ECE 661 Computer Vision

Homework 8

Yi-Hong Liao | liao163@purdue.edu | PUID 0031850073

1. Logic

Implement Zhang's calibration algorithm and use non-linear least square to optimize camera parameters.

2. Step

- Dataset

  There are two datasets used in this work. The provided dataset with 40 camera poses and my own dataset with 20 camera poses.

- Corner detection

  I use OpenCV canny edge detection to extract the edges and use OpenCV Hough transform to detect lines. Then, I find the intersection of lines as the corners.

- Camera calibration (including radial distortion)
  - I find the homographies H between corners in the world coordinate and corners in the image coordinates.
  - Then I implemented Zhang's algorithm. By using

$$\overrightarrow{h_1}^T \omega \overrightarrow{h_1} - \overrightarrow{h_2}^T \omega \overrightarrow{h_2} = 0$$

$$\overrightarrow{h_1}^T \omega \overrightarrow{h_2} = 0$$

we can solve for ω. Then we use Cholesky decomposition to recover K.

$$K = \begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$y_0 = \frac{\omega_{12}\omega_{13} - \omega_{11}\omega_{23}}{\omega_{11}\omega_{22} - \omega_{12}\omega_{12}}$$

$$\lambda = \omega_{33} - \frac{\omega_{13}^2 + x_0(\omega_{12}\omega_{13} - \omega_{11}\omega_{23})}{\omega_{11}}$$

$$\alpha_x = \sqrt{\frac{\lambda}{\omega_{11}}}$$

$$\alpha_y = \sqrt{\frac{\lambda\omega_{11}}{\omega_{11}\omega_{22} - \omega_{12}\omega_{12}}}$$

$$s = -\frac{\omega_{12}\alpha_x^2\alpha_y}{\lambda}$$

$$x_0 = \frac{s x_0}{\alpha_y} - \frac{\omega_{13}\alpha_x^2}{\lambda}$$

For each pose, we can calculate the extrinsic parameters $R = [\overrightarrow{r_1} \quad \overrightarrow{r_2} \quad \overrightarrow{r_3}]$ and $\vec{t}$.

$$\xi = \frac{1}{\left\|K^{-1}\overrightarrow{h_1}\right\|}$$

$$\overrightarrow{r_1} = \xi K^{-1}\overrightarrow{h_1}$$

$$\overrightarrow{r_2} = \xi K^{-1}\overrightarrow{h_2}$$

$$\overrightarrow{r_3} = \overrightarrow{r_1} \times \overrightarrow{r_2}$$

$$\vec{t} = \xi K^{-1}\overrightarrow{h_3}$$

- Refine the parameters using non-linear least square
  The goal is to minimize

$$d_{geom}^2 = \left\|\vec{X} - \vec{f}(\vec{p})\right\|^2,$$

where $\vec{p} = (K, R_i, t_i k_1, k_2)^T, i = 1, 2, \ldots\ldots$

The 3-paramters representation of a rotation matrix R is $\vec{w}$.

$$\psi = arccos\frac{trace(R) - 1}{2}$$

$$\vec{w} = \frac{\psi}{2\,sin\,\psi}\begin{pmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{pmatrix}$$

To incorporate radial distortion

$$\hat{x}_{rad} = \hat{x} + (\hat{x} - x_0)[k_1 r^2 + k_2 r^4]$$
$$\hat{y}_{rad} = \hat{y} + (\hat{y} - y_0)[k_1 r^2 + k_2 r^4]$$
$$r^2 = (\hat{x} - x_0)^2 + (\hat{y} - y_0)^2.$$

3. Result

- **Theory question**

  NO. The pixels of the image of the absolute conic are imaginary. ω is independent of the extrinsic matrix, and it can help us find the intrinsic matrix of the camera.

- **Edge detection**

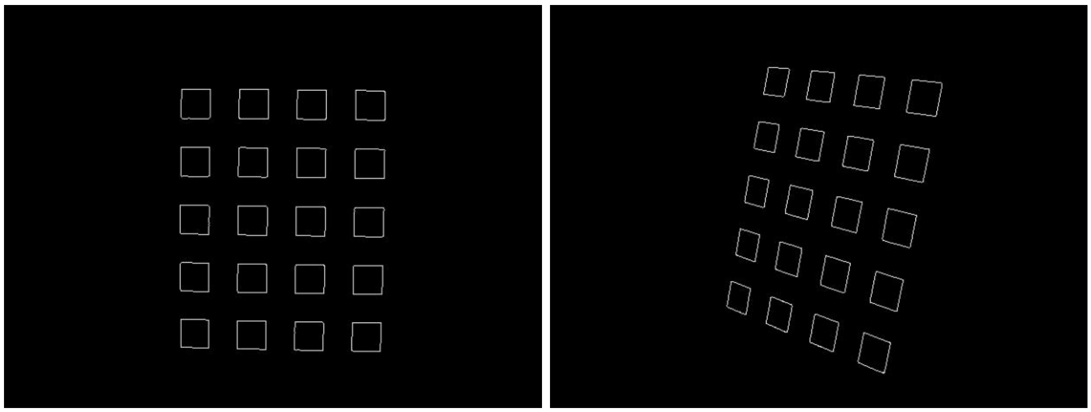

Figure 1: Edge detection results from two poses of the provided dataset.

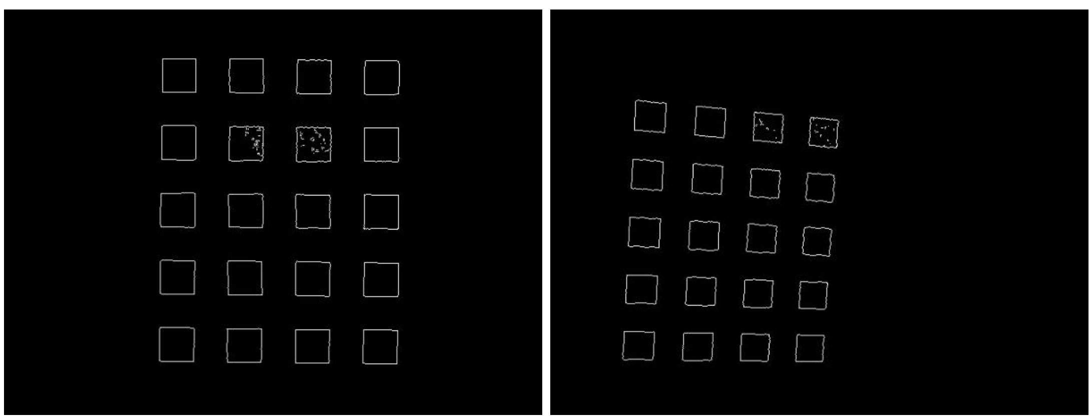

Figure 2: Edge detection results from two poses of my dataset.

