Question 1:

First, we will load our drive and import the necessary imports:

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

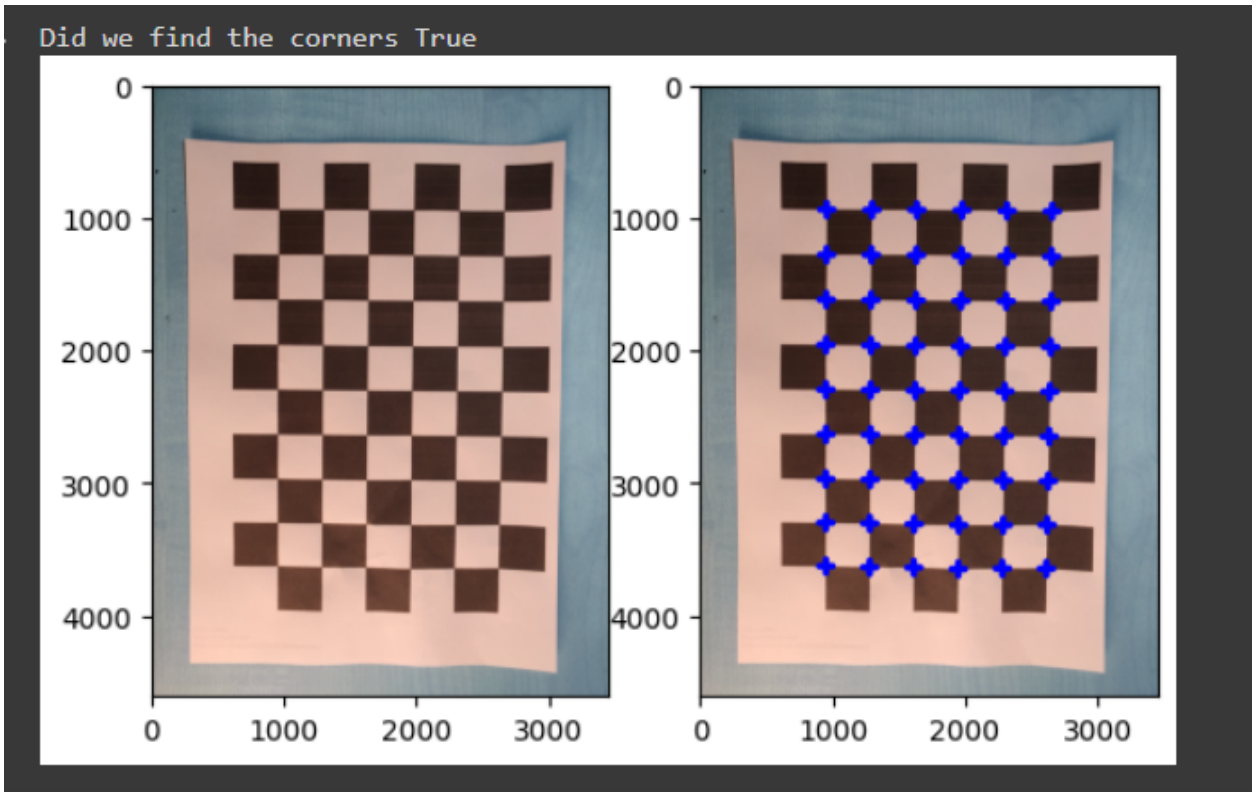
```
[2] import cv2 as cv
    import numpy as np
    import matplotlib.pyplot as plt
    %matplotlib inline
```

Then we will import our photos from the drive using.

```
import numpy as np

all_images = []
import os
for dirname, _, filenames in os.walk('/content/drive/MyDrive/Chessboard images/new_chessboard/New chessboard/'):
    for filename in filenames:

        all_images.append(os.path.join(dirname, filename))
```

Then we will take our first image and see if we can plot the image corners:

```
# First we will run the code on 1 image to see if we can find the corners in the image

img = cv.imread("/content/drive/MyDrive/Chessboard images/new_chessboard/New chessboard/IMG_20230406_160516.jpg")
plt.subplot(1, 2, 1)
plt.imshow(img)
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 5)

ret, corners = cv.findChessboardCorners(gray, (6, 9), cv.CALIB_CB_ADAPTIVE_THRESH + cv.CALIB_CB_FAST_CHECK + cv.CALIB_CB_NORMALIZE_IMAGE)

print("Did we find the corners", ret)
corners2 = cv.cornerSubPix(gray, corners, (11,11),(-1,-1), criteria)
# img = cv.drawChessboardCorners(img, (6, 9), corners2, ret)
marker_size = 100
marker_thickness = 50
for corner in corners2:
    x, y = corner.ravel()
    cv.drawMarker(img, (int(x), int(y)), (0, 0, 255), cv.MARKER_CROSS, marker_size, marker_thickness)
plt.subplot(1, 2, 2)
plt.imshow(img)

plt.show()
```



But not all images we take are going to be perfect. Some images will have some defects. We have to remove these images:

```
#This code is to check if there are some images that dont work. In case the sum of images that dont work is less than 25 we have to take more.
index = 0
for image in all_images:
  print(index)
  index+=1
  print(image)
  img = cv.imread(image)
  gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
  ret, corners = cv.findChessboardCorners(gray, (9, 6), cv.CALIB_CB_ADAPTIVE_THRESH + cv.CALIB_CB_FAST_CHECK + cv.CALIB_CB_NORMALIZE_IMAGE)
  print(ret)
```

So using this function, we will get the image name and “TRUE” or “FALSE” printed with it.
If the value is false, we can't find corners, and we must leave this image. In that case, I deleted the image from the dataset.

Next, we do the camera calibration. For that we will first try to find corners of the chessboard, and then we will use them to calibrate the camera.

```
# Now we will do camera calibration on rest of the images
CHECKERBOARD = (9,6)
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)

objpoints = []
imgpoints = []


# Defining the world coordinates for 3D points
objp = np.zeros((1, CHECKERBOARD[0] * CHECKERBOARD[1], 3), np.float32)
objp[0,:,:2] = np.mgrid[0:CHECKERBOARD[0], 0:CHECKERBOARD[1]].T.reshape(-1, 2)

for fname in all_images:
    print(fname)
    img = cv.imread(fname)
    plt.imshow(img)
    gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
    ret, corners = cv.findChessboardCorners(gray, CHECKERBOARD, cv.CALIB_CB_ADAPTIVE_THRESH + cv.CALIB_CB_FAST_CHECK + cv.CALIB_CB_NORMALIZE_IMAGE)

    objpoints.append(objp)
    corners2 = cv.cornerSubPix(gray, corners, (11,11),(-1,-1), criteria)

    imgpoints.append(corners2)


h,w = img.shape[:2]
```

```
h,w = img.shape[:2]

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

print("Camera matrix : \n")
print(mtx)
print("dist : \n")
print(dist)
print("rvecs : \n")
print(rvecs)
print("tvecs : \n")
print(tvecs)
```

Using this, we will get th intrinsic camera matrix, along with the rotational vectors 3x1 and translational vectors 3x1 for all the images. These can then be used for reprojection.

```
Camera matrix :

[[3.65247123e+03 0.00000000e+00 2.21289370e+03]
 [0.00000000e+00 3.66268242e+03 1.84409788e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
```

Using the documentation:

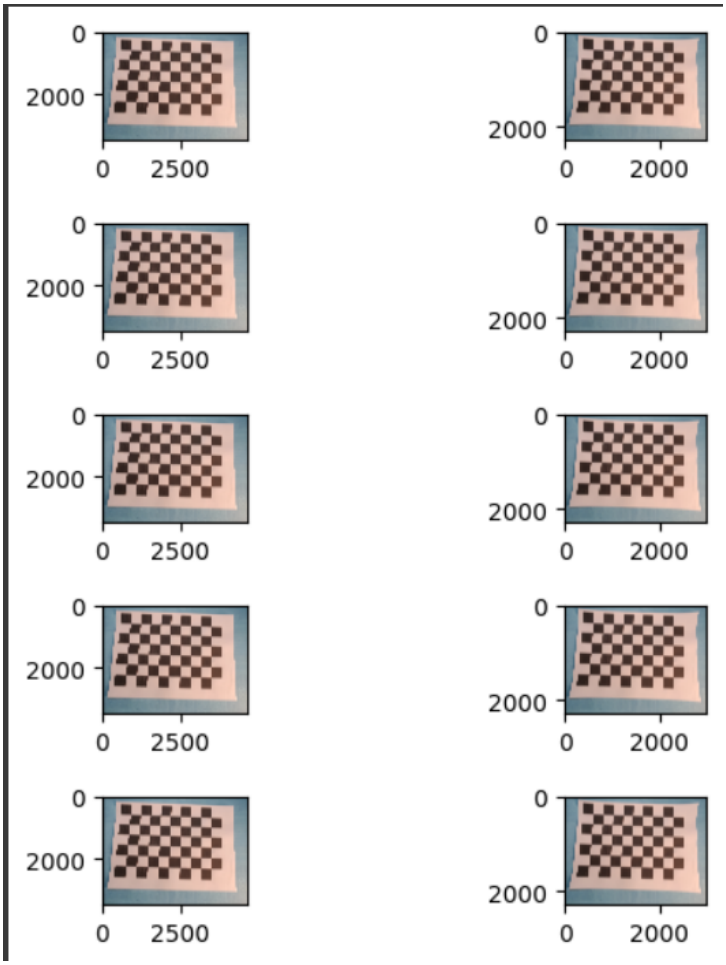
cameraMatrix Input/output 3x3 floating-point camera intrinsic matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$. If **CALIB_USE_INTRINSIC_GUESS** and/or **CALIB_FIX_ASPECT_RATIO**, **CALIB_FIX_PRINCIPAL_POINT** or **CALIB_FIX_FOCAL_LENGTH** are specified, some or all of fx, fy, cx, cy must be initialized before calling the function.

