Computer Vision

Assignment 2

Answer 1

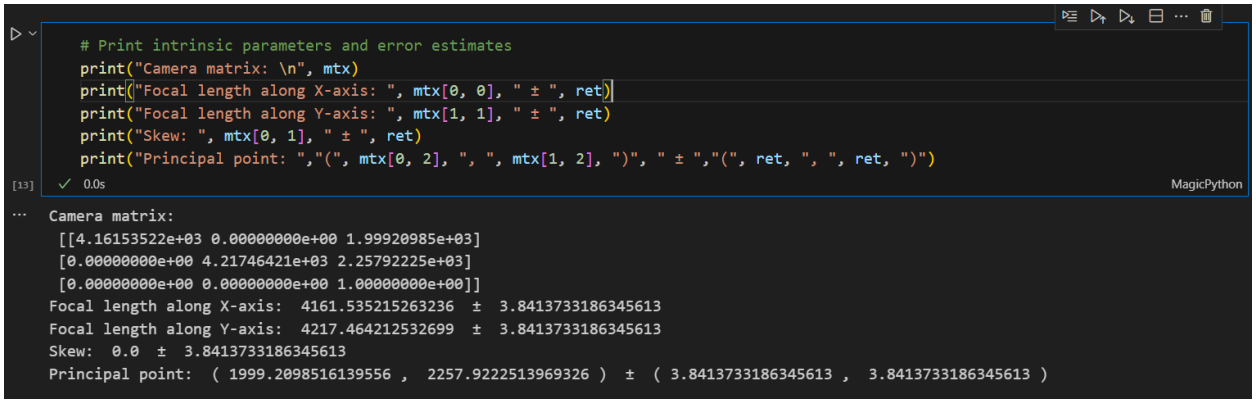
1.
At the beginning of the code, I defined the dimensions of the checkerboard used for calibration. In this case, the pattern is a 7x7 square grid. I also established some criteria for the corner detection algorithm that I will use later.
Next, I create two empty lists to store the 3D and 2D coordinates of the corners of each image. Then I define the 3D coordinates of the corners in the world coordinate system, which are the same in all images. These coordinates are set as a grid of points in the X-Y plane.
Then I specify the path of the folder containing the calibration images and the file extension of the images. I use the glob module to extract the path of each image file in the directory corresponding to the specified file extension.
Next, I iterate over each image file and perform corner detection using OpenCV's **findChessboardCorners** function. When corners are detected in the image, I print a message indicating which frame number was processed and then specify the location of the corners using the **cornerSubPix** function. Then I link the 3D and 2D angular coordinates to the corresponding lists.

Then I call the **cv2.calibrateCamera** function, which takes input as the obj points and the image points, and the image size and gives the camera matrix, distortion coefficients, rotation vectors, and translational vectors as the output.

Then I call the **cv2.calibrationValues** function which gives the output as
fovx: The horizontal field of view in degrees.
fovy: The vertical field of view in degrees.
focalLength: The focal length of the camera in pixels.
principalPoint: The principal point of the camera in pixels, represented as a tuple (x,y).
aspectRatio: The aspect ratio of the camera.

This code is written to calibrate the camera using a known pattern (in this case a checkerboard). The process involves identifying the corners of a pattern in multiple images taken with the same camera and then using the coordinates of those corners to estimate the internal and external parameters of the camera. These parameters can then be used to correct lens distortions and other image errors.

The focal length, skew parameter, and principal point along with error estimates are:
The error estimates are given by the function




```
# Print intrinsic parameters and error estimates
print("Camera matrix: \n", mtx)
print("Focal length along X-axis: ", mtx[0, 0], " ± ", ret)
print("Focal length along Y-axis: ", mtx[1, 1], " ± ", ret)
print("Skew: ", mtx[0, 1], " ± ", ret)
print("Principal point: ","(", mtx[0, 2], ", ", mtx[1, 2], ")", " ± ","(", ret, ", ", ret, ")")
```

```
Camera matrix:
 [[4.16153522e+03 0.00000000e+00 1.99920985e+03]
 [0.00000000e+00 4.21746421e+03 2.25792225e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Focal length along X-axis:  4161.535215263236  ±  3.8413733186345613
Focal length along Y-axis:  4217.464212532699  ±  3.8413733186345613
Skew:  0.0  ±  3.8413733186345613
Principal point:  ( 1999.2098516139556 ,  2257.9222513969326 )  ±  ( 3.8413733186345613 ,  3.8413733186345613 )
```

2.
In this part of the code, I found out the rotation matrix and translation vector for each of the images. The steps to do so are:-

First, I specified the source folder and the file extension of the chessboard images I wanted to use. Then, I used the glob module to create a list of all the file paths in that folder that matched the specified extension.