- **Hough line fitting**

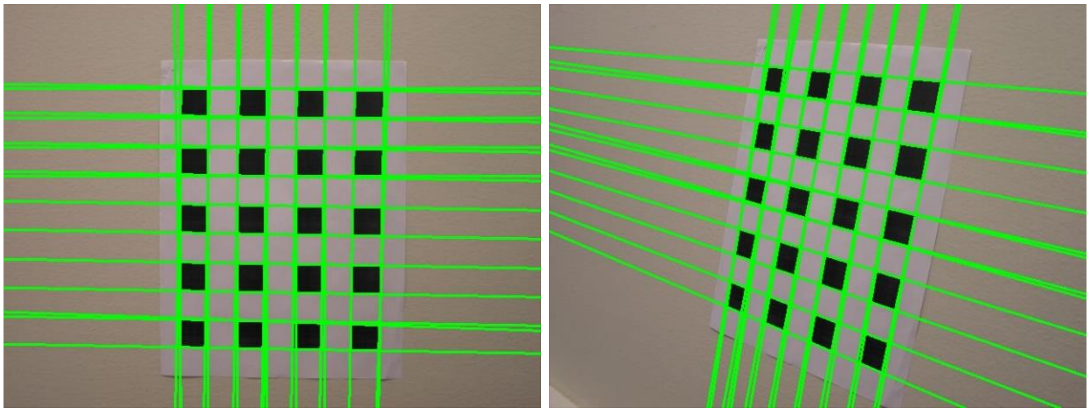

Figure 3: Hough line fitting results from two poses of the provided dataset.

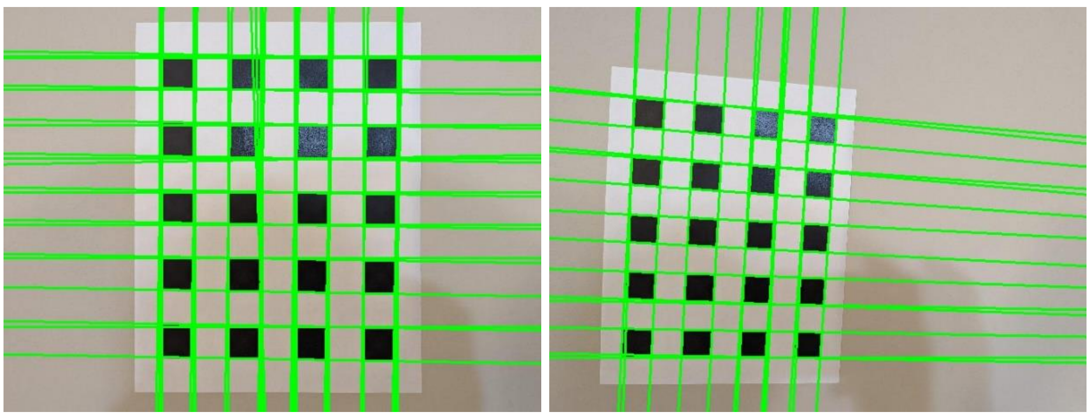

Figure 4: Hough line fitting results from two poses of my dataset.

- **Corner detection**

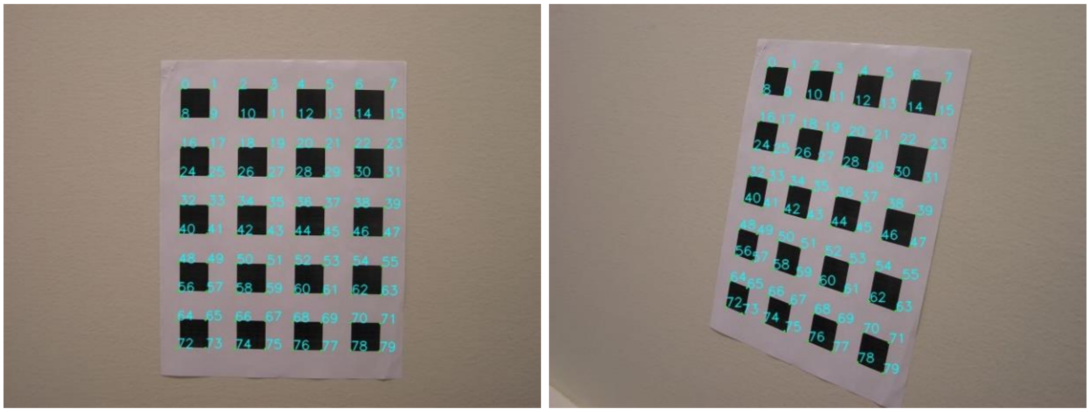

Figure 5: Corner detection results from two poses of the provided dataset.

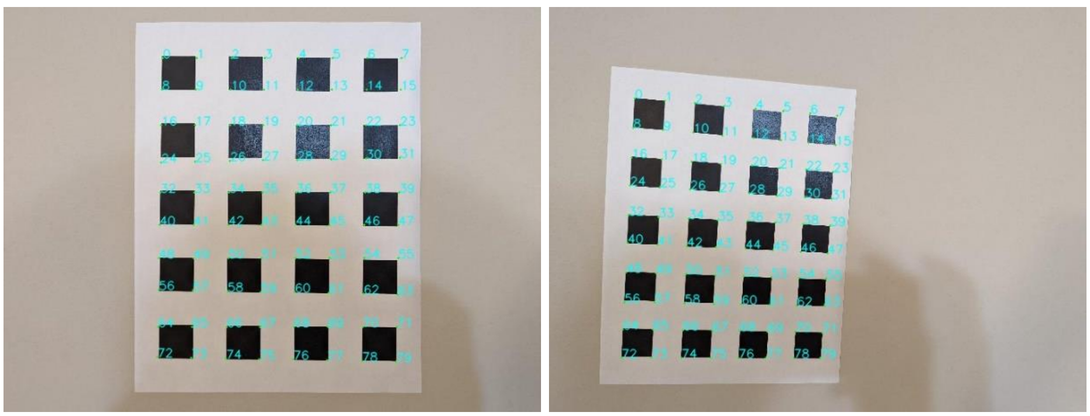

Figure 6: Corner detection results from two poses of my dataset.