For this we can see that the **skew parameters is 0 focal length are fx = 3652, fy = 3662, principal points are cx = 2212 and cy = 1844.** , and other parameters like

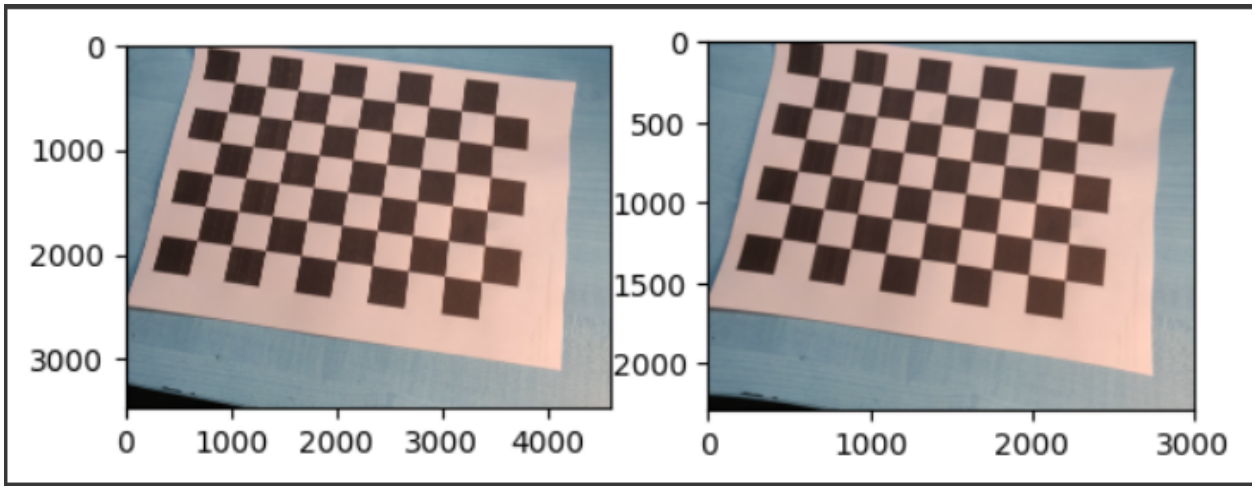
The next task is to undistort the image. For this, we will call the inbuilt function in opencv library: cv.undistort(img, mtx, dist, None, newcameramtx).

After using this we can see that the corners get wider.

```
for i in range(5):
  img = cv.imread(all_images[0])
  plt.subplot(5, 2, 2*i + 1)
  plt.imshow(img)
  h,  w = img.shape[:2]
  # print("")
  newcameramtx, roi = cv.getOptimalNewCameraMatrix(mtx, dist, (w,h), 1, (w,h))
  # undistort
  dst = cv.undistort(img, mtx, dist, None, newcameramtx)
  # crop the image
  x, y, w, h = roi
  dst = dst[y:y+h, x:x+w]
  plt.subplot(5, 2, 2*i + 2)
  plt.imshow(dst)
  plt.show()
```



Lets see at a closeup range:



The left one is the original image and the right one is the undistorted image. We can see the corner points are wider there. This due to the radial distortion. We see bulging due to magnification at the corners because of radial distortion.
This can be understood as:
X_new = x * (1 + k1*r*r + k2*r*r*r*r …)
Y_new = y * (1 + k1*r*r + k2*r*r*r*r …)
So the x will be multiplied by some positive (or negative number causing it to collapse in itself) number that you can see.

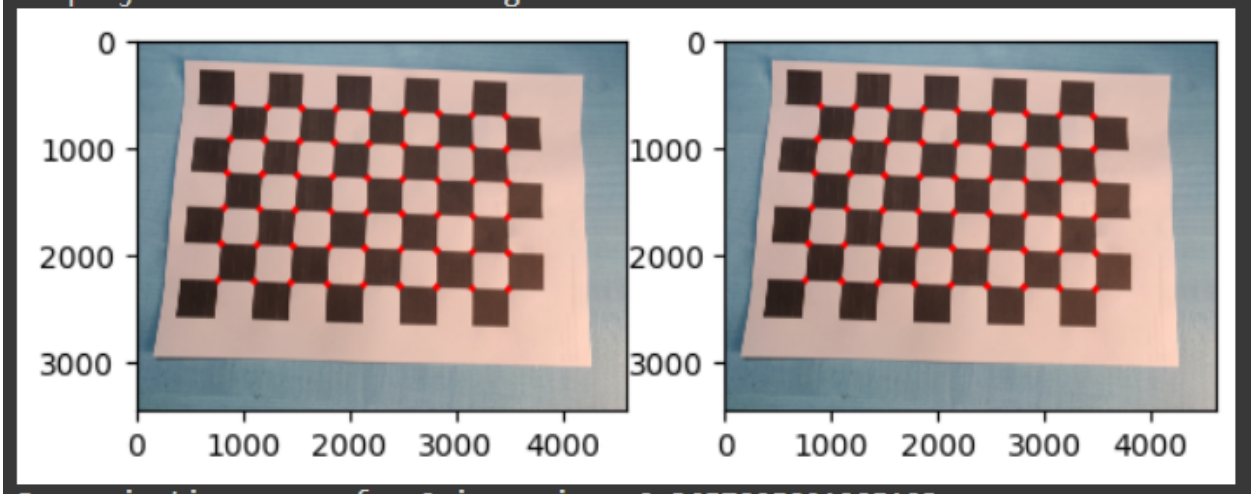
distCoeffs Input/output vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6[, s_1, s_2, s_3, s_4[, \tau_x, \tau_y]]]])$ of 4, 5, 8, 12 or 14 elements.

From this, we can conclude that the distortion coefficient is:
K1 = -0.04215362 K2 = 0.00633794

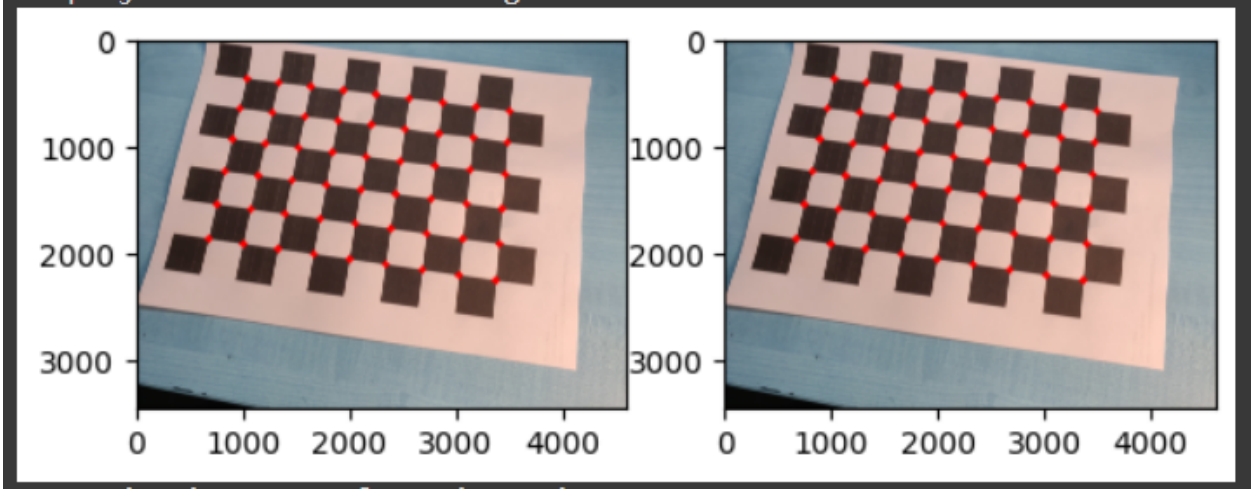
Now we will reproject.
For reprojection, the idea is to use the dist, translation vector and rotation vector to compute the 3-D coordinates of the point. Then we use these points to compute the 2-D coordinates of the corner points. Then we will use these corner points along with our original ones and check the difference using L2 norm. We will also project these points on the original image to see how closely they match.

```python
for i in range(5):
  imgpoints, _ = cv.projectPoints(objp, rvecs[i], tvecs[i], mtx, dist[0])
  # imgpoints = np.squeeze(imgpoints)
  img = cv.imread(all_images[i])
  gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
  ret, corners = cv.findChessboardCorners(gray, (9, 6), cv.CALIB_CB_ADAPTIVE_THRESH + cv.CALIB_CB_FAST_CHECK + cv.CALIB_CB_NORMALIZE_IMAGE)
  # Compute the re-projection error
  corners_float = corners.astype(np.float32)
  err = cv.norm(corners_float, imgpoints, cv.NORM_L2) / len(corners)
  print(f'Re-projection error for {i+1} image is : {err}')
  corners2 = cv.cornerSubPix(gray, corners, (11,11),(-1,-1), criteria)
  img_original = img
  img_projected = img

  marker_size = 50
  marker_thickness = 30
  for corner in corners:
      x, y = corner.ravel()
      cv.drawMarker(img_original, (int(x), int(y)), (255, 0, 0), cv.MARKER_CROSS, marker_size, marker_thickness)
  for corner in corners2:
      x, y = corner.ravel()
      cv.drawMarker(img_projected, (int(x), int(y)), (255, 0, 0), cv.MARKER_CROSS, marker_size, marker_thickness)
  plt.subplot(1, 2, 1)
  plt.imshow(img_original)
  plt.subplot(1, 2, 2)
  plt.imshow(img_projected)

  plt.show()
```

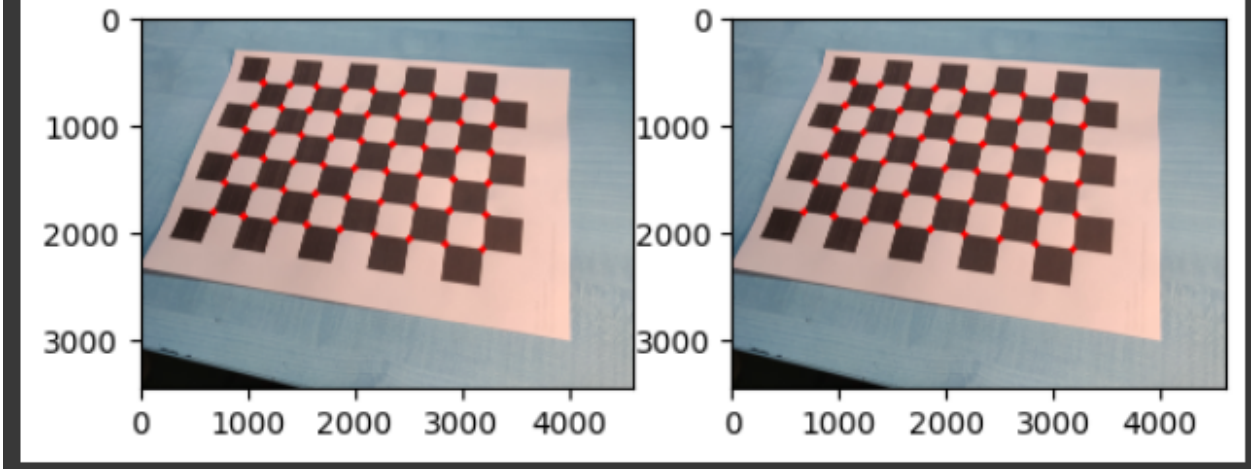
```
Re-projection error for 1 image is : 0.40966762410979995
```