Next, I looped through each file path in the list and read in the corresponding image using the **cv2.imread()** function. I then converted the image to grayscale, which is necessary for corner detection.

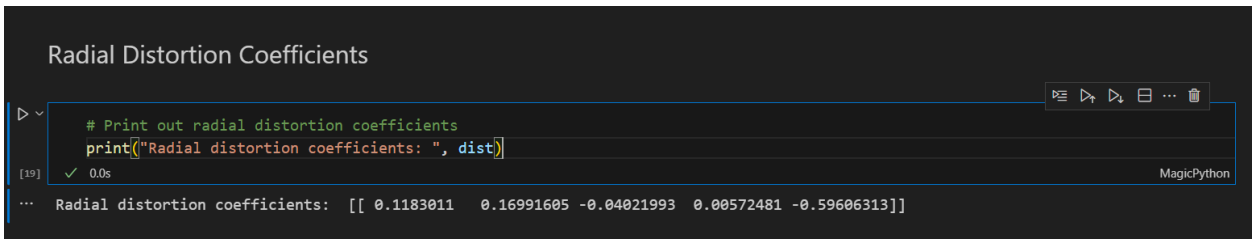
Using the grayscale image, I used the **cv2.findChessboardCorners()** function to detect the corners of the chessboard. This function returned the pixel coordinates of the detected corners.

With the 2D pixel coordinates of the chessboard corners in hand, I used the **cv2.solvePnP()** function to estimate the camera's extrinsic parameters, which include the rotation and translation of the camera relative to the chessboard. This was necessary for undistorting the images and performing accurate measurements.

I then used the **cv2.Rodrigues()** function to convert the rotation vector to a rotation matrix. This rotation matrix can be used to undistort the images and transform them into a common reference frame for accurate measurements.

Finally, I printed out the rotation matrix and translation vector for each image. This was useful for understanding how the camera is oriented and positioned in each image and for verifying that the calibration process was successful.

3.
The distortion coefficients are returned by the **cv2.calibrateCamera** function, and output is shown as below



Now, using the radial distortion coefficients, I undistorted 5 images and the explanation of the code is as follows.

I used the glob module to get a list of all the image files in a given directory with a specific file extension. Then I looped through each image file using a for loop. Inside the loop, I checked if the image number had exceeded 6, and if so, I broke out of the loop.

Next, I loaded the raw image using OpenCV's **imread()** function. I then used the **cv2.undistort()** function to undistort the image using the camera matrix and distortion coefficients obtained from the calibration process.

After undistorting the image, I created a window name using the current image number. Finally, I used Matplotlib's **imshow()** function to display the undistorted image in a new window with the specified window name.

The undistorted images are shown below:

Image 1

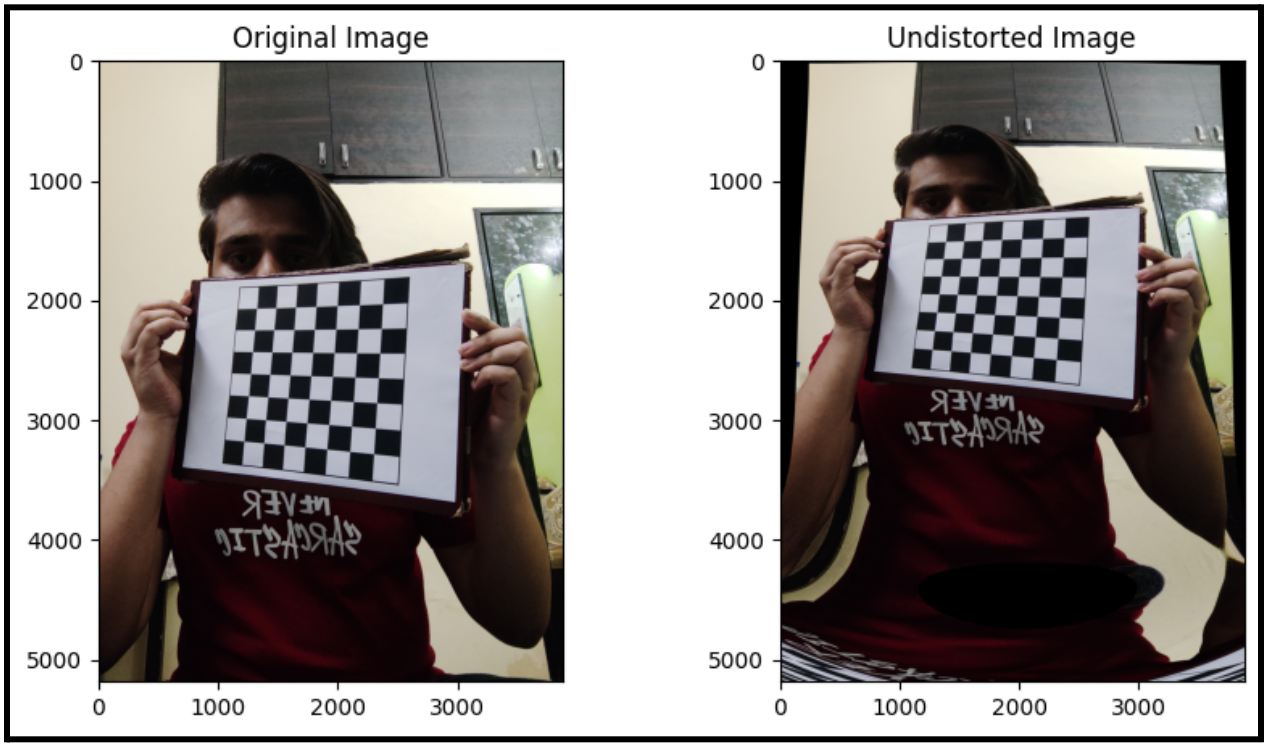


Image 2

