Assignment – 1 **2018900087**
Computer Vision **Akarshan Sajja**

Question 1: Direct Linear Transformation based camera calibration

Description:

- Camera calibration is to determine the parameters of the transformation between an object in 3D space and the 2D image observed by the camera.
- DLT is used to determine the orientation of a single image without knowing parameters of the intrinsic and extrinsic camera orientation a priori.
- Lets suppose a point in 3D world coordinate is, $\mathbf{X}=(X,Y,Z,1)$. Then the same corresponding 3D point in camera frame is $\mathbf{X}_{cam.}$

$$\mathbf{X}_{cam} = [\mathbf{R} \mid \mathbf{t}]\, \mathbf{X}$$

  Where, $\mathbf{R}$ is a Rotation Matrix 3x3 and $\mathbf{t}$ is a Translation Matrix 3x1
- Now, let $\mathbf{x}=(x,y,1)$ be the image coordinate of that 3D point, while converting 3D to 2D becomes

$$\mathbf{x} = \mathbf{K}\,[\mathbf{R} \mid \mathbf{t}]\, \mathbf{X}$$

- In above, $\mathbf{K}$ is a 3x3 camera matrix contains the information intrinsic parameters of the camera. The $\mathbf{R}$ and $\mathbf{t}$ matrices constitutes the extrinsic parameters of the camera.
- The orientation of a single image has total 11 unknowns, in that 5 are intrinsic parameters and 6 are extrinsic parameters. To find the 11 unknowns at least 12 equations is required. So here we are considering the 6 points for each 2 equations. Finding out these unknown parameters is known as camera calibration.

Algorithm:

- The defined function **findImagePoints()** will be used to find out the image coordinates in 2D plane. To find out these points used the **HoughCircles** openCv inbuilt function. It finds circles in a gray scale image using the Hough transform. Note: Here, I attached the code to find image points, but I used the manually taken image point coordinates.
- We need to find the camera matrix and Rotation, Translation vectors.
- To find the 11 unknowns, here considering 6 points means 12 equation. For 12 equations will get 12 vectors, inserted into a multidimensional numpy array.

- By above equations, got the 12x12 matrix. Passing this matrix to the Singular Value Decomposition(**SVD**). It returns the **U, s, V^T** unitary arrays. Will take the last column of the V^T and reshape into 3x4 matrix. This **P** where *i.e.,* x = **P**X
- Now, will pass this P through **np.linalg.qr(P)** to compute the qr factorization of a matrix. And it returns **K, R**.
- And the final step is to find the translation vector. $\mathbf{t} = \mathbf{K}^{-1}\mathbf{h}$. **h** is the last column of the R matrix.

Code:

```
import numpy as np
import cv2 as cv
from scipy import linalg
from numpy.linalg import inv

def findImagePoints():
    img = cv2.imread('img.jpg',0)
    img = cv2.medianBlur(img,5)
    cimg = cv2.cvtColor(img,cv2.COLOR_GRAY2BGR)

     circles = cv2.HoughCircles( img, cv2.HOUGH_GRADIENT, 1, 20, param1=100,
         param2=30, minRadius=10, maxRadius=80)
    circles = np.uint16(np.around(circles))
    imgp = []
    for i in circles[0,:]:
        cv2.circle(cimg,(i[0],i[1]),i[2],(0,255,0),2)
        imgp.append(i[0],i[1])

    imgp = np.array(imgp)
    return imgp

def DLT(objp, imgp):

    mat = np.zeros((12,12), np.int)
    for i in range(0,12):
        if(i%2==0):
            mat[i][0:3] = np.multiply(objp[int(i/2)][0:3], -1)
            mat[i][3] = -1
            x = imgp[int(i/2)][0]
            mat[i][8:11] = np.multiply(objp[int(i/2)][0:3],x)
            mat[i][11] = x
        else:
            mat[i][4:7] = np.multiply(objp[int(i/2)][0:3], -1)
            mat[i][7] = -1
            y = imgp[int(i/2)][1]
            mat[i][8:11] = np.multiply(objp[int(i/2)][0:3],y)
```

```
        mat[i][11] = y

    U, s, V_h = linalg.svd(mat)
    proj = np.reshape(V_h[-1], (3, 4))

    K, R = np.linalg.qr(proj)
    h = R.T[-1]
    K_inv = inv(K)
    t = np.matmul(K_inv, h)
    return K, R, ts

imgp = findImagePoints()

objp = np.array([[36,36,0,1], [0,36,0,1],[0,72,0,1],[36,0,36,1], [0,0,36,1], [0,0,72,1]])
imgp = np.array([[4037,1356], [4886,1422], [4931, 583], [3883, 2404], [4743, 2481], [4655,
2812]])

K, R, t = DLT(objp,imgp)
```

Question 2: RANSAC based variant of the calibration

Description:

- Random sample consensus (RANSAC) is an iterative technique for camera calibration. It takes set of sample points, fit those points to model.
- RANSAC assumption is that the data consists of "**inliers**", i.e., data whose distribution can be explained by some set of model parameters, though may be subject to noise, and "**outliers**" which are data that do not fit the model.