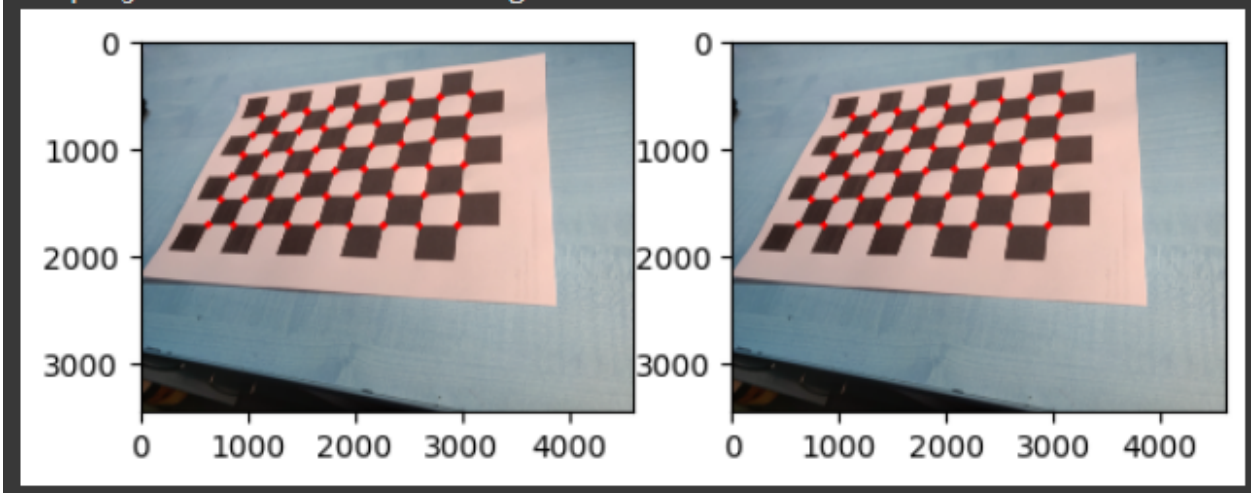

```
Re-projection error for 2 image is : 0.3657805891285123
```



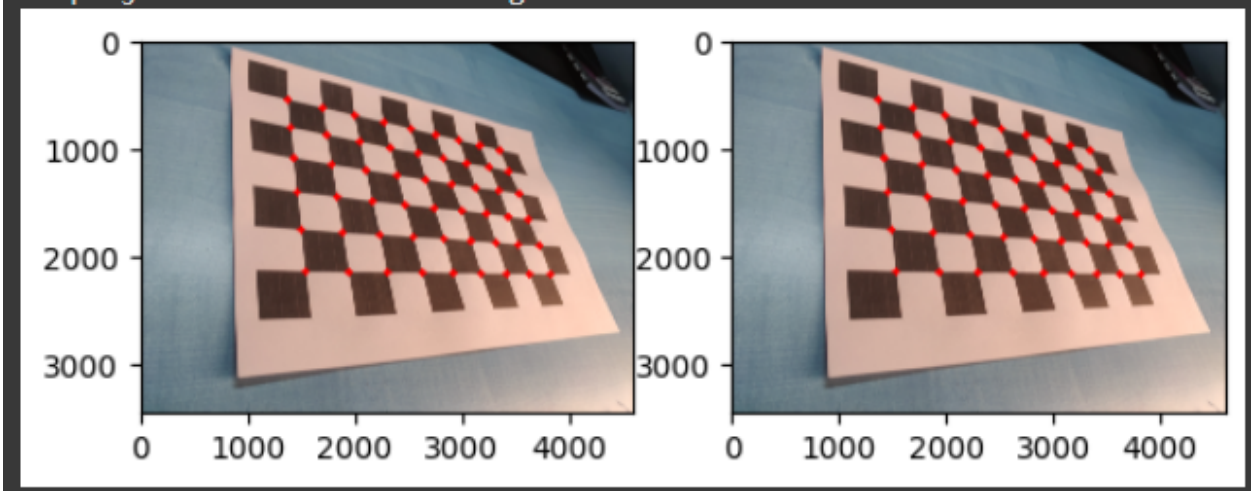
```
Re-projection error for 3 image is : 0.5095678327732708
```




```
Re-projection error for 4 image is : 0.7298770554725228
```



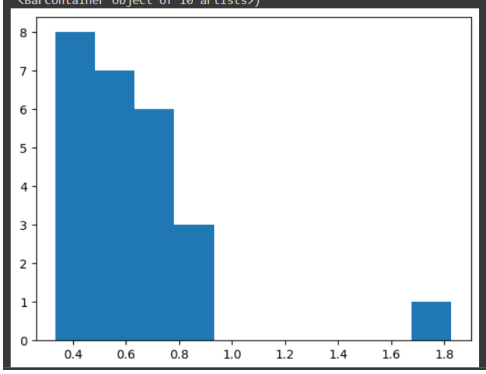

```
Re-projection error for 5 image is : 0.8183539389047119
```



So taking the error of first 25 images we get the error matrix is:

```
] error = []
  for i in range(25):
    imgpoints, _ = cv.projectPoints(objp, rvecs[i], tvecs[i], mtx, dist[0])
    # imgpoints = np.squeeze(imgpoints)
    img = cv.imread(all_images[i])
    gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
    ret, corners = cv.findChessboardCorners(gray, (9, 6), cv.CALIB_CB_ADAPTIVE_THRESH + cv.CALIB_CB_FAST_CHECK + cv.CALIB_CB_NORMALIZE_IMAGE)
    # Compute the re-projection error
    corners_float = corners.astype(np.float32)
    err = cv.norm(corners_float, imgpoints, cv.NORM_L2) / len(imgpoints)
    error.append(err)
```

```
[0.40966762410979995, 0.3657805891285123, 0.5095678327732708, 0.7298770554725228, 0.8183539389047119, 0.5585518232545768, 0.6821291140563579, 0.44336182711439465, 0.4842453067109595, 0
(array([8., 7., 6., 3., 0., 0., 0., 0., 0., 1.]),
 array([0.33494711, 0.48410298, 0.63325884, 0.78241471, 0.93157058,
        1.08072644, 1.22988231, 1.37903818, 1.52819404, 1.67734991,
        1.82650578]),
 <BarContainer object of 10 artists>)
```




```
[18] np.std(error)

0s

0.28685382227615513
```

Reprojection error is calculate using the euclidean distance between the 2 points.
Point 1 we got from using the findchessboard pattern and the other we got is by applying the euclidean transformation (translation and rotation) to a 3D image that we created.

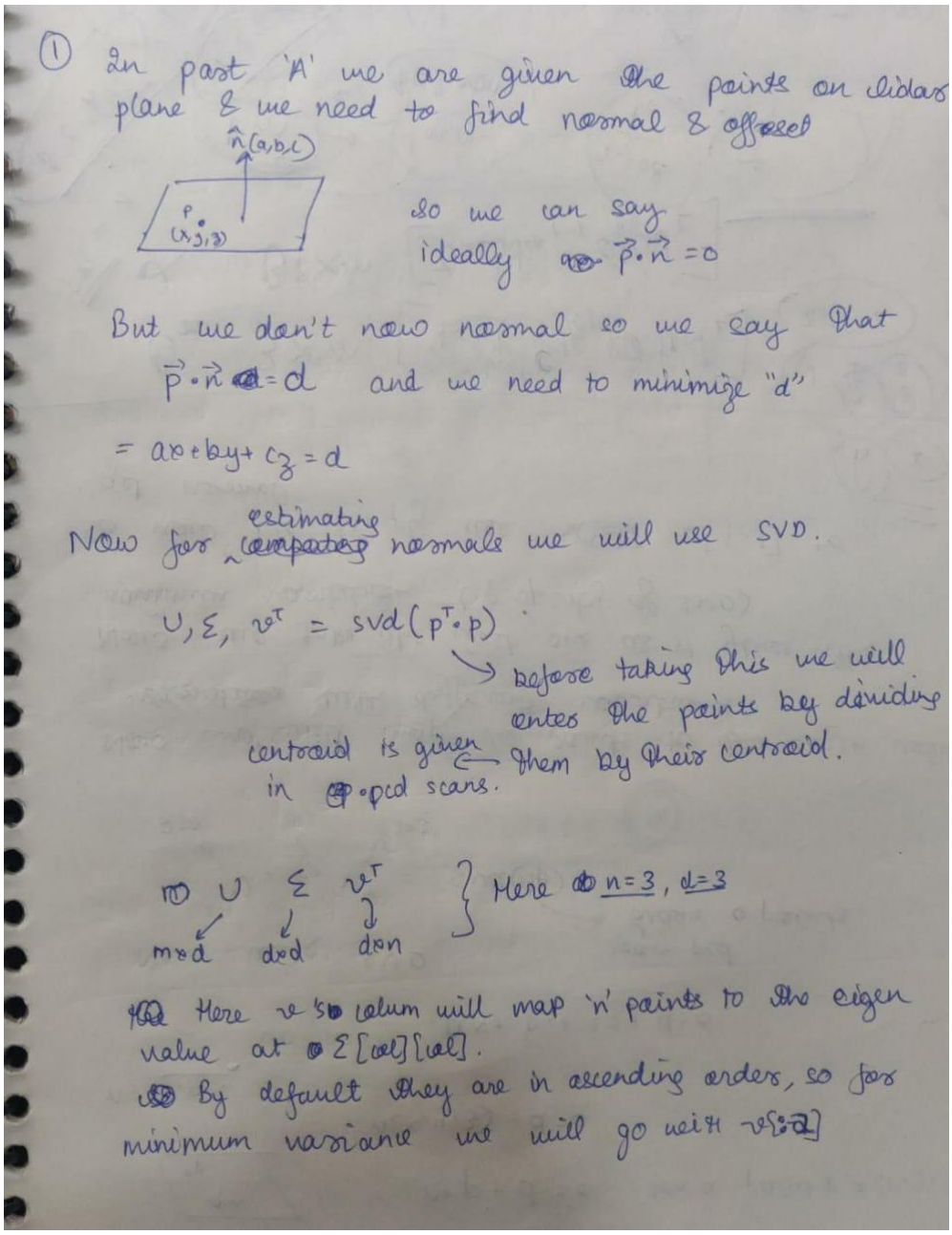
```python
for i in range(5):
  rotation_matrix = cv.Rodrigues(rvecs[i])[0]
  extrinsic_matrix = np.zeros((4,4))
  extrinsic_matrix[:3, :3] = rotation_matrix
  extrinsic_matrix[3, :3] = tvecs[i].reshape(3)
  extrinsic_matrix[3][3] = 1

  normal_matrix = np.linalg.inv(extrinsic_matrix).T
  camera_normal = np.dot(normal_matrix, np.array([0,0,1,1]).reshape(4,1))
  print(f"The camera normals for {i + 1}th image are: \n  {(camera_normal / cv.norm(camera_normal))[:3]}")
```