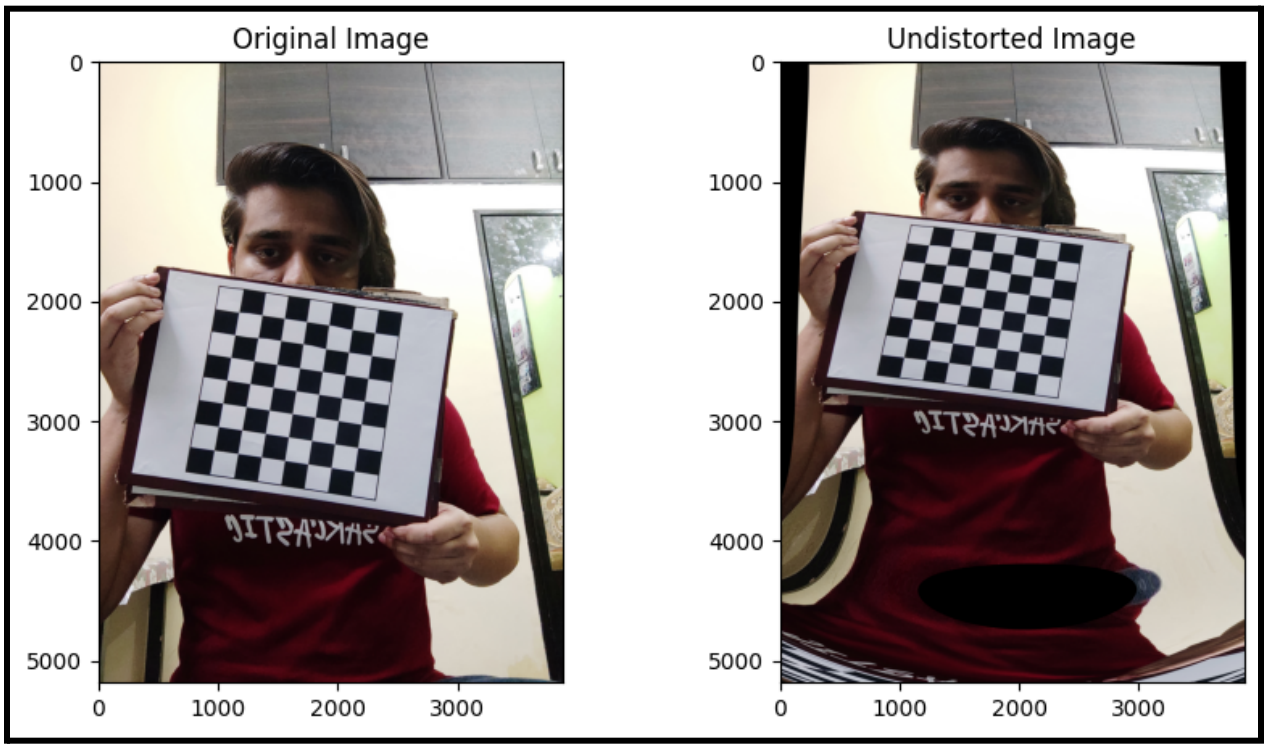


Image 3