Algorithm:

- Select a six(>=6) random points of the original data. Assuming these points are hypothetical inliers.
- Here, Considering the same kind of DLT model is fitted to the initially taken set of inliers.
- Except inliers, remaining all other data are then tested against the fitted model. According to some model-specific loss function, calculated the error for tested data. And set the threshold distance 10.0. The points error which less than the threshold considered. Find out the error and projection matrix for this points.

- This procedure is repeated a fixed number of times, each time producing error and projection matrix. From all of these considering the projection matrix which one error is less.
- Now, will pass this projection matrix through **np.linalg.qr(P)** to compute the qr factorization of a matrix. And it returns **K, R**.
- And the final step is to find the translation vector. $\mathbf{t = K^{-1}h}$. **h** is the last column of the R matrix.

Code:

```
import numpy as np
import random
import cv2 as cv
from scipy import linalg
from numpy.linalg import inv

def check(maybeInliers,objp):
    return any(np.array_equal(x, maybeInliers) for x in objp)

def RANSAC(objp, imgp):

    mat = np.zeros((2*objp.shape[0],12), np.int)

    for i in range(2*objp.shape[0]):
        if(i%2==0):
            mat[i][0:3] = np.multiply(objp[int(i/2)][0:3], -1)
            mat[i][3] = -1
            x = imgp[int(i/2)][0]
            mat[i][8:11] = np.multiply(objp[int(i/2)][0:3],x)
            mat[i][11] = x
        else:
            mat[i][4:7] = np.multiply(objp[int(i/2)][0:3], -1)
            mat[i][7] = -1
            y = imgp[int(i/2)][1]
            mat[i][8:11] = np.multiply(objp[int(i/2)][0:3],y)
            mat[i][11] = y

    U, s, V_h = linalg.svd(mat)
    proj = np.reshape(V_h[-1], (3, 4))
    return proj


def ErrorCalc(proj, objp), imgp:
    inliers = []
    inliers1 = []
    Error = []
```

```
    for i in range(36):
        verify = np.matmul(proj, objp[i])
        verify = [(verify[0]/verify[2]), (verify[1]/verify[2])]
        Error.append(np.average(np.array(imgp[i]) - verify))
        if((np.average(np.array(imgp[i]) - verify)) <= 10.0):
            inliers.append(imgp[i])
            inliers1.append(objp[i])
    return(Error, inliers, inliers1)

objp = np.array([[36,36,0,1], [0,36,0,1],[0,72,0,1],[36,0,36,1], [0,0,36,1], [0,0,72,1],
[216,72,0,1], [180,72,0,1], [144,72,0,1], [108,72,0,1], [72,72,0,1], [36,72,0,1], [216,36,0,1],
[180,36,0,1], [144,36,0,1], [108,36,0,1], [72,36,0,1], [180,0,36,1], [144,0,36,1], [108,0,36,1],
[72,0,36,1], [180,0,72,1], [144,0,72,1], [108,0,72,1], [72,0,72,1], [36,0,72,1], [144,0,108,1],
[108,0,108,1], [72,0,108,1], [36,0,108,1], [0,0,108,1], [144,0,144,1], [108,0,144,1],
[72,0,144,1], [36,0,144,1], [0,0,144,1]])
imgp = np.array([[4037,1356], [4886,1422], [4931, 583], [3883, 2404], [4743, 2481], [4655,
2812], [143,308], [904, 352], [1665, 396], [2438,451], [3254,495], [4070,550], [198,1080],
[948,1135], [1687, 1190], [2448, 1245], [3232,1301], [672, 2128], [1445, 2194], [2239,
2249], [3044, 2338], [342, 2415], [1158, 2492], [1996, 2569], [2857, 2658], [3739, 2735],
[838, 2845], [1721, 2983], [2636, 3022], [3574, 3110], [4556, 3198], [474, 3242], [1423,
3342], [2394, 3430], [3397, 3518], [4446, 3628]])

max1 = 9999999

for i in range(3):
    imgp = imgp.astype(float)
    pairs = list(zip(imgp, objp))
    pairs = random.sample(pairs, 6)
    A1, B1 = zip(*pairs)
    imgp_temp=list(A1)
    objp_temp=list(B1)
    objp_temp = np.array(objp_temp)
    imgp_temp = np.array(imgp_temp)
    print(objp_temp, imgp_temp)
    proj_temp = RANSAC(objp_temp, imgp_temp)
    Error, inliers, inliers1 = ErrorCalc(proj_temp, objp, imgp)
    if(len(inliers)>=6):
        proj_temp = RANSAC(objp_temp, imgp_temp)
        Error, inliers, inliers1 = ErrorCalc(proj_temp, objp, imgp)
        if(np.average(Error)< max1):
            print(np.average(Error))
            proj = proj_temp

K, R = np.linalg.qr(proj)
h = R.T[-1]
K_inv = inv(K)
t = np.matmul(K_inv, h)
print(K, R, t)
```