- **Fixed image with the reprojected corners (blue: gt, red: reproject)**

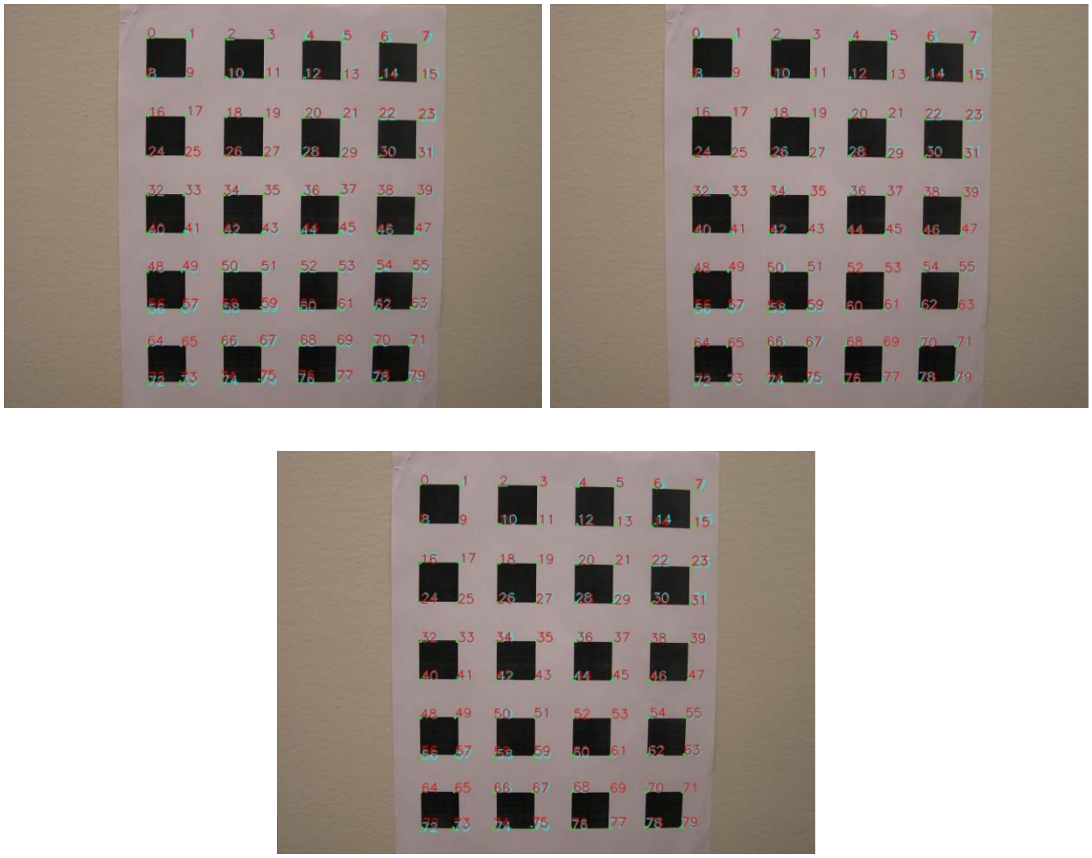

Figure 7: Reproject results from one of the poses of the provided dataset. (Top left: linear least square, top right: LM w/o radial distortion, bottom: LM w/ radial distortion)

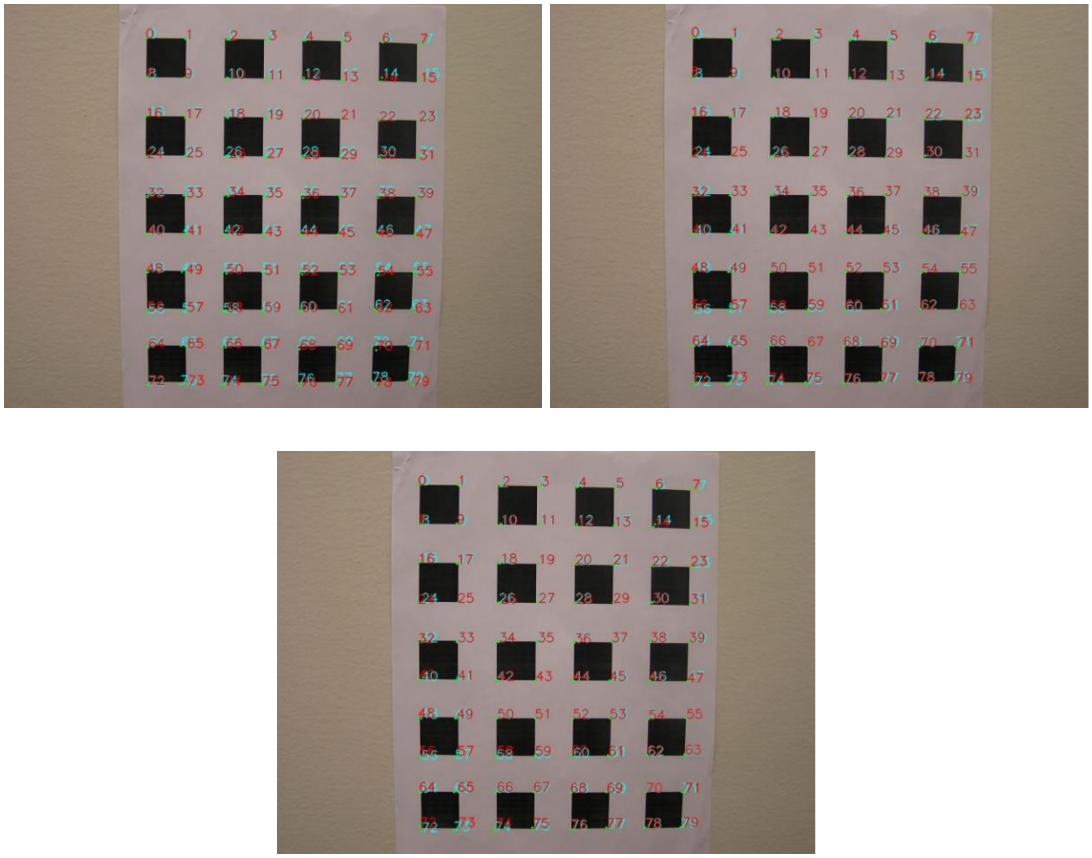

Figure 8: Reproject results from one of the poses of the provided dataset. (Top left: linear least square, top right: LM w/o radial distortion, bottom: LM w/ radial distortion)