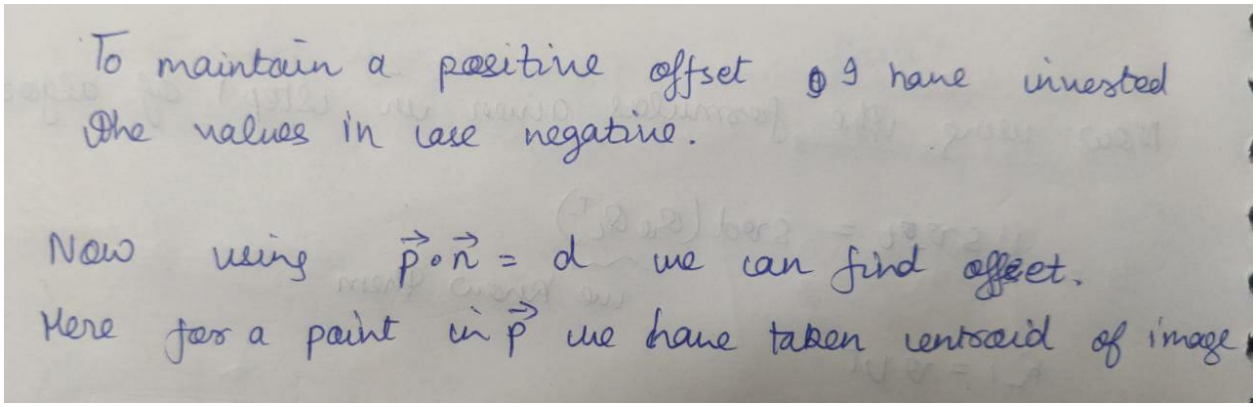
```
The camera normals for 1th image are:
  [[ 0.33806276]
 [ 0.21505585]
 [-0.9122757 ]]
The camera normals for 2th image are:
  [[ 0.22066743]
 [ 0.23903604]
 [-0.94221577]]
The camera normals for 3th image are:
  [[ 0.11168634]
 [-0.03047521]
 [-0.99078929]]
The camera normals for 4th image are:
  [[-0.05265201]
 [-0.22134292]
 [-0.97165988]]
The camera normals for 5th image are:
  [[ 0.672322  ]
 [-0.10791926]
 [-0.72623813]]
```

Question 2:

In this question, we are given information for the camera parameter and LIDAR .pcd scans along with the original image. The calibration of the scans is already done, and our task is to cross-calibrate the image.

So in the first case, we need to find the LIDAR normal of the image using the .pcd file.






```python
import numpy as np
normals = []
offsets = []
# Load PCD file into a NumPy array

for lidar_scan in all_frame_names:
    cloud = PyntCloud.from_file(f"/kaggle/input/lidar-dataset/lidar_scans/{lidar_scan}.pcd")
#     print(lidar_scan)
    pcd = np.array(cloud.points[["x", "y", "z"]])
    pcd = pcd - cloud.centroid

    u, sigma, v_t = np.linalg.svd(pcd.T @ pcd, full_matrices = False)
    normal = u[:, 2]

    offset = np.dot(normal, cloud.centroid)
    if offset < 0:
        offset = -offset
        normal = -normal


    normals.append(normal.reshape(3))
    offsets.append(offset)

    print("Chessboard plane normal vector:", normal)
    print("Chessboard plane offset:", offset)
```

```
Chessboard plane normal vector: [ 0.93223065 -0.19842705 -0.3026098 ]
Chessboard plane offset: 6.265358
Chessboard plane normal vector: [ 0.9328821  -0.24403861  0.26490793]
Chessboard plane offset: 5.6742587
Chessboard plane normal vector: [ 0.87718356 -0.41582555  0.24007946]
Chessboard plane offset: 7.435547
Chessboard plane normal vector: [ 0.76477665 -0.64422625 -0.0094462 ]
Chessboard plane offset: 6.9878364
Chessboard plane normal vector: [0.9415104  0.18949716 0.27865583]
Chessboard plane offset: 6.323572
Chessboard plane normal vector: [ 0.8187873  0.5076373 -0.2681264]
Chessboard plane offset: 6.4454384
Chessboard plane normal vector: [ 0.4734635  -0.75992006 -0.44536924]
Chessboard plane offset: 5.2549844
Chessboard plane normal vector: [ 0.60407287 -0.774129    0.18926245]
Chessboard plane offset: 5.0623527
Chessboard plane normal vector: [ 0.6187274  -0.7829378   0.06469016]
Chessboard plane offset: 6.589564
Chessboard plane normal vector: [ 0.40409026 -0.88184714  0.24301586]
Chessboard plane offset: 5.436721
Chessboard plane normal vector: [ 0.9493562   0.22833396 -0.21583882]
Chessboard plane offset: 7.271752
Chessboard plane normal vector: [ 0.6369344  -0.7649984   0.09535237]
Chessboard plane offset: 4.9938116
Chessboard plane normal vector: [ 0.9593644  -0.00899234 -0.28202662]
Chessboard plane offset: 8.724273
Chessboard plane normal vector: [ 0.9065703  -0.30866036 -0.2878525 ]
Chessboard plane offset: 8.020993
Chessboard plane normal vector: [ 0.8333353  -0.5520825   0.02751785]
Chessboard plane offset: 5.220803
Chessboard plane normal vector: [ 0.70169747 -0.7011893   0.12631017]
Chessboard plane offset: 4.72078
Chessboard plane normal vector: [ 0.5252832  -0.8445016  -0.10437677]
Chessboard plane offset: 6.103645
Chessboard plane normal vector: [ 0.7235265  -0.68812776 -0.05467756]
Chessboard plane offset: 4.793502
Chessboard plane normal vector: [0.949021   0.13112718 0.28664404]
```

```
Chessboard plane offset: 8.020993
Chessboard plane normal vector: [ 0.8333353  -0.5520825   0.02751785]
Chessboard plane offset: 5.220803
Chessboard plane normal vector: [ 0.70169747 -0.7011893   0.12631017]
Chessboard plane offset: 4.72078
Chessboard plane normal vector: [ 0.5252832  -0.8445016  -0.10437677]
Chessboard plane offset: 6.103645
Chessboard plane normal vector: [ 0.7235265  -0.68812776 -0.05467756]
Chessboard plane offset: 4.793502
Chessboard plane normal vector: [0.949021   0.13112718 0.28664404]
Chessboard plane offset: 9.499026
Chessboard plane normal vector: [ 0.96691203 -0.15794556 -0.2003356 ]
Chessboard plane offset: 8.493222
Chessboard plane normal vector: [ 0.9520712 -0.1539658  0.264301 ]
Chessboard plane offset: 5.743694
Chessboard plane normal vector: [ 0.66224194 -0.745637    0.07389876]
Chessboard plane offset: 7.7494645
Chessboard plane normal vector: [ 0.73737764 -0.5739657  -0.35614258]
Chessboard plane offset: 3.971542
Chessboard plane normal vector: [ 0.9942745  -0.10421911  0.02359328]
Chessboard plane offset: 8.8461685
Chessboard plane normal vector: [ 0.91935325 -0.3607026   0.15710914]
Chessboard plane offset: 5.702525
Chessboard plane normal vector: [ 0.825289   -0.10756965 -0.55437064]
Chessboard plane offset: 5.9781313
Chessboard plane normal vector: [ 0.89093965  0.24536382 -0.3821297 ]
Chessboard plane offset: 7.6186996
Chessboard plane normal vector: [ 0.6037154  -0.7923984   0.08736379]
Chessboard plane offset: 3.703876
Chessboard plane normal vector: [ 0.9014971   0.39908054 -0.16744459]
Chessboard plane offset: 8.140582
Chessboard plane normal vector: [ 0.9633469  -0.25648046 -0.0786166 ]
Chessboard plane offset: 8.213132
Chessboard plane normal vector: [ 0.93809235 -0.22958975  0.25936702]
Chessboard plane offset: 5.204305
```

In the next step we will implement the paper.

$\theta_c = [\theta_{c,1}, \theta_{c,2} \ldots \theta_{c,n}]^T$ $\theta_e = [\theta_{e,1}, \theta_{e,2} \ldots \theta_{e,n}]^T$

$\alpha_c = [\alpha_{c,1}, \alpha_{c,2} \ldots \alpha_{c,n}]^T$ $\alpha_e = [\alpha_{e,1} \ldots \ldots]^T$

These steps are done when I called alpha_c.append(current_alpha_c)

& then alpha_c = np.array(alpha_c)

finally we squeeze so as to remove '1' dimension $(x,1,y) \rightarrow (x,y)$.