Question 6:

Description:

- Zhang's camera calibration one variant of Multiplane calibration. It means will find out the camera parameters by using two or more views of planar surface.
- Zhang's method calibrates cameras by solving a particular system of homogeneous linear equations that captures the homo-graphic relationships between multiple perspective views of the same plane.

Algorithm:

- Zhang's calibration need at least 10 test patterns. Here onsidering the checkerboard images in multiple views of planar surface. This chess board has 9x7 squares and 8x6 internal corners.
- Next step is to find out the 3D world coordinates (**X,Y,Z)** and 2D image coordinates (**x, y)**. For simplicity, chess board was kept stationary at XY plane, (so Z=0 always), (**X,Y,0)**.
- We have object points (3D world points), need to find image points (2D image points). For that using the openCV function "**cv.findChessboardCorners()"**. This function returns the image points.
- Now we have object points and image points. Next step is to calibrate the camera. Passing the image and object points to the function **"cv.calibrate Camera()",** which returns camera matrix, distortion coefficients, rotation and translation vectors.

Code:

```
import numpy as np
import cv2 as cv
import glob

criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)

objp = np.zeros((9*6, 3), np.float32)
objp[:,:2] = np.mgrid[0:9,0:6].T.reshape(-1,2)

objpoints = []
imgpoints = []


images = glob.glob('*.jpg')
```

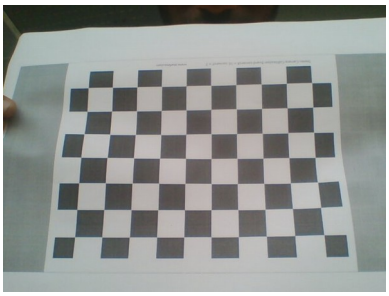
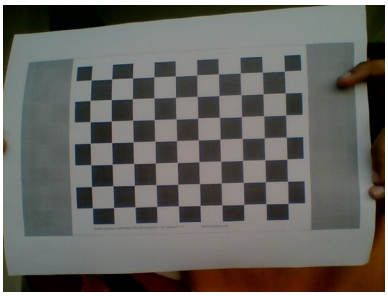
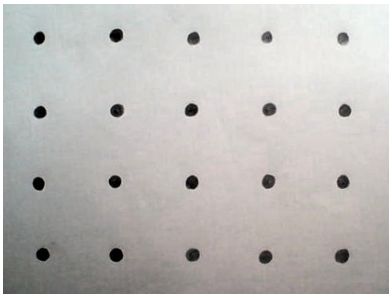
```
for fname in images:
    img = cv.imread(fname)
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

    ret, corners = cv.findChessboardCorners(gray, (9,6), None)
    print(corners)

    if ret == True:
        objpoints.append(objp)
        imgpoints.append(corners)

ret, mtx, dist, rvecs, tvecs = cv.calibrateCamera(objpoints, imgpoints, gray.shape[::-1],
None, None)

cv.destroyAllWindows()
```

Observation:

- Zhang's multiview approach is accurate, and it is more natural to capture multiple views of a single planar surface - like a checkerboard . But in Direct Linear camera calibration need to construct a precise 3D calibration rig.

Question 9:

Camera sample images:

Manually computed points:

Image_points= [[40,40], [123, 37], [205, 40], [286, 38], [367, 41], [41, 123], [123, 121], [204, 121], [287, 120], [369, 122], [39, 204], [121, 202], [204, 203], [287, 201], [367, 201], [44, 284], [122, 285], [205, 283], [284, 286]]

Object_Points = [[140,108,0,1], [108,108,0,1], [72,108,0,1], [36,108,0,1], [0,108,0,1], [140,72,0,1], [108,72,0,1], [72,72,0,1], [36,72,0,1], [0,72,0,1], [140,36,0,1], [108,36,0,1], [72, 36, 0,1], [36,36,0,1], [0,36,0,1],
[140,0,0,1], [108,0,0,1], [72,0,0,1], [36,0,0,1], [0,0,0,1]]

Question 10:

- In all there camera calibration algorithms RANSAC computing time is more.