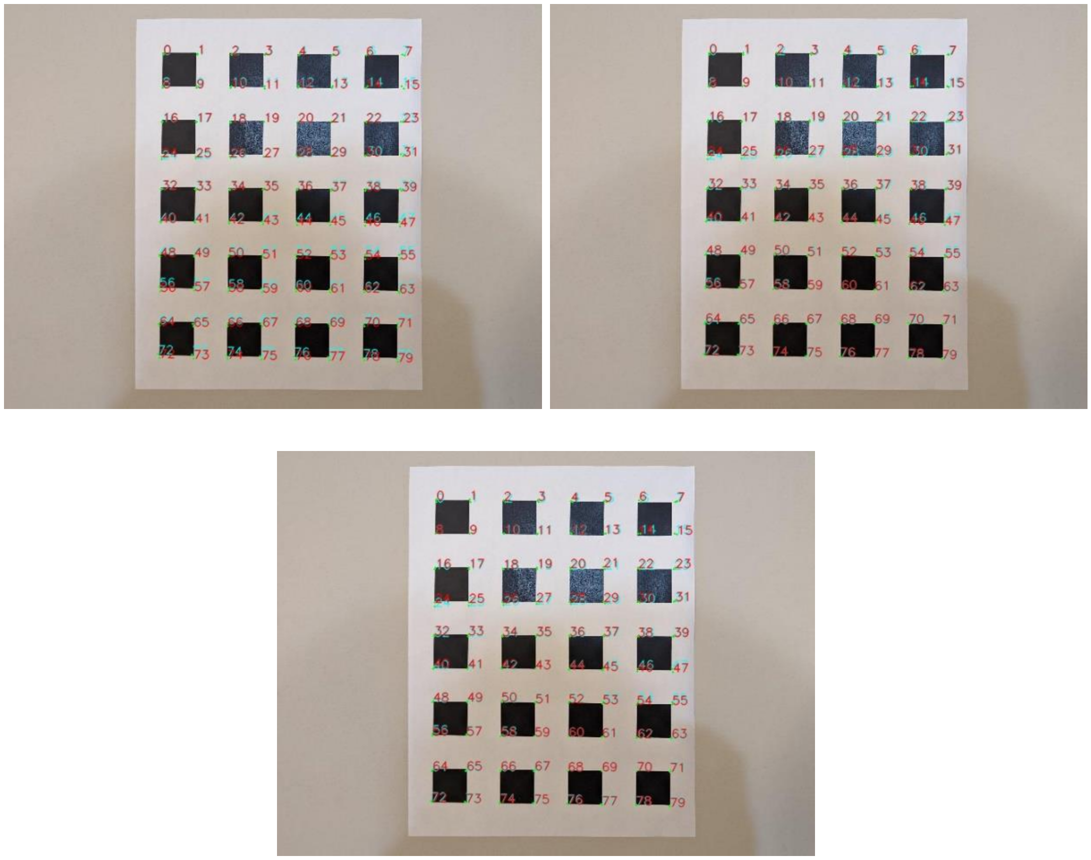

Figure 9: Reproject results from one of the poses of my dataset. (Top left: linear least square, top right: LM w/o radial distortion, bottom: LM w/ radial distortion)

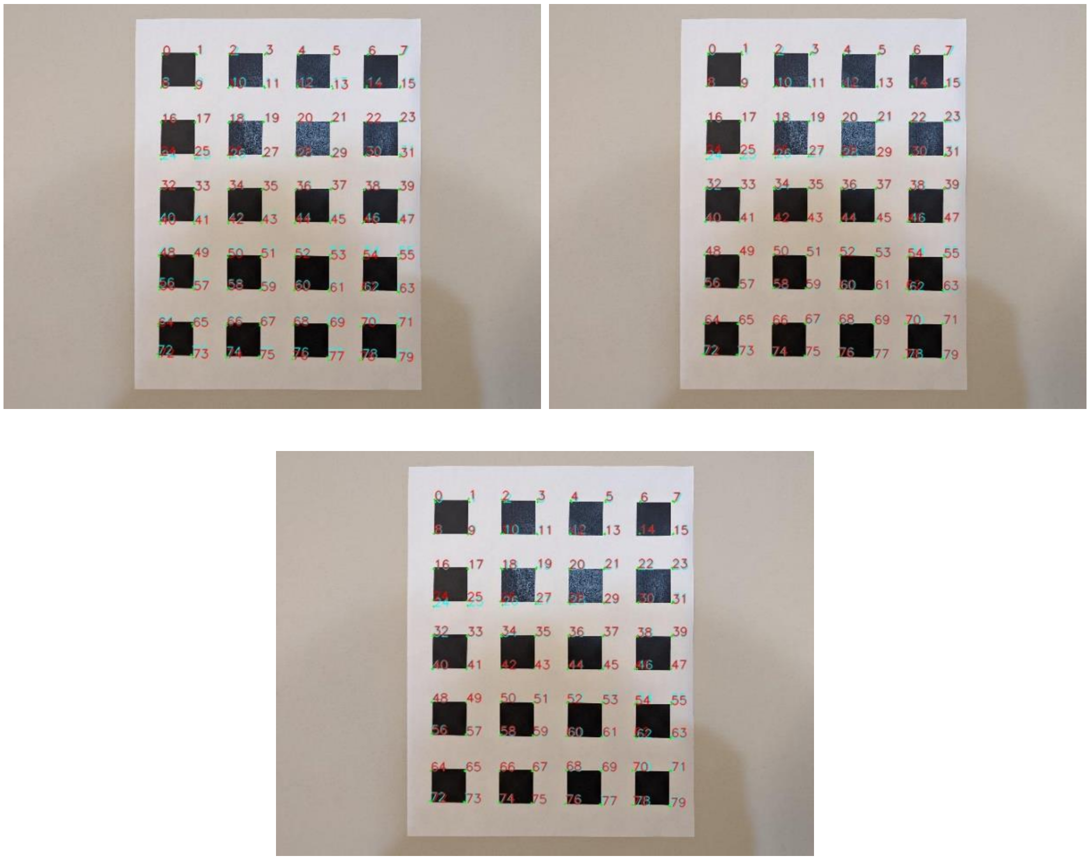

Figure 10: Reproject results from one of the poses of my dataset. (Top left: linear least square, top right: LM w/o radial distortion, bottom: LM w/ radial distortion)