Now $\alpha_e \rightarrow \alpha_e$, $n_e \rightarrow \theta_e$ } used in previous part.

we can read θ_c & compute α_c.

Now using the formulas given in step 1 of algorith

$$U S V^T = svd(Q_e Q_c^T)$$

we know them

$$R_1 = V U^T$$

$$t_1 = (Q_c^T Q_c)^{-1} Q_c^T (d_c - d_e)$$

Here Q_c is $(35, \phi)$ which means all Q_c's are combined into 1 matrics (theory told us to do it)

Now using R_1 & t_1 we can use stage II to optimize them. (Left)

So using R_1 & t_1 we can estimate a transformation matrix ${}^c\tilde{T}_L$

$${}^c\tilde{T}_L = \begin{bmatrix} R_1 & T_1 \\ 0 & 1 \end{bmatrix}$$

→ dimensions are 4×4

This will help us to shift points from lidar plane to camera plane.

```
index = 0
alpha_c = []
theta_c = []
alpha_l = []
theta_l = []
for camera in all_frame_names:
    t = np.array(np.loadtxt(f"/kaggle/input/lidar-dataset/camera_parameters/{camera}.jpeg/translation_vecto
    normal_camera = np.array(np.loadtxt(f"/kaggle/input/lidar-dataset/camera_parameters/{camera}.jpeg/camer


    t = np.array([[t[0], t[1], t[2]]])
    current_theta_c = np.array([[normal_camera[0], normal_camera[1], normal_camera[2]]])
    current_theta_l = normals[index]
    current_alpha_l = offsets[index]
    current_alpha_c = t @ current_theta_c.T
    if(current_alpha_c < 0):
        current_alpha_c *=-1
        current_theta_c*=-1
    index+=1

    alpha_c.append(current_alpha_c)
    theta_c.append(current_theta_c)
    alpha_l.append(current_alpha_l)
    theta_l.append(current_theta_l)
```

```
alpha_c = np.array(alpha_c)
alpha_l = np.array(alpha_l)
theta_l = np.array(theta_l)
theta_c = np.array(theta_c)


theta_c = np.squeeze(theta_c)
alpha_c = np.squeeze(alpha_c)

print("The shape of normals is: ", theta_l.shape, theta_c.shape)
print("The shape of offset is: ", alpha_l.shape, alpha_c.shape)
#Now we will get the "R"
# print(lidar_alphas.shape, camera_alphas.shape)
A = theta_l.T @ theta_c
u, sigma, v_t = np.linalg.svd(A)
R_1 = v_t.T @ u.T

print(f"Determinant of matrix: {np.linalg.det(R_1)}")


#Now we will calculate "T"
inv = np.linalg.inv(theta_c.T @ theta_c)
T_1 = inv @ theta_c.T @ (alpha_c - alpha_l)
print("The value of T_1 is", T_1)
print("The value of R_1 is", R_1)
```

```
The shape of normals is:  (38, 3) (38, 3)
The shape of offset is:  (38,) (38,)
Determinant of matrix: 1.0
The value of T_1 is [ 0.08855644 -0.36208099 -0.5992567 ]
The value of R_1 is [[-1.7515889e-01 -9.8454016e-01  1.3609635e-04]
 [ 1.5356026e-02 -2.8701860e-03 -9.9987799e-01]
 [ 9.8442042e-01 -1.7513540e-01  1.5621364e-02]]
```

```python
cTl = np.zeros((4,4))

cTl[:3,:3] = R_1
cTl[:3,3] = T_1
cTl[3][3] = 1

print("The estimated tranformation matrix is: \n",cTl)
```

```
The estimated tranformation matrix is:
 [[-1.75158888e-01 -9.84540164e-01  1.36096351e-04  8.85564404e-02]
 [ 1.53560257e-02 -2.87018600e-03 -9.99877989e-01 -3.62080989e-01]
 [ 9.84420419e-01 -1.75135404e-01  1.56213641e-02 -5.99256704e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  1.00000000e+00]]
```

```python
K = np.array(np.loadtxt("/kaggle/input/lidar-dataset/camera_parameters/camera_intrinsic.txt"))
camera_intrinsic = np.zeros((3,4))
camera_intrinsic[:3,:3] = K
projection = []
for lidar_scan in all_frame_names:
    cloud = PyntCloud.from_file(f"/kaggle/input/lidar-dataset/lidar_scans/{lidar_scan}.pcd")
    pcd = np.array(cloud.points[["x", "y", "z"]])
    img = cv2.imread(f"/kaggle/input/lidar-dataset/camera_images/{lidar_scan}.jpeg")
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    ret, corners = cv2.findChessboardCorners(gray, (8, 6), None)

    points = np.ones((pcd.shape[0], pcd.shape[1]+1))
    points[:points.shape[0],:3] = pcd
    camera_points = (camera_intrinsic @ cTl @ points.T).T

#     print(camera_points.shape)

    homogenous = []
    for current_point in camera_points:
#         print(current_point.shape)
        homogenous.append((current_point / current_point[2])[:2])
#     print(cTl.shape, camera_intrinsic.shape, points.T.shape)
    if(ret):
        projection.append(homogenous)
```

```
index = 0
# lidar_frame = [f"/kaggle/input/lidar-dataset/lidar_scans/{all_frame_names[0]}.pcd"]
print(all_frame_names[0])
for lidar_scan in all_frame_names:
    if(index == 15): break
    img = cv2.imread(f"/kaggle/input/lidar-dataset/camera_images/{lidar_scan}.jpeg")
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    ret, corners =cv2.findChessboardCorners(gray, (8, 6), None)
    if(ret == True):
        points = projection[index]
        index+=1
#         print(len(points), len(points[0]))

        for p in points:
#             if(p[0] < 0 or p[1] < 0): continue
#             print(p[0], p[1])
            cv2.circle(gray, (int(p[0]), int(p[1])), 10, (255, 0, 0), 1)

    plt.imshow(gray)
    plt.show()
```

As discussed in image pipelining in class we can add another row of '0'.

$$K = \begin{bmatrix} R & \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \end{bmatrix}$$

K = camera_intrinsic

So we will get camera_points = $3 \times m$

which we transposed $\Rightarrow$ $m \times 3$

So this is basicall $\begin{bmatrix} x \\ y \\ 3 \end{bmatrix}$ points which according to paper & theory in class we $\rightarrow [x/3, y/3]$

After homogenising, we can just go to the scans & print them.

An optimization, as we need only "25" scans and we have "38" I will only take images in which cv. findChessboard( ) is able to find the corners.

After that I simply printed the points on the image using cv2. circle library.

After finding the transformation matrix, we will bring points from LiDAR plane.

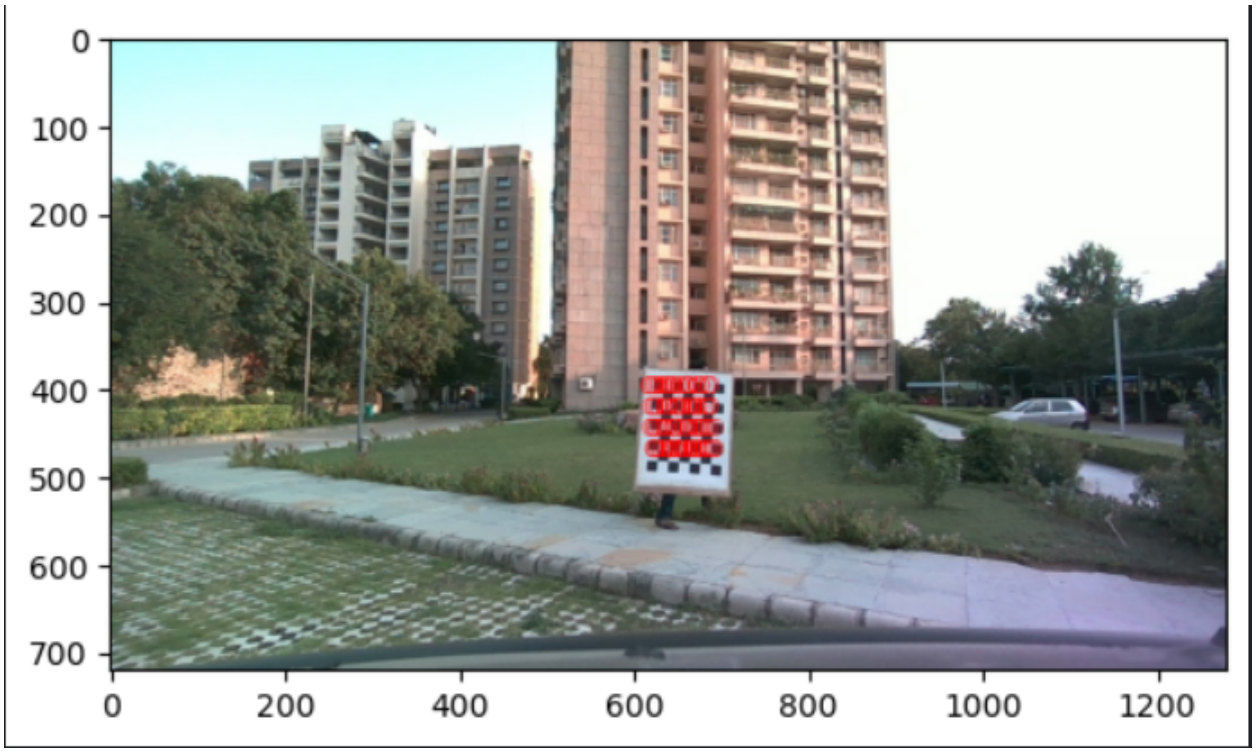
So far that we will first take all the points in a scan ⇒ "points"