- DLT calibration results. Img1: Given image results and Img2: Own camera results

```
Camera Matrix:  [[-9.97886199e-01  6.49856396e-02  1.51905692e-05]
 [-6.49856403e-02 -9.97886198e-01 -5.00586872e-05]
 [ 1.19053635e-05 -5.09400419e-05  9.99999999e-01]]
Rotation Matrix:  [[-4.05882026e-03 -1.02853829e-03 -1.97952578e-03  9.35480110e-01]
 [ 0.00000000e+00 -4.27674282e-03  8.06981252e-04  3.53322145e-01]
 [ 0.00000000e+00  0.00000000e+00  3.99216491e-07 -1.81572303e-04]]
Translation Vector: [-9.56463560e-01 -2.91782509e-01 -1.85048670e-04]
```

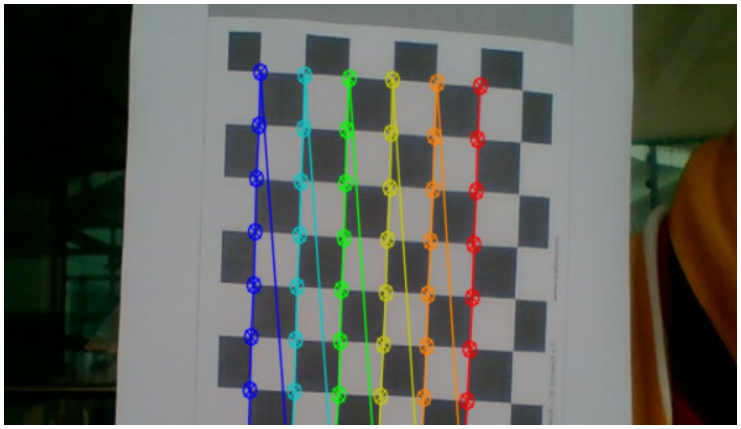
```
(cv) sunil@sunil-inspiron-5559:~/Desktop$ python DLT.py
Camera Matrix:  [[ 0.          0.99894827  0.04585151]
 [-0.99972553  0.0010742  -0.02340314]
 [-0.02342778 -0.04583893  0.99867409]]
Rotation Matrix:  [[ 2.05517774e-32  1.63818834e-18  2.34277783e-02 -1.76924341e-16]
 [ 0.00000000e+00 -5.46683618e-19  4.58389263e-02  5.90418307e-17]
 [ 0.00000000e+00  0.00000000e+00 -9.98674087e-01  6.93334780e-32]]
Translation Vector: [-5.90256256e-17 -1.76674841e-16 -9.49401252e-18]
```

- RANSAC calibration results. Img1: Given image results and Img2: Own camera results

```
[[-9.98252912e-01  5.90857351e-02  1.28135208e-05]
 [-5.90857349e-02 -9.98252912e-01  1.79504815e-05]
 [ 1.38517518e-05  1.71620241e-05  1.00000000e+00]] [[ 4.22169283e-03 -4.3827924
3e-05  1.26722209e-03 -9.41743618e-01]
 [ 0.00000000e+00  3.84339285e-03 -1.54614254e-03 -3.36277153e-01]
 [ 0.00000000e+00  0.00000000e+00 -1.30401164e-07  2.05617699e-04]] [9.59967494e
-01 2.80046037e-01 1.87514311e-04]
```

- Zhang calibration results. Img1: Given image results and Img2: Own camera results

```
Camera Matrix:  [[456.26680347   0.         346.87219504]
 [  0.         455.16196624 285.06341961]
 [  0.           0.           1.        ]]
Rotation Matrix:  [array([[-0.01876042],
       [-0.01789012],
       [ 1.61601875]]), array([[-0.02121749],
       [-0.38539686],
       [-3.12263703]]), array([[-0.01876042],
       [-0.01789012],
       [ 1.61601875]])]
Translation Vector: [array([[ 2.81918346],
       [-4.87602467],
       [13.87645048]]), array([[4.03754889],
       [2.22159561],
       [9.85987883]]), array([[ 2.81918346],
       [-4.87602467],
       [13.87645048]])]
```



```
Camera Matrix:  [[1.36634771e+04 0.00000000e+00 3.33653576e+03]
 [0.00000000e+00 1.36813826e+04 1.49660066e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Rotation Matrix:  [array([[-0.01318089],
       [-0.05189252],
       [-0.00310183]]), array([[-0.4414999 ],
```