- **Camera calibration parameters**

Provided Dataset

	Linear least square		
	K	**R**	**t**
Pos 1	$\begin{bmatrix} 720.44 & -8.04 & 325.72 \\ 0 & 722.00 & 237.48 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.790 & -0.178 & 0.587 \\ 0.196 & 0.980 & 0.033 \\ -0.581 & 0.089 & 0.809 \end{bmatrix}$	$\begin{bmatrix} -51.86 \\ -125.78 \\ 554.89 \end{bmatrix}$
Pos 2	$\begin{bmatrix} 720.44 & -8.04 & 325.72 \\ 0 & 722.00 & 237.48 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.999 & -0.008 & 0.048 \\ 0.010 & 0.999 & -0.044 \\ -0.048 & 0.044 & 0.998 \end{bmatrix}$	$\begin{bmatrix} -85.49 \\ -100.61 \\ 528.287 \end{bmatrix}$

	LM w/o radial distortion		
	K	**R**	**t**
Pos 1	$\begin{bmatrix} 721.69 & --8.63 & 329.30 \\ 0 & 722.87 & 238.83 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.795 & -0.179 & 0.579 \\ 0.196 & 0.980 & 0.034 \\ -0.574 & 0.086 & 0.814 \end{bmatrix}$	$\begin{bmatrix} -54.84 \\ -127.37 \\ 557.12 \end{bmatrix}$
Pos 2	$\begin{bmatrix} 721.69 & --8.63 & 329.30 \\ 0 & 722.87 & 238.83 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.999 & -0.007 & 0.047 \\ 0.009 & 0.999 & -0.044 \\ -0.046 & 0.044 & 0.998 \end{bmatrix}$	$\begin{bmatrix} -88.22 \\ -101.64 \\ 528.831 \end{bmatrix}$

	LM w radial distortion		
	K	**R**	**t**
Pos 1	$\begin{bmatrix} 725.11 & -8.70 & 328.65 \\ 0 & 726.55 & 238.81 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.794 & -0.179 & 0.580 \\ 0.196 & 0.980 & 0.034 \\ -0.575 & 0.086 & 0.813 \end{bmatrix}$	$\begin{bmatrix} -54.28 \\ -127.35 \\ 557.59 \end{bmatrix}$
Pos 2	$\begin{bmatrix} 725.11 & -8.70 & 328.65 \\ 0 & 726.55 & 238.81 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.999 & -0.008 & 0.047 \\ 0.009 & 0.999 & -0.040 \\ -0.047 & 0.040 & 0.998 \end{bmatrix}$	$\begin{bmatrix} -87.74 \\ -101.65 \\ 529.81 \end{bmatrix}$
	$\mathbf{k_1}$	$\mathbf{k_2}$	
	2.209023490667813e-07	1.3670161069250435e-12	

My Dataset

	Linear least square		
	K	**R**	**t**
Pos 1	$\begin{bmatrix} 516.69 & -3.32 & 338.94 \\ 0 & 514.50 & 262.88 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1.000 & -0.006 & 0.019 \\ 0.006 & 1.000 & -0.002 \\ -0.019 & 0.002 & 1.000 \end{bmatrix}$	$\begin{bmatrix} -82.63 \\ -118.50 \\ 307.08 \end{bmatrix}$
Pos 2	$\begin{bmatrix} 516.69 & -3.32 & 338.94 \\ 0 & 514.50 & 262.88 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.987 & -0.064 & -0.144 \\ 0.056 & 0.997 & -0.056 \\ -0.148 & 0.047 & 0.988 \end{bmatrix}$	$\begin{bmatrix} -158.01 \\ -101.34 \\ 350.94 \end{bmatrix}$

	LM w/o radial distortion		
	K	**R**	**t**
Pos 1	$\begin{bmatrix} 515.89 & -3.32 & 336.80 \\ 0 & 513.67 & 263.50 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1.000 & -0.007 & 0.019 \\ 0.007 & 1.000 & -0.002 \\ -0.019 & 0.002 & 1.000 \end{bmatrix}$	$\begin{bmatrix} -81.82 \\ -119.97 \\ 307.62 \end{bmatrix}$
Pos 2	$\begin{bmatrix} 515.89 & -3.32 & 336.80 \\ 0 & 513.67 & 263.50 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.988 & -0.063 & -0.144 \\ 0.056 & 0.997 & -0.055 \\ -0.147 & 0.046 & 0.988 \end{bmatrix}$	$\begin{bmatrix} -156.47 \\ -101.63 \\ 350.05 \end{bmatrix}$

	LM w radial distortion		
	K	**R**	**t**
Pos 1	$\begin{bmatrix} 514.88 & -3.36 & 337.53 \\ 0 & 512.80 & 263.45 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1.000 & -0.007 & 0.019 \\ 0.007 & 1.000 & -0.002 \\ -0.019 & 0.002 & 1.000 \end{bmatrix}$	$\begin{bmatrix} -82.16 \\ -119.93 \\ 308.11 \end{bmatrix}$
Pos 2	$\begin{bmatrix} 515.89 & -3.32 & 336.80 \\ 0 & 513.67 & 263.50 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.988 & -0.063 & -0.140 \\ 0.056 & 0.997 & -0.055 \\ -0.143 & 0.046 & 0.989 \end{bmatrix}$	$\begin{bmatrix} -157.13 \\ -101.60 \\ 351.07 \end{bmatrix}$
	k_1	**k_2**	
	1.5393265063742962e-07	-1.3489504029772135e-12	

The parameters matched with the manually measured ground-truth.

- **Reprojection error**

Provided dataset

	Linear least square	
	Mean	**Variance**
Pos 1	2.183	1.380
Pos 2	4.087	3.607

	LM w/o radial distortion	
	Mean	**Variance**
Pos 1	1.810	0.849
Pos 2	2.316	1.798

	LM w radial distortion	
	Mean	**Variance**
Pos 1	1.852	1.261
Pos 2	2.279	2.166

My dataset

	Linear least square	
	Mean	**Variance**
Pos 1	3.076	2.752
Pos 2	2.766	1.794

	LM w/o radial distortion	
	Mean	**Variance**
Pos 1	1.676	0.845
Pos 2	1.774	1.339

	LM w radial distortion	
	Mean	**Variance**
Pos 1	1.728	0.915
Pos 2	1.814	1.429

Reprojection error decreases when using non-linear least square optimization. It decreases further after including radial distortion. Some errors increase after including radial distortion because the ground truth corners are not accurate.