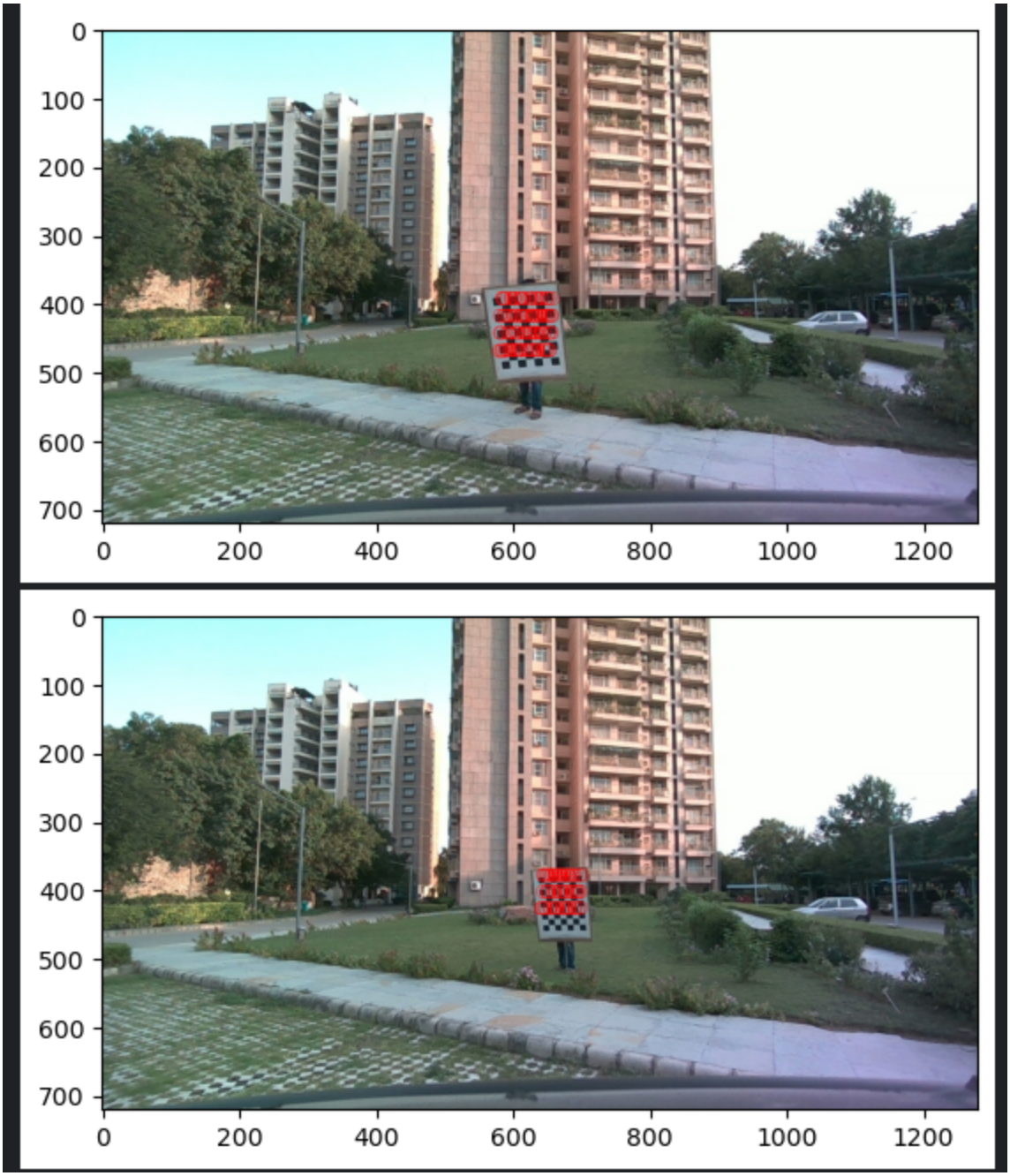
m × 3

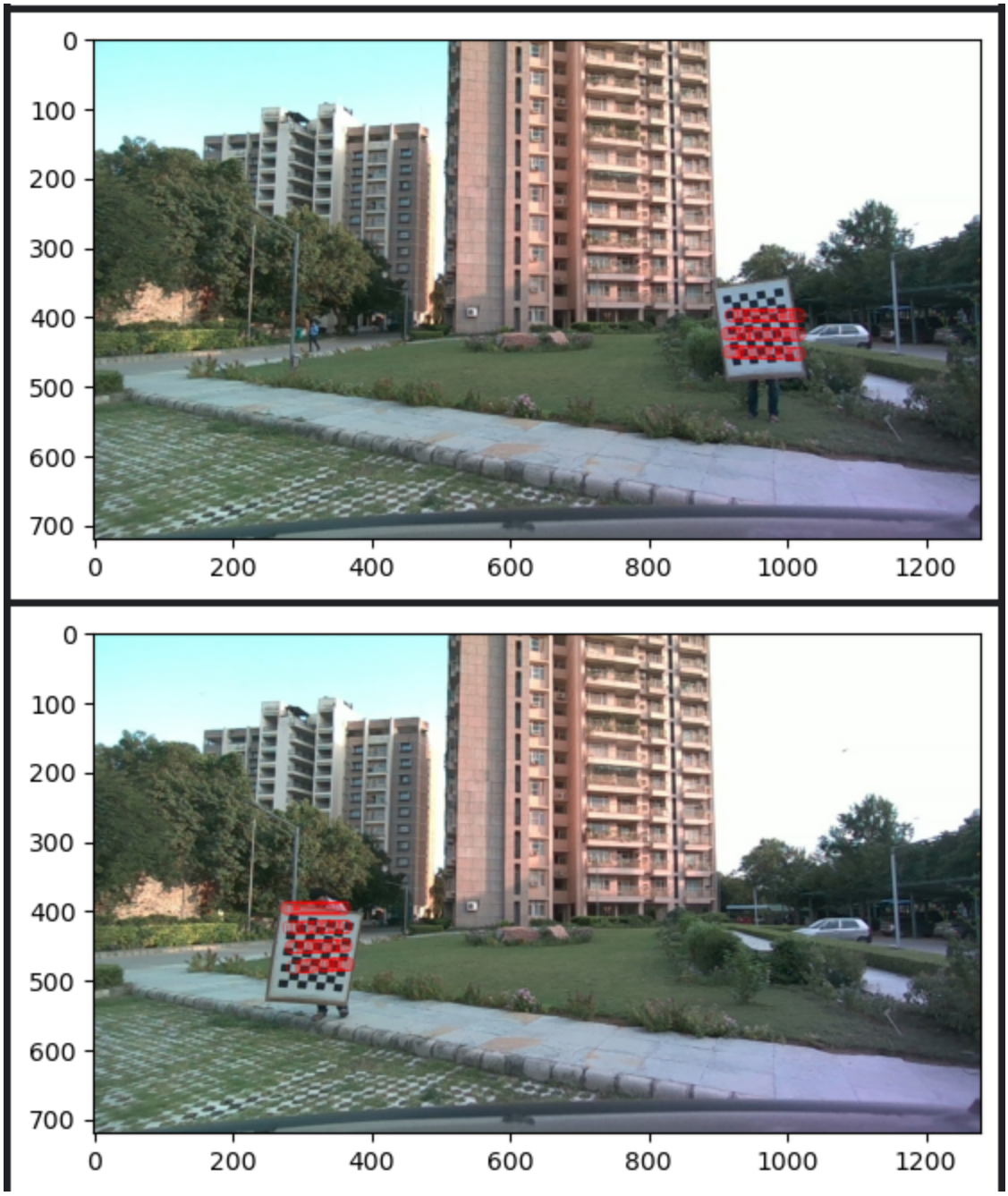
WARNING :- These change image to image. So can't predefine it.

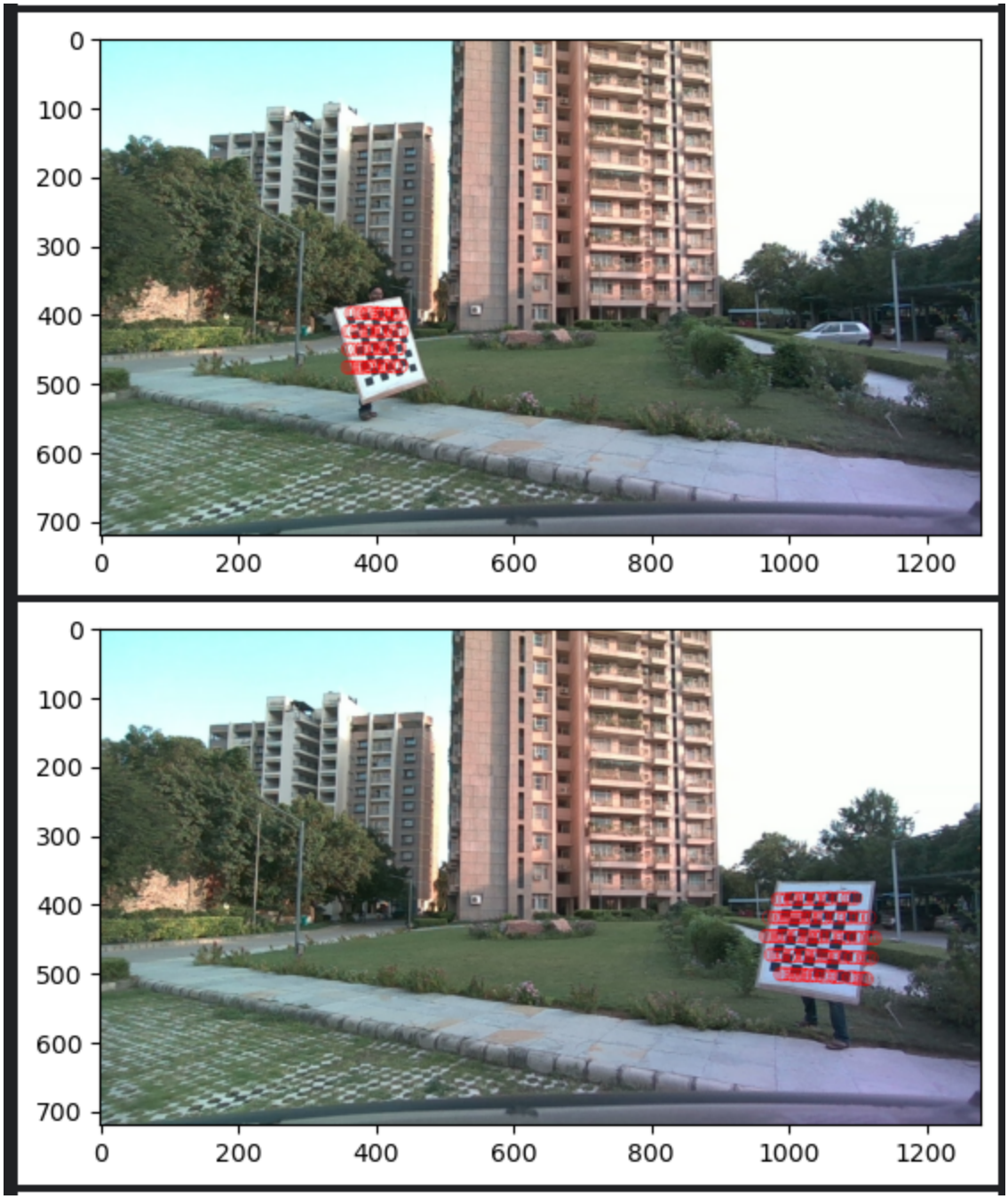
$$camera\_points = (camera\_intrinsic_{3\times4} \cdot {}^{C}\tilde{T}_{L_{4\times4}} \cdot points^{T}_{(m\times3)^{T}})^{T}$$

