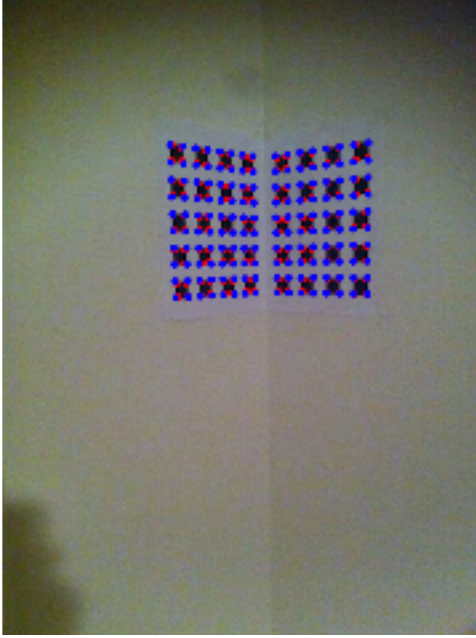


Figure 6 Projective

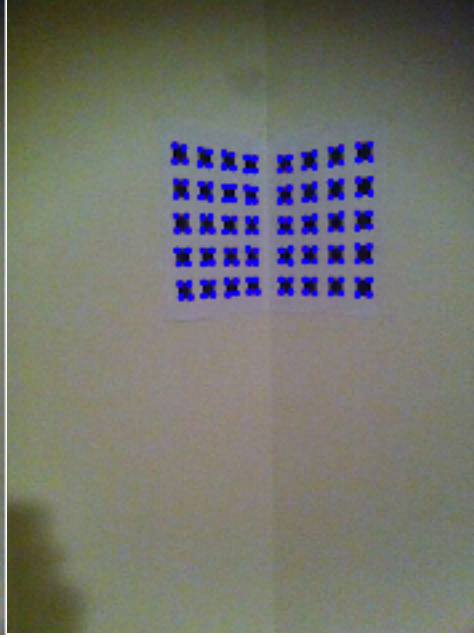


Figure 7 Weak Projective

In this case, the weak projective with reprojection has perfect matching. But the computational time for this one is bit longer.

c) Sampson error

As an approximation, the Sampson error for a single point has the following mathematic expression:

$$\delta_x = -J^T(JJ^T)\epsilon$$

And the norm

$$||\delta_x||^2 = \delta_x^T \delta_x = \epsilon^T (JJ^T) \epsilon$$

Here, the mathematic expression for each term is

$$\epsilon = \begin{bmatrix} 0 & 0 & 0 & 0 & X_i & Y_i & Z_i & 1 \\ X_i & Y_i & Z_i & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \end{bmatrix} - \begin{bmatrix} y_1 \\ x_1 \end{bmatrix}$$

$$J = \begin{bmatrix} \frac{\partial(Ap)_1}{\partial X} & \frac{\partial(Ap)_1}{\partial Y} & \frac{\partial(Ap)_1}{\partial Z} & \frac{\partial(Ap)_1}{\partial x} & \frac{\partial(Ap)_1}{\partial y} \\ \frac{\partial(Ap)_2}{\partial X} & \frac{\partial(Ap)_2}{\partial Y} & \frac{\partial(Ap)_2}{\partial Z} & \frac{\partial(Ap)_2}{\partial x} & \frac{\partial(Ap)_2}{\partial y} \end{bmatrix}$$

$$Ap = \begin{bmatrix} p_5 X_i + p_6 Y_i + p_7 Z_i + p_8 - y_i \\ p_1 X_i + p_2 Y_i + p_3 Z_i + p_4 - x_i \end{bmatrix}$$

$$J = \begin{bmatrix} p_5 & p_6 & p_7 & 0 & -1 \\ p_1 & p_2 & p_3 & -1 & 0 \end{bmatrix}$$

Applying this to all the points on 3D to 2D correspondence, the error must be summed:

$$D = \sum_1^n \epsilon^T (JJ^T) \epsilon$$

Incorporate the constraint for orthogonality, then the cost function becomes:

$$D = \sum_1^n \epsilon^T (JJ^T) \epsilon + (p_1 p_5 + p_2 p_6 + p_3 p_7)$$

In code ,the size of parameter p is 8, but after we add the orthogonality constraint ,the degree of freedom becomes 7.

```
opt = optimoptions('lsqcurvefit','Algorithm','levenberg-marquardt');
[q,res] = lsqcurvefit(@funsampson,p,xWn,obj,[],[],opt); % Refined parameter
P = [q(1) q(2) q(3) q(4); q(5) q(6) q(7) q(8); 0 0 0 1];
P = T2\P*T1;

function [ F ] = funsampson(p,X)
% This function describes the camera projection
% from the world plane to the image plane.
% This function is used for the Levenberg-Marquardt method.
global coors;
P = [p(1) p(2) p(3) p(4); p(5) p(6) p(7) p(8); 0 0 0 1]; % Intrinsic parameter matrix
[M, ~] = size(X);
F = zeros(M+1,1);
for k = 1:M
    Y = P * X(k,:)';
    e = Y(1:2)/Y(3) - coors(k,1:2)';
    J = [p(1) p(2) p(3) -1 0; p(5) p(6) p(7) 0 -1];
    error = e'*(J*J')^(-1)*e;
    F(k) = sqrt(error);
end
F(M+1,1) = [p(1) p(2) p(3)]*[p(5) p(6) p(7)]';
end
```