4. Source Code

```
#!/usr/bin/env python
# coding: utf-8

# In[10]:


import numpy as np
import matplotlib.pyplot as plt
import cv2
import math
import os
from os import listdir
from scipy.spatial import distance
from scipy import optimize
import statistics

from pathlib import Path


# In[11]:


def find_intersection(line1, line2):
    r1, theta1 = line1[0], line1[1]
    r2, theta2 = line2[0], line2[1]

    A = [[np.cos(theta1), np.sin(theta1)],
        [np.cos(theta2), np.sin(theta2)]]
    b = [r1, r2]
    x, y = np.linalg.solve(A, b)
    x, y = int(np.round(x)), int(np.round(y))
    return x, y

def draw_houghlines(image, lines):
    # Code modified from https://www.geeksforgeeks.org/line-detection-
python-opencv-houghline-method/
    for line in lines:
        r, theta = line[0], line[1]
        # Stores the value of cos(theta) in a
        a = np.cos(theta)

        # Stores the value of sin(theta) in b
        b = np.sin(theta)

        # x0 stores the value rcos(theta)
        x0 = a*r

        # y0 stores the value rsin(theta)
        y0 = b*r

        # x1 stores the rounded off value of (rcos(theta)-1000sin(theta))
        x1 = int(x0 + 1000*(-b))

        # y1 stores the rounded off value of (rsin(theta)+1000cos(theta))
```

```
        y1 = int(y0 + 1000*(a))

        # x2 stores the rounded off value of (rcos(theta)+1000sin(theta))
        x2 = int(x0 - 1000*(-b))

        # y2 stores the rounded off value of (rsin(theta)-1000cos(theta))
        y2 = int(y0 - 1000*(a))

        # cv2.line draws a line in img from the point(x1,y1) to (x2,y2).
        # (0,0,255) denotes the colour of the line to be
        # drawn. In this case, it is red.
        cv2.line(image, (x1, y1), (x2, y2), (0, 255, 0), 2)
    return image

def draw_corners(image, corners, color):
    for i, corner in enumerate(corners):
        image = cv2.circle(image, corner, radius=1, color=(0, 255, 0),
thickness=-1)
        image = cv2.putText(image, str(i), corner,
cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 1, cv2.LINE_AA)
    return image

def corner_detection(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
#     gray = cv2.GaussianBlur(gray, (3, 3), 1)
    edges = cv2.Canny(gray, 255*1.5, 255)
#     cv2.imwrite("output1/edge1.jpg", edges)

    lines = cv2.HoughLines(edges, 1, np.pi/180, 50)
    lines = np.squeeze(lines)

    tmp_img = image.copy()
    lines_img = draw_houghlines(tmp_img, lines)
#     cv2.imwrite("output1/line1.jpg", lines_img)

    h_lines = [line for line in lines if line[1] > np.pi/4 and line[1] <
3*np.pi/4]
    v_lines = [line for line in lines if line[1] <= np.pi/4 or line[1] >=
3*np.pi/4]

    h_lines = sorted(h_lines, key=lambda x: x[0])
    v_lines = sorted(v_lines, key=lambda x: np.abs(x[0]))

    corners = []
    for h_line in h_lines:
        for v_line in v_lines:
            x, y = find_intersection(h_line, v_line)
            corners.append([x, y])

    filter_corners = corners.copy()
    for corner in corners:
        if sum(np.squeeze(distance.cdist([corner], filter_corners) < 20))
> 1:
            filter_corners.remove(corner)

    tmp_img = image.copy()
    corners_img = draw_corners(tmp_img, filter_corners, (255,255,0))
```

```
#     cv2.imwrite("output1/corners1.jpg", corners_img)

    return filter_corners, edges, lines_img, corners_img

def make_grid(num_row, num_col, scale_x, scale_y):
    x = np.linspace(0, num_col-1, num_col) * scale_x
    y = np.linspace(0, num_row-1, num_row) * scale_y
    xv, yv = np.meshgrid(x, y)
    xv = xv.reshape((-1, 1))
    yv = yv.reshape((-1, 1))
    grid = np.concatenate((xv, yv), axis=1)
    return grid


def calculate_homography(X, Xp):
    n = len(X)
    A = np.zeros((n*2, 8))
    B = np.zeros((n*2, 1))
    for i in range(n):
        A[2*i][0] = X[i][0]
        A[2*i][1] = X[i][1]
        A[2*i][2] = 1
        A[2*i][6] = -X[i][0]*Xp[i][0]
        A[2*i][7] = -X[i][1]*Xp[i][0]
        A[2*i+1][3] = X[i][0]
        A[2*i+1][4] = X[i][1]
        A[2*i+1][5] = 1
        A[2*i+1][6] = -X[i][0]*Xp[i][1]
        A[2*i+1][7] = -X[i][1]*Xp[i][1]
        B[2*i] = Xp[i][0]
        B[2*i+1] = Xp[i][1]
    H = np.linalg.inv(A.transpose().dot(A)).dot(A.transpose()).dot(B)
    H = np.append(H, 1)
    H = H.reshape(3, 3)
    return H

def find_omega(Hs):
    V = []
    for H in Hs:
        h11, h12, h13 = H[0,0], H[1,0], H[2,0]
        h21, h22, h23 = H[0,1], H[1,1], H[2,1]
        v1 = [h11*h21, h11*h22+h12*h21, h12*h22,
h13*h21+h11*h23, h13*h22+h12*h23, h13*h23]
        v2 = [h11**2-h21**2, 2*h11*h12-2*h21*h22, h12**2-h22**2,
2*h11*h13-2*h21*h23, 2*h12*h13-2*h22*h23, h13**2-h23**2]
        V.append(v1)
        V.append(v2)
    V = np.asarray(V)
    VT_V = np.transpose(V).dot(V)

    _, _, v = np.linalg.svd(VT_V)
    b = v[-1]
    omega = np.array([[b[0], b[1], b[3]], [b[1], b[2], b[4]], [b[3], b[4],
b[5]]])

    return omega
```

```
def find_k(omega):
    w11, w12, w13 = omega[0, 0], omega[0, 1], omega[0, 2]
    w21, w22, w23 = omega[1, 0], omega[1, 1], omega[1, 2]