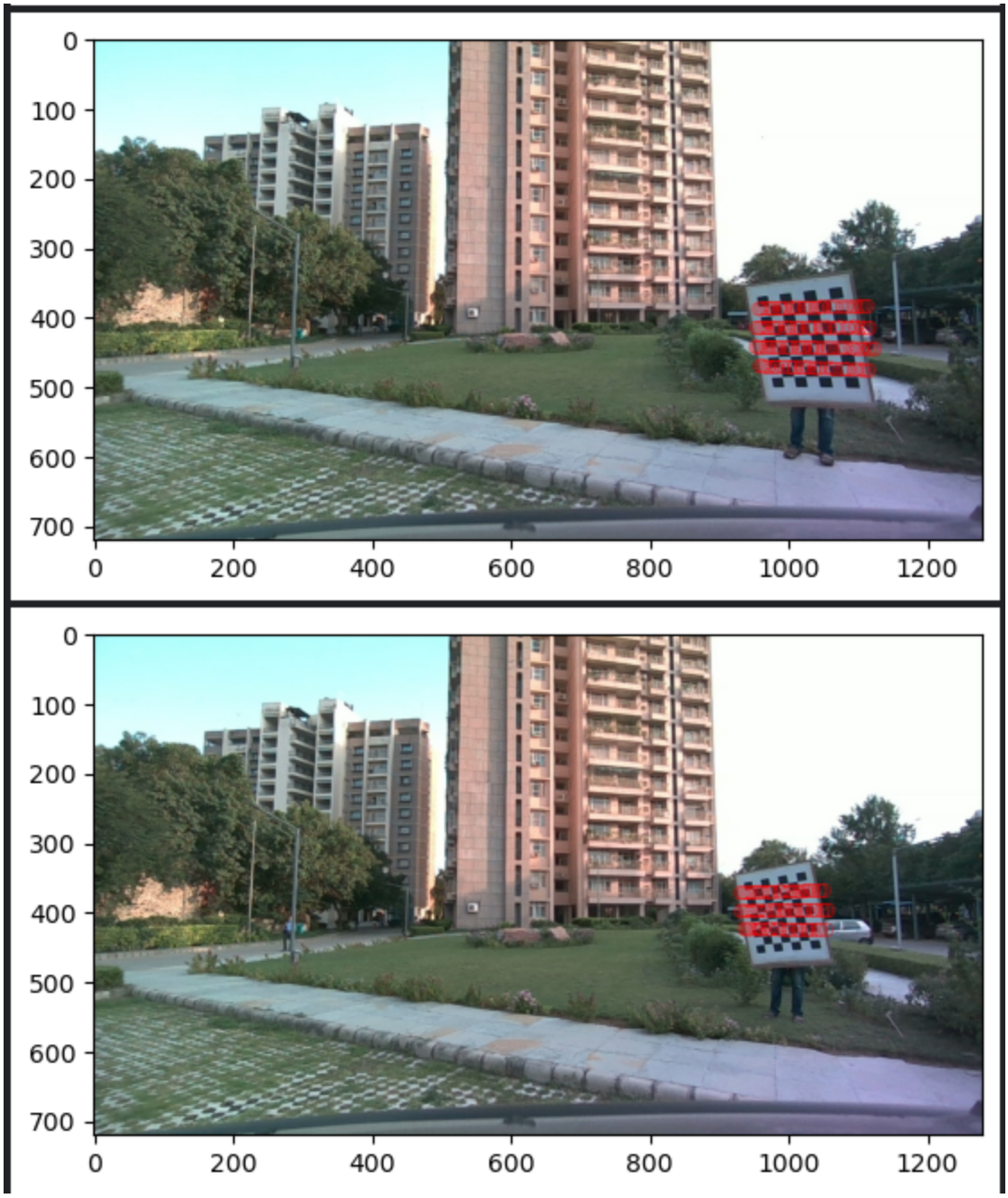
we need to make this 4×4



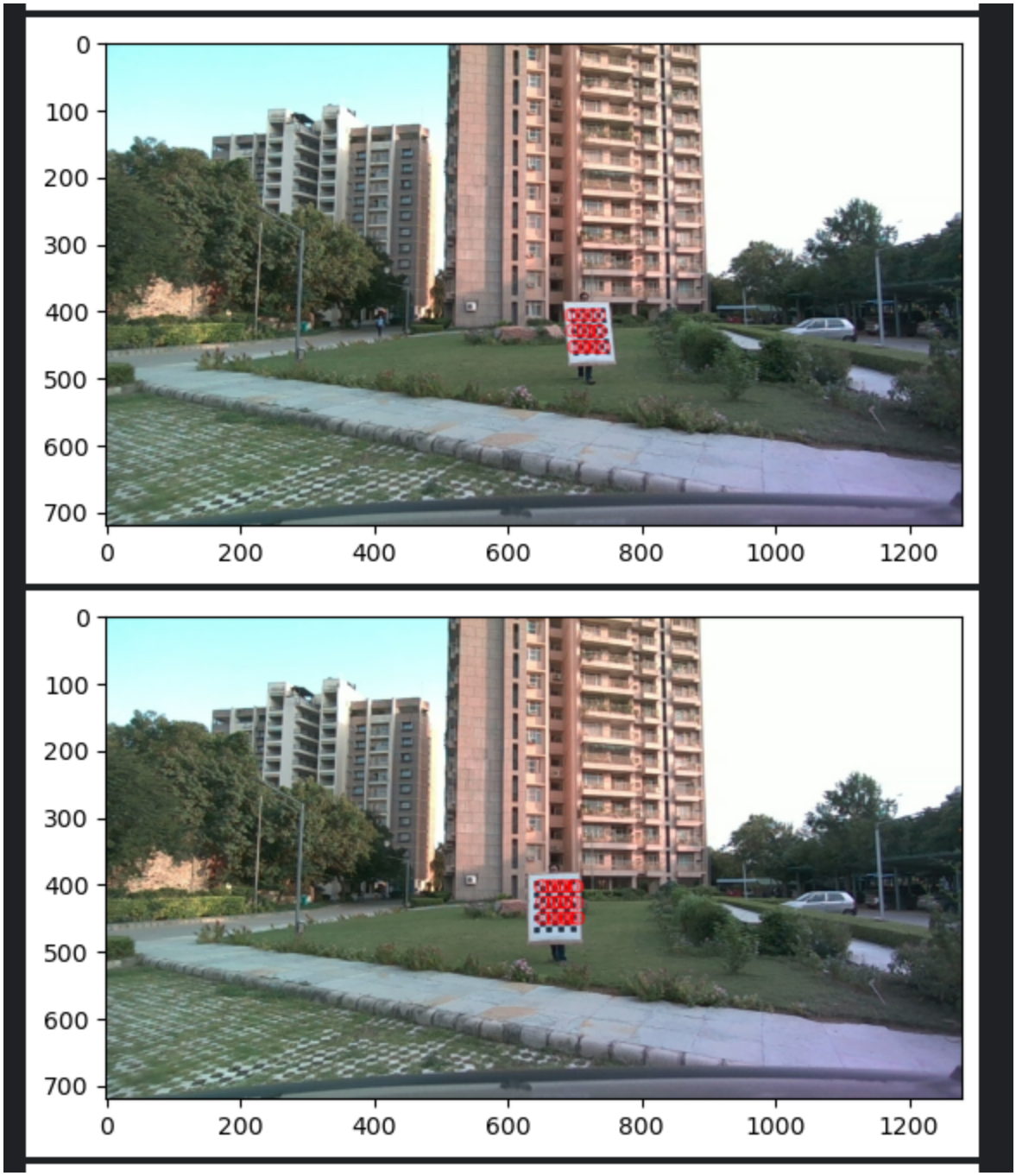

0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

Some of the points lie outside the boundary, but most of the points will lie inside the boundary.