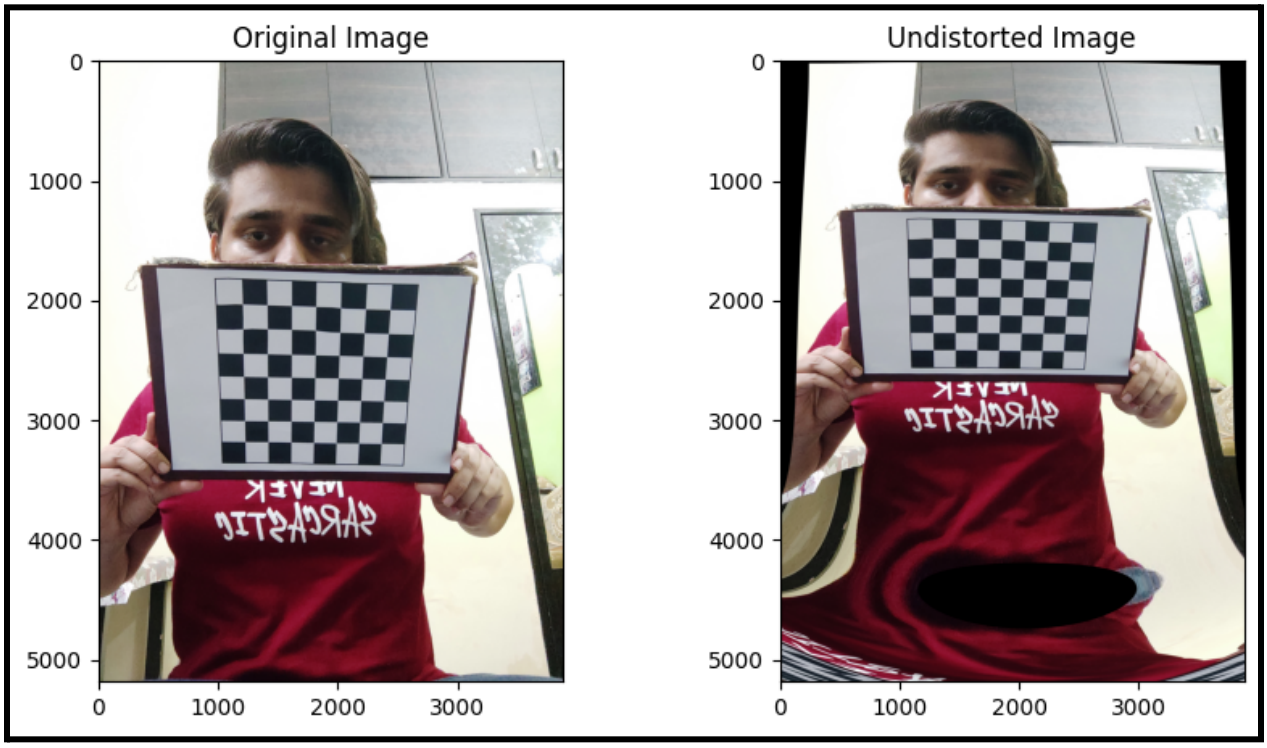


Image 4

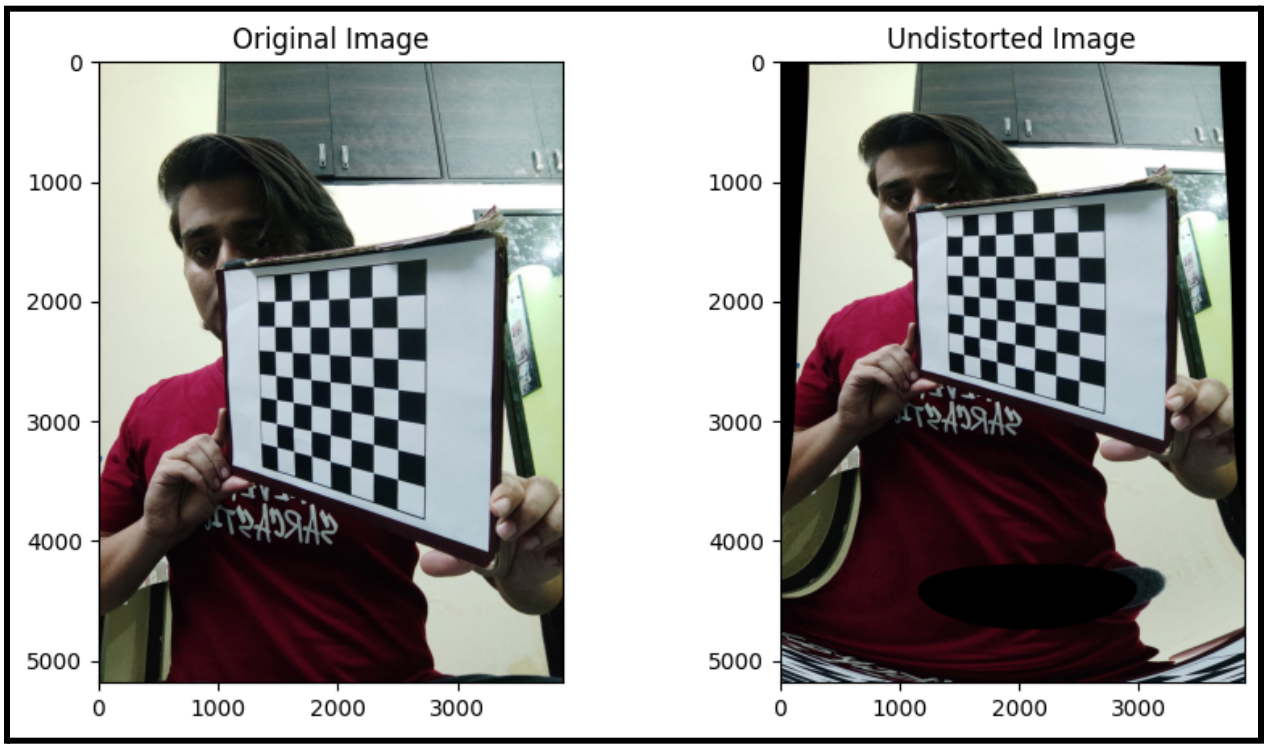