    w31, w32, w33 = omega[2, 0], omega[2, 1], omega[2, 2]

    x0 = (w12*w13-w11*w23) / (w11*w22-w12**2)
    l = w33 - (w13**2+x0*(w12*w13-w11*w23)) / w11
    alpha_x = math.sqrt(l/w11)
    alpha_y = math.sqrt((l*w11) / (w11*w22-w12**2))
    s = -(w12*(alpha_x**2)*alpha_y) / l
    y0 = (s*x0)/alpha_y - (w13*alpha_x**2)/l
    K = np.array([[alpha_x, s, y0], [0, alpha_y, x0], [0, 0, 1]])
#??????????????????????????????????????????????????????????????
    return K

def find_extrinsic(Hs, K):
    # Modified from Homework 9 Xingguang Zhang (Fall 2020)
    Rs = []
    ts = []
    for H in Hs:
        RT_tmp = np.linalg.inv(K).dot(H)
        E = 1 / np.linalg.norm(RT_tmp[:,0])
        RT = RT_tmp * E
        r3 = np.cross(RT[:,0], RT[:,1])
        Q = RT.copy()
        Q[:, 2] = r3
        u, _, v = np.linalg.svd(Q)
        R = np.dot(u, v)
        Rs.append(R)
        ts.append(RT[:,2])

    return Rs, ts

def construct_param(K, Rs, ts, dist_param=[0, 0]):
    K_param = [K[0, 0], K[0, 1], K[0, 2], K[1, 1], K[1, 2]]

    rt_params = np.array([])
    for R, t in zip(Rs, ts):
        phi = np.arccos((np.trace(R)-1)/2)
        w = np.array([R[2][1]-R[1][2], R[0][2]-R[2][0], R[1][0]-R[0][1]])
* phi/(2*np.sin(phi))
        t_f = t.flatten()
        rt_params = np.concatenate((rt_params, w, t_f))

    params = np.concatenate((K_param, dist_param, rt_params))

    return params

def restore_param(params):
    K = np.array([[params[0], params[1], params[2]], [0, params[3],
params[4]], [0, 0, 1]])
    d = [params[5], params[6]]
    offset = 7

    Rs = []
    ts = []
    N = int((len(params)-offset) / 6)
```

```
    for i in range(N):
        wx, wy, wd = params[i*6+offset], params[i*6+offset+1],
params[i*6+offset+2]
        phi = np.linalg.norm([wx, wy, wd])
        W = np.array([[0, -wd, wy], [wd, 0, -wx], [-wy, wx, 0]])
        R = np.identity(3) + (np.sin(phi)/phi)*W + (1-
np.cos(phi))/(phi**2)*W.dot(W)
        t = params[i*6+offset+3:i*6+offset+6]
        Rs.append(R)
        ts.append(t)

    return K, d, Rs, ts

def cost_func(params, corners, grid, dist):
    K, d, Rs, ts = restore_param(params)
    x0, y0 = K[0, 2], K[1, 2]

    grid = np.insert(grid, 2, 0, axis=1)
    grid = np.insert(grid, 3, 1, axis=1)

    errors = np.array([])
    for R, t, corner in zip(Rs, ts, corners):
        P = K.dot(np.concatenate((R, np.transpose([t])), axis=1))
        projected_points = P.dot(np.transpose(grid))
        projected_points =
np.transpose(projected_points/projected_points[2, :])
        projected_points = projected_points[:, 0:2]
        if dist:
            x, y = projected_points[:, 0], projected_points[:, 1]
            r2 = (x-x0)**2 + (y-y0)**2
            x = x + (x-x0)*(d[0]*r2+d[1]*r2**2)
            y = y + (y-y0)*(d[0]*r2+d[1]*r2**2)
            projected_points[:, 0], projected_points[:, 1] = x, y
        error = corner - projected_points
        errors = np.concatenate((errors, error.flatten()))
    return errors


def calibration(folder_dir, grid, out_dir):

    i = 0
    Hs = []
    corners = []
    inputs = []
    for img in os.listdir(folder_dir):
        # check if the image ends with jpg
        if (img.endswith(".jpg") or img.endswith(".jpeg")):
            image = cv2.imread(str(folder_dir / img))
            if(image is not None):
                corner, edges, lines_img, corners_img =
corner_detection(image)
                if len(corner) == 80:
                    H = calculate_homography(grid, corner)
                    corners.append(corner)
                    Hs.append(H)
                    inputs.append(img)
                    if i < 2:
```

```
                        cv2.imwrite(out_dir + "/edges_" + img, edges)
                        cv2.imwrite(out_dir + "/lines_" + img, lines_img)
                        cv2.imwrite(out_dir + "/corners_" + img,
corners_img)
                        i = i + 1
    omega = find_omega(Hs)
    K = find_k(omega)
    Rs, ts = find_extrinsic(Hs, K)

    return K, Rs, ts, Hs, corners, inputs

def non_linear_refine(K, Rs, ts, corners, grid, dist):
    params = construct_param(K, Rs, ts)
    params_nonlinear = optimize.least_squares(cost_func, params, args =
(corners, grid, dist), method = 'lm').x
    K, d, Rs, ts = restore_param(params_nonlinear)

    return K, d, Rs, ts


def reproject_to_fixed_image(data_path, inputs, fixed_img_name, img_name,
corners, K, d, Rs, ts):

    fixed_image = cv2.imread(str(data_path / fixed_img_name))

    fixed_img_id = inputs.index(fixed_img_name)
    img_id = inputs.index(img_name)

    ex_fixed = np.concatenate((Rs[fixed_img_id][:, 0:2],
np.transpose([ts[fixed_img_id]])), axis=1)
    P_fixed = np.dot(K, ex_fixed)

    ex = np.concatenate((Rs[img_id][:, 0:2], np.transpose([ts[img_id]])),
axis=1)
    P = np.dot(K, ex)

    H = P_fixed.dot(np.linalg.pinv(P))

    corner = np.insert(corners[img_id], 2, 1, axis=1)

    corner_fixed_proj = H.dot(np.transpose(corner))
    corner_fixed_proj =
np.transpose(corner_fixed_proj/corner_fixed_proj[2, :])[:, 0:2]