Projective: P =

0.5063	0.0418	0.2851	126.7492
0.1383	0.5681	-0.1391	111.0071
0.0007	0.0002	-0.0007	1.0000

Weak Projective: P =

0.3482	0.0079	0.3940	129.8674
0.0151	0.5003	-0.0257	114.9066
0	0	0	1.0000

Projective pixel error: 1.212394e+00

Weak Projective pixel error: 2.465077e+00

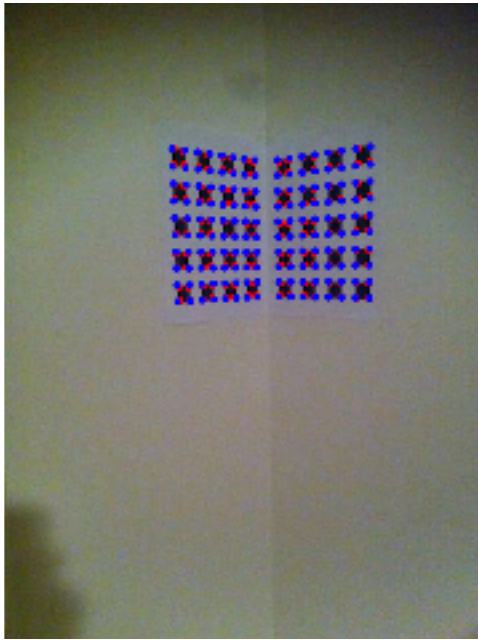


Figure 8 Projective

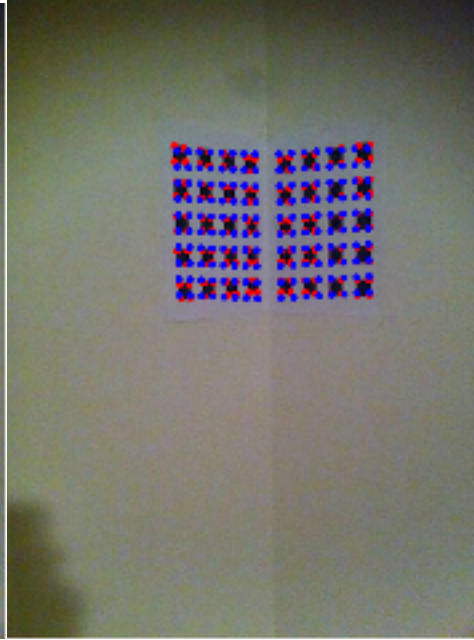


Figure 9 Weak Projective

We can see that the error is large in for Sampson error.

• Summary:

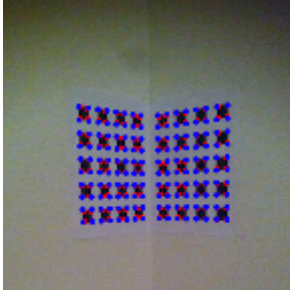
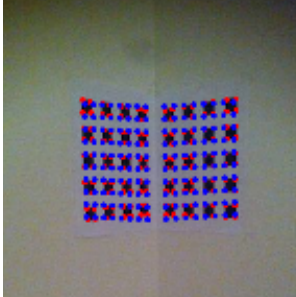
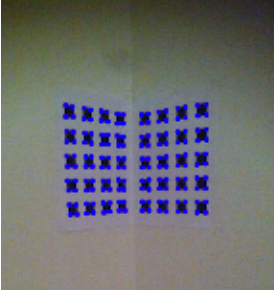
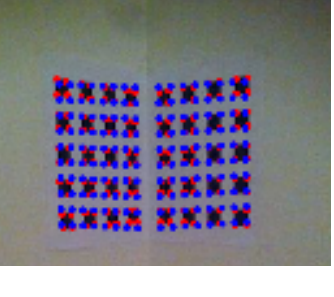
Camera matrix P in each case

Projective	Weak Projective Algebraic Error	Weak Projective Reprojection Error	Weak Projective Sampson Error
0.5063 0.0418 0.2851 126.7492 0.1383 0.5681 -0.1391 111.0071 0.0007 0.0002 -0.0007 1.0000	0.3481 0.0079 0.3938 129.8975 0.0151 0.4997 -0.0256 114.9750 0 0 0 1.0000	0.3482 0.0090 0.3936 129.7733 0.0154 0.4997 -0.0250 114.8945 0 0 0 1.0000	0.3482 0.0079 0.3940 129.8674 0.0151 0.5003 -0.0257 114.9066 0 0 0 1.0000

Averaged Projective pixel error: $error = \frac{1}{n} \sum_i^n d(x_i - PX)^2$

Projective	Weak Projective Algebraic Error	Weak Projective Reprojection Error	Weak Projective Sampson Error
1.212394e+00	2.772136e+00	6.753827e-26	2.465077e+00

Visualize the projective 3D points on image:

Projective	Weak Projective Algebraic Error	Weak Projective Reprojection Error	Weak Projective Sampson Error
			

Findings:

The weak projective reprojection gives the most accurate results because it optimizes not only the parameter P but also the world coordinate X with averaged projective error $6.753827e-26$, while the Weak Projective Algebraic Error gives the worst averaged projective error $2.772136e+00$, weak Projective Sampson Error provides slightly better averaged projective error $2.465077e+00$, in contrast the projective model gives better result with averaged projective error $1.212394e+00$.

Overall speaking, the projective model give better approximation compared with weak projective model because weak projective model is the simplified model. However, if we use the full reprojection error to optimize both the camera matrix and also world coordinates, the result is far better than the projective model because project model only used the geometric distance error.