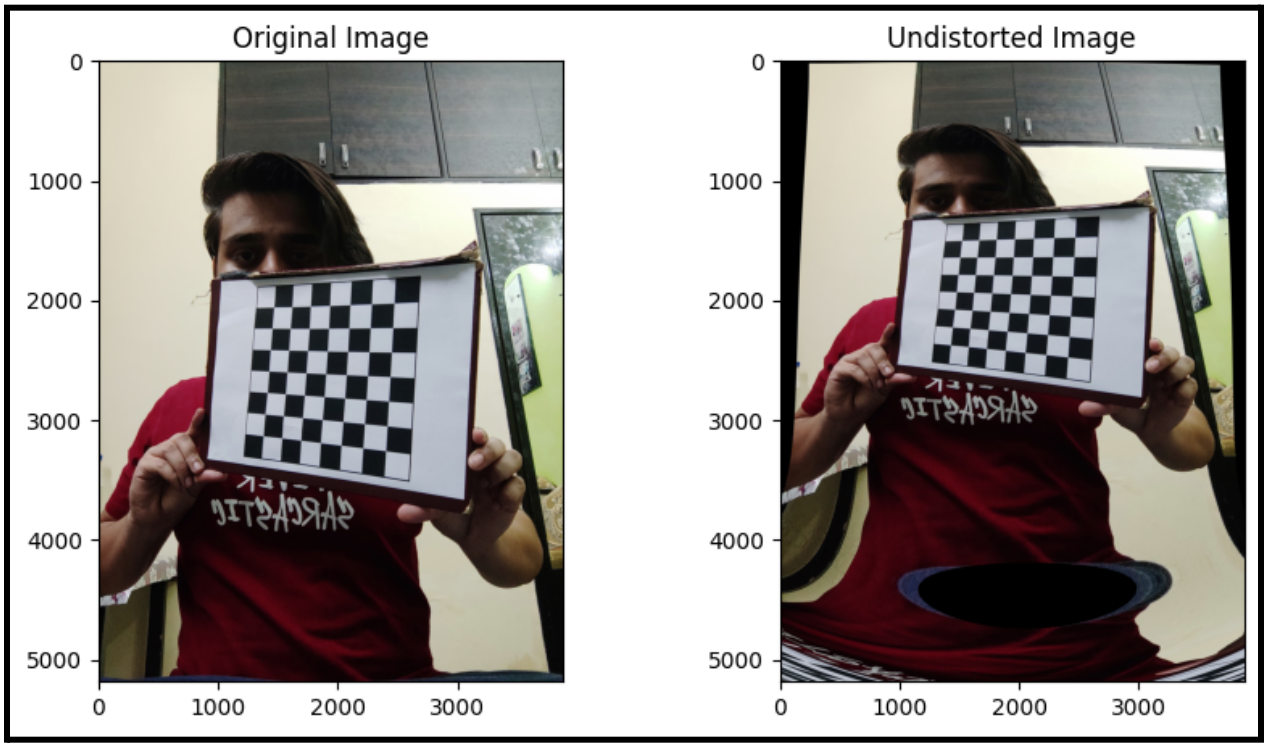