    x0, y0 = K[0, 2], K[1, 2]
    x, y = corner_fixed_proj[:, 0], corner_fixed_proj[:, 1]
    r2 = (x-x0)**2 + (y-y0)**2
    x = x + (x-x0)*(d[0]*r2+d[1]*r2**2)
    y = y + (y-y0)*(d[0]*r2+d[1]*r2**2)
    corner_fixed_proj[:, 0], corner_fixed_proj[:, 1] = x, y

    error = corners[fixed_img_id] - corner_fixed_proj
    norm = np.linalg.norm(error, axis=1)
    mean = statistics.mean(norm)
    variance = statistics.variance(norm)
```

```
    out_image = draw_corners(fixed_image, corners[fixed_img_id], (255,
255, 0))
    out_image = draw_corners(out_image, corner_fixed_proj.astype(int), (0,
0, 255))

    return mean, variance, out_image




# In[12]:


if __name__ == '__main__':
    path = Path("C:/Users/yhosc/Desktop/ECE661/HW8/HW8-Files")
    outputPath = Path("C:/Users/yhosc/Desktop/ECE661/HW8")

    grid = make_grid(10, 8, 25.5, 25.33)
    K0, Rs0, ts0, Hs0, corners, inputs = calibration(path / "Dataset1",
grid, "output1")
    print("K: ", K0)
    print("RT1: ", Rs0[0], ts0[0])
    print("RT2: ", Rs0[1], ts0[1])

    mean, variance, out_image = reproject_to_fixed_image(path /
"Dataset1", inputs, "Pic_28.jpg", "Pic_1.jpg", corners, K0, [0, 0], Rs0,
ts0)
    cv2.imwrite(str(outputPath / "output1" / "reproject1.jpg"), out_image)
    print(mean, variance)

    mean, variance, out_image = reproject_to_fixed_image(path /
"Dataset1", inputs, "Pic_28.jpg", "Pic_2.jpg", corners, K0, [0, 0], Rs0,
ts0)
    cv2.imwrite(str(outputPath / "output1" / "reproject2.jpg"), out_image)
    print(mean, variance)

    # nonlinear refine
    K1, d1, Rs1, ts1 = non_linear_refine(K0, Rs0, ts0, corners, grid,
False)
    print("K: ", K1)
    print("RT1: ", Rs1[0], ts1[0])
    print("RT2: ", Rs1[1], ts1[1])

    mean, variance, out_image = reproject_to_fixed_image(path /
"Dataset1", inputs, "Pic_28.jpg", "Pic_1.jpg", corners, K1, d1, Rs1, ts1)
    cv2.imwrite(str(outputPath / "output1" / "reproject1_lm.jpg"),
out_image)
    print(mean, variance)

    mean, variance, out_image = reproject_to_fixed_image(path /
"Dataset1", inputs, "Pic_28.jpg", "Pic_2.jpg", corners, K1, d1, Rs1, ts1)
    cv2.imwrite(str(outputPath / "output1" / "reproject2_lm.jpg"),
out_image)
    print(mean, variance)

    # nonlinear refine with radio distortion
```

```
    K2, d2, Rs2, ts2 = non_linear_refine(K0, Rs0, ts0, corners, grid,
True)
    print("K: ", K2)
    print("RT1: ", Rs2[0], ts2[0], d2)
    print("RT2: ", Rs2[1], ts2[1], d2)

    mean, variance, out_image = reproject_to_fixed_image(path /
"Dataset1", inputs, "Pic_28.jpg", "Pic_1.jpg", corners, K2, d2, Rs2, ts2)
    cv2.imwrite(str(outputPath / "output1" / "reproject1_rad.jpg"),
out_image)
    print(mean, variance)

    mean, variance, out_image = reproject_to_fixed_image(path /
"Dataset1", inputs, "Pic_28.jpg", "Pic_2.jpg", corners, K2, d2, Rs2, ts2)
    cv2.imwrite(str(outputPath / "output1" / "reproject2_rad.jpg"),
out_image)
    print(mean, variance)

    # My dataset
    K0, Rs0, ts0, Hs0, corners, inputs = calibration(path / "Dataset2",
grid, "output2")
    print("K: ", K0)
    print("RT1: ", Rs0[0], ts0[0])
    print("RT2: ", Rs0[1], ts0[1])

    mean, variance, out_image = reproject_to_fixed_image(path /
"Dataset2", inputs, "000 (1).jpg", "000 (2).jpg", corners, K0, [0, 0],
Rs0, ts0)
    cv2.imwrite(str(outputPath / "output2" / "reproject1.jpg"), out_image)
    print(mean, variance)

    mean, variance, out_image = reproject_to_fixed_image(path /
"Dataset2", inputs, "000 (1).jpg", "000 (3).jpg", corners, K0, [0, 0],
Rs0, ts0)
    cv2.imwrite(str(outputPath / "output2" / "reproject2.jpg"), out_image)
    print(mean, variance)

    # nonlinear refine
    K1, d1, Rs1, ts1 = non_linear_refine(K0, Rs0, ts0, corners, grid,
False)
    print("K: ", K1)
    print("RT1: ", Rs1[0], ts1[0])
    print("RT2: ", Rs1[1], ts1[1])

    mean, variance, out_image = reproject_to_fixed_image(path /
"Dataset2", inputs, "000 (1).jpg", "000 (2).jpg", corners, K1, d1, Rs1,
ts1)
    cv2.imwrite(str(outputPath / "output2" / "reproject1_lm.jpg"),
out_image)
    print(mean, variance)

    mean, variance, out_image = reproject_to_fixed_image(path /
"Dataset2", inputs, "000 (1).jpg", "000 (3).jpg", corners, K1, d1, Rs1,
ts1)
    cv2.imwrite(str(outputPath / "output2" / "reproject2_lm.jpg"),
out_image)
    print(mean, variance)
```

```
    # nonlinear refine with radio distortion
    K2, d2, Rs2, ts2 = non_linear_refine(K0, Rs0, ts0, corners, grid,
True)
    print("K: ", K2)
    print("RT1: ", Rs2[0], ts2[0], d2)
    print("RT2: ", Rs2[1], ts2[1], d2)

    mean, variance, out_image = reproject_to_fixed_image(path /
"Dataset2", inputs, "000 (1).jpg", "000 (2).jpg", corners, K2, d2, Rs2,
ts2)
    cv2.imwrite(str(outputPath / "output2" / "reproject1_rad.jpg"),
out_image)
    print(mean, variance)

    mean, variance, out_image = reproject_to_fixed_image(path /
"Dataset2", inputs, "000 (1).jpg", "000 (3).jpg", corners, K2, d2, Rs2,
ts2)
    cv2.imwrite(str(outputPath / "output2" / "reproject2_rad.jpg"),
out_image)
    print(mean, variance)
```