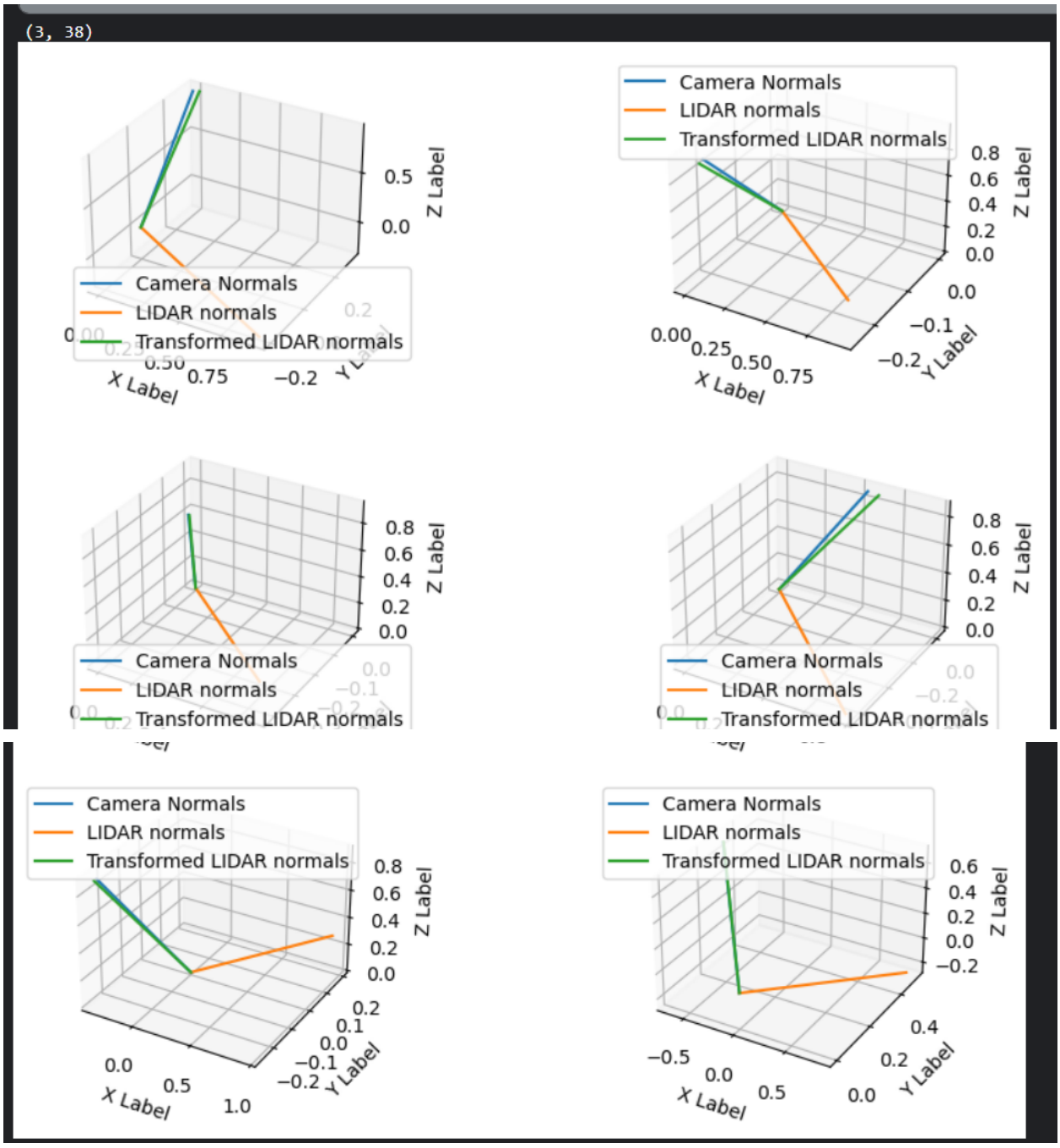


Not for the last part we need to plot all the camera normals, LIDAR normals and transformed LIDAR normals. As given in the question we are taking the dot project of R_1 and the theta_l.

This is basically giving the direction of the LIDAR normal after transformation. The rotated normal should match the camera normals. So check this we will plot the graph.

```
fig = plt.figure(figsize=(10, 10))
ax1 = fig.add_subplot(3, 2, 1, projection='3d')
ax2 = fig.add_subplot(3, 2, 2, projection='3d')
ax3 = fig.add_subplot(3, 2, 3, projection='3d')
ax4 = fig.add_subplot(3, 2, 4, projection='3d')
ax5 = fig.add_subplot(3, 2, 5, projection='3d')
ax6 = fig.add_subplot(3, 2, 6, projection='3d')
ax = [ax1, ax2, ax3, ax4, ax5, ax6]

normals_rotated_lidar = R_1 @ theta_l.T
print(normals_rotated_lidar.shape)


# Plot the data
for i in range(6):

    ax[i].plot([0,theta_c.T[0][i]], [0,theta_c.T[1][i]], [0,theta_c.T[2][i]])
    ax[i].plot([0,theta_l.T[0][i]], [0,theta_l.T[1][i]], [0,theta_l.T[2][i]])
    ax[i].plot([0,normals_rotated_lidar[0][i]], [0,normals_rotated_lidar[1][i]], [0,normals_rotated_lidar[2

    # Set the axis labels
    ax[i].set_xlabel('X Label')
    ax[i].set_ylabel('Y Label')
    ax[i].set_zlabel('Z Label')

    ax[i].legend(['Camera Normals', 'LIDAR normals', 'Transformed LIDAR normals'])
```

```
  # Plot the data
  for i in range(6):

      ax[i].plot([0,theta_c.T[0][i]], [0,theta_c.T[1][i]], [0,theta_c.T[2][i]])
      ax[i].plot([0,theta_l.T[0][i]], [0,theta_l.T[1][i]], [0,theta_l.T[2][i]])
      ax[i].plot([0,normals_rotated_lidar[0][i]], [0,normals_rotated_lidar[1][i]], [0,normals_rotated_lidar[2

      # Set the axis labels
      ax[i].set_xlabel('X Label')
      ax[i].set_ylabel('Y Label')
      ax[i].set_zlabel('Z Label')

      ax[i].legend(['Camera Normals', 'LIDAR normals', 'Transformed LIDAR normals'])

      # Show the plot
  plt.show()
```

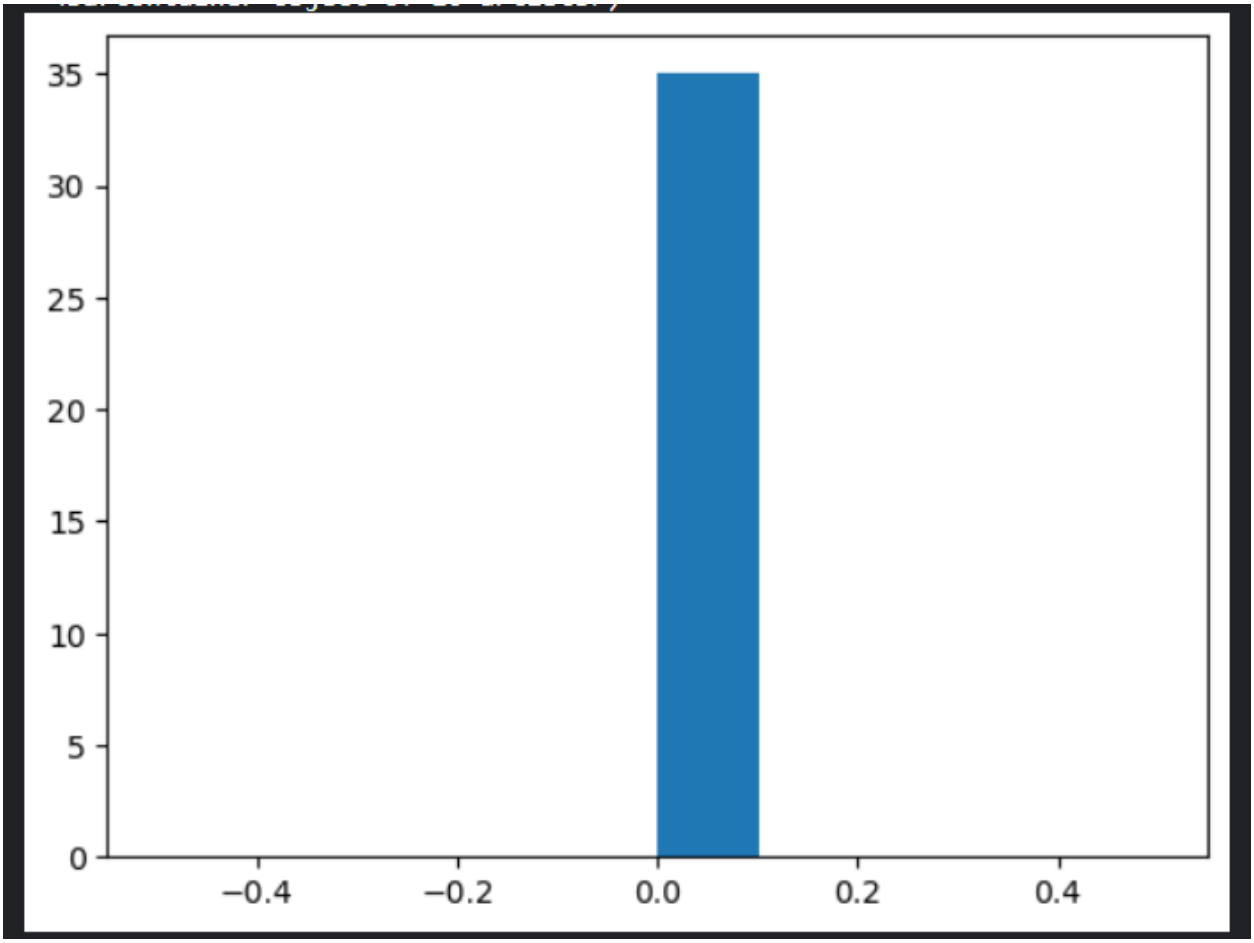


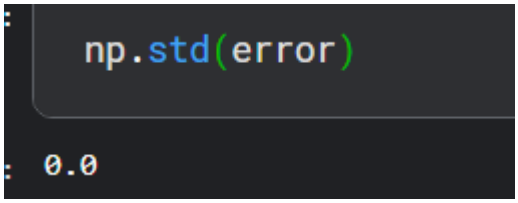
From this we can see that they match very closely.
Now lets see error. Error can be calculated using the dot product of the camera normal and our transformed normal. These should be very close to each other, giving cos(0) = 1.
Hence the error is basically 1 - camera_normal @ transformed normals.

```python
error = []
for i in range(35):
    error.append(theta_c[0] @ normals_rotated_lidar.T[0])
error = np.array(error)
error = 1 - error

print(error)
```

```
[0.00054121 0.00054121 0.00054121 0.00054121 0.00054121 0.00054121
 0.00054121 0.00054121 0.00054121 0.00054121 0.00054121 0.00054121
 0.00054121 0.00054121 0.00054121 0.00054121 0.00054121 0.00054121
 0.00054121 0.00054121 0.00054121 0.00054121 0.00054121 0.00054121
 0.00054121 0.00054121 0.00054121 0.00054121 0.00054121 0.00054121
 0.00054121 0.00054121 0.00054121 0.00054121 0.00054121]
```

```python
plt.hist(error)
```



Std error = 

.