Image 5



As we can see in the original images, the lines which were straight in the actual world, appear to be a bit distorted in the camera world (i.e., in the original image). When we used the radial distortion coefficients to undistort the images, the straight lines that appeared distorted in the original image were now converted to straight lines in the undistorted image.
This distortion can be caused by lens imperfections or other factors and can result in distorted or warped images. The **cv2.undistort()** function can correct this distortion and produce a more accurate representation of the scene.

The algorithm used by cv2.undistort() involves several steps:

- Compute the camera matrix and distortion coefficients: The camera matrix contains information about the camera's intrinsic parameters, such as the focal length and principal point, while the distortion coefficients represent the lens distortion. These parameters can be obtained through a calibration process.

- Generate an undistortion map: The undistortion map is a look-up table that maps each pixel in the distorted image to its corresponding pixel in the undistorted image. This map is computed using the camera matrix and distortion coefficients.

- Apply the undistortion map to the image: The undistortion map is used to remap each pixel in the distorted image to its corresponding pixel in the undistorted image.

4.
As I was calibrating my camera, I wanted to make sure that my calibration process was accurate. So, I wrote some code to calculate the re-projection error for each image in my calibration set.

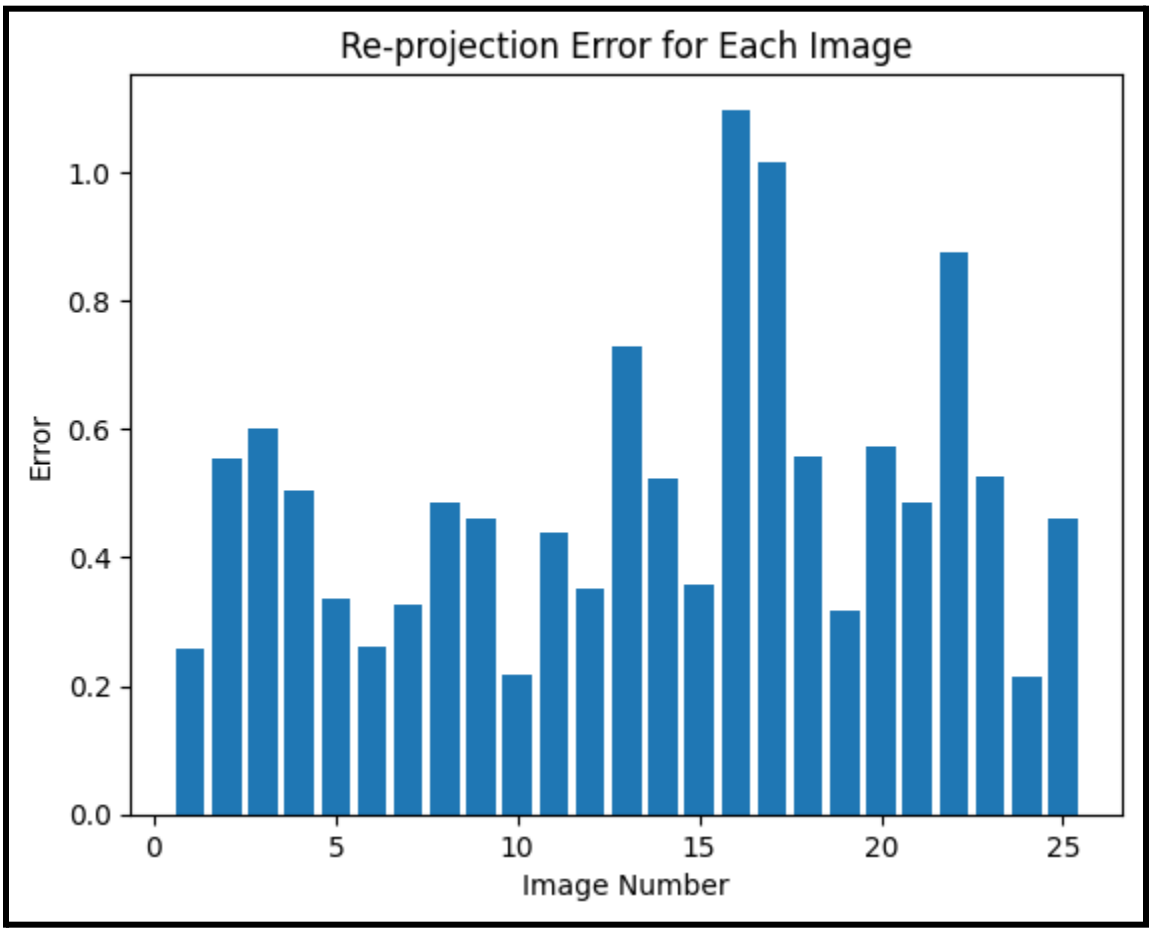
To do this, I first initialized an empty list called errors. I then looped through each image in my calibration set and projected the object points onto the image plane using the **cv2.projectPoints()** function. This gave me the projected points for each image.

Next, I calculated the Euclidean distance between the projected points and the detected image points using the **cv2.norm()** function. I then divided the result by the number of projected points to get the average error per point. I added this error to the errors list.

After calculating the re-projection error for each image, I plotted a bar graph showing the error for each image. I used the **plt.bar()** function to create the bar graph, and labeled it with a title, x-axis label, y-axis label. Finally, I displayed the graph using the **plt.show()** function.

By calculating and visualizing the re-projection errors for each image, I was able to get a sense of how accurate my camera calibration was. I also computed the mean and standard deviation of the errors using the **np.mean()** and **np.std()** functions, respectively, to get a more quantitative sense of the calibration accuracy. This allowed me to fine-tune my calibration process and improve the accuracy of my camera measurements.

The reprojection error bar chart is shown below:



The Mean re-projection error and Standard deviation of re-projection error output is shown below

```
Mean re-projection error: 0.50 pixels
Standard deviation of re-projection error: 0.22 pixels
```

5.

I wrote a script that processes images to find and draw corners. First, I loop through a list of image files using enumerate() to get the index and the file name. Then, I load the image using the **cv2.imread()** function.

Next, I convert the image to grayscale using the **cv2.cvtColor()** function. I then use the **cv2.findChessboardCorners()** function to find the corners in the image. This function takes three parameters: the grayscale image, a tuple specifying the number of corners in the checkerboard, and None.

After finding the corners, I create a copy of the original image using the copy() method. I then use the **cv2.drawChessboardCorners()** function to draw the detected corners onto the copy of the image. This function takes four parameters: the copy of the image, the tuple specifying the number of corners, the corners that were detected, and the return value of the **cv2.findChessboardCorners()** function.

I then use the **cv2.projectPoints()** function to re-project the corners onto the image. This function takes five parameters: the object points, rotation vectors, translation vectors, camera matrix, and distortion coefficients. The function returns the re-projected points and an error value, which I ignore.

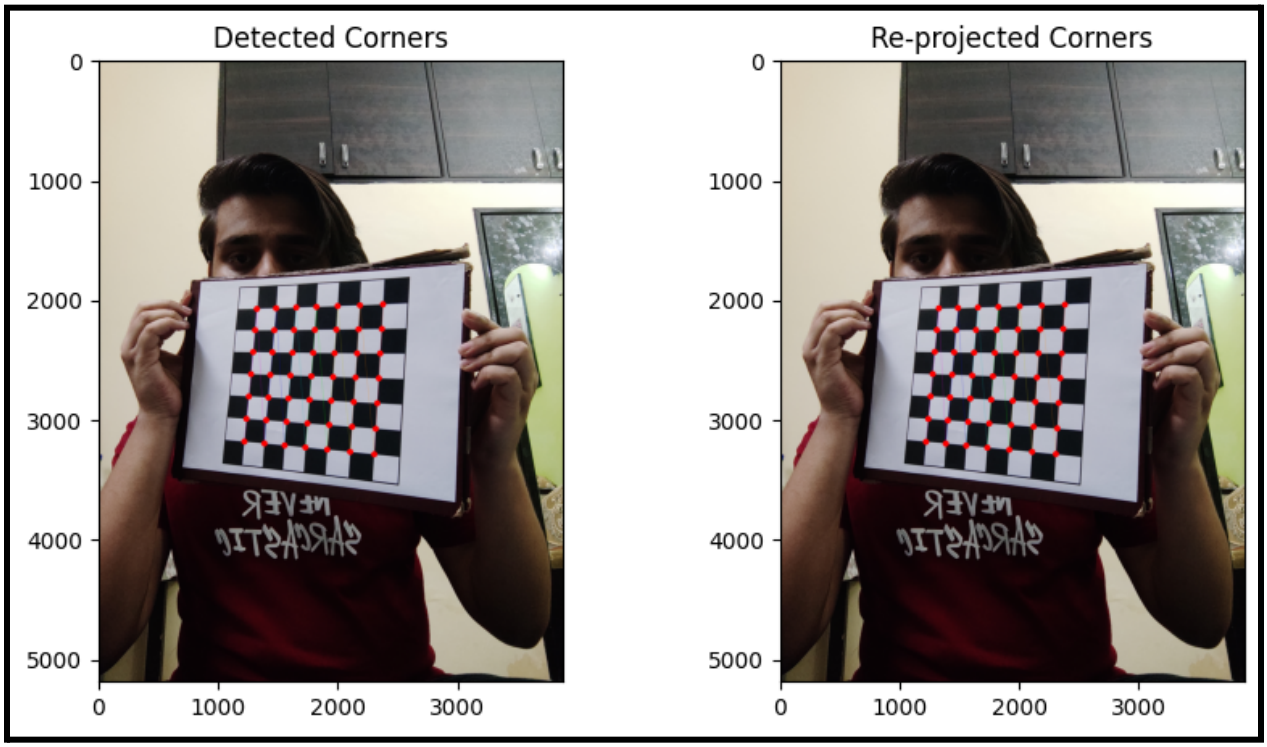
I create another copy of the original image and use the **cv2.drawChessboardCorners()** function again to draw the re-projected corners onto the copy. I set the last parameter of this function to True to indicate that the corners should be drawn as a closed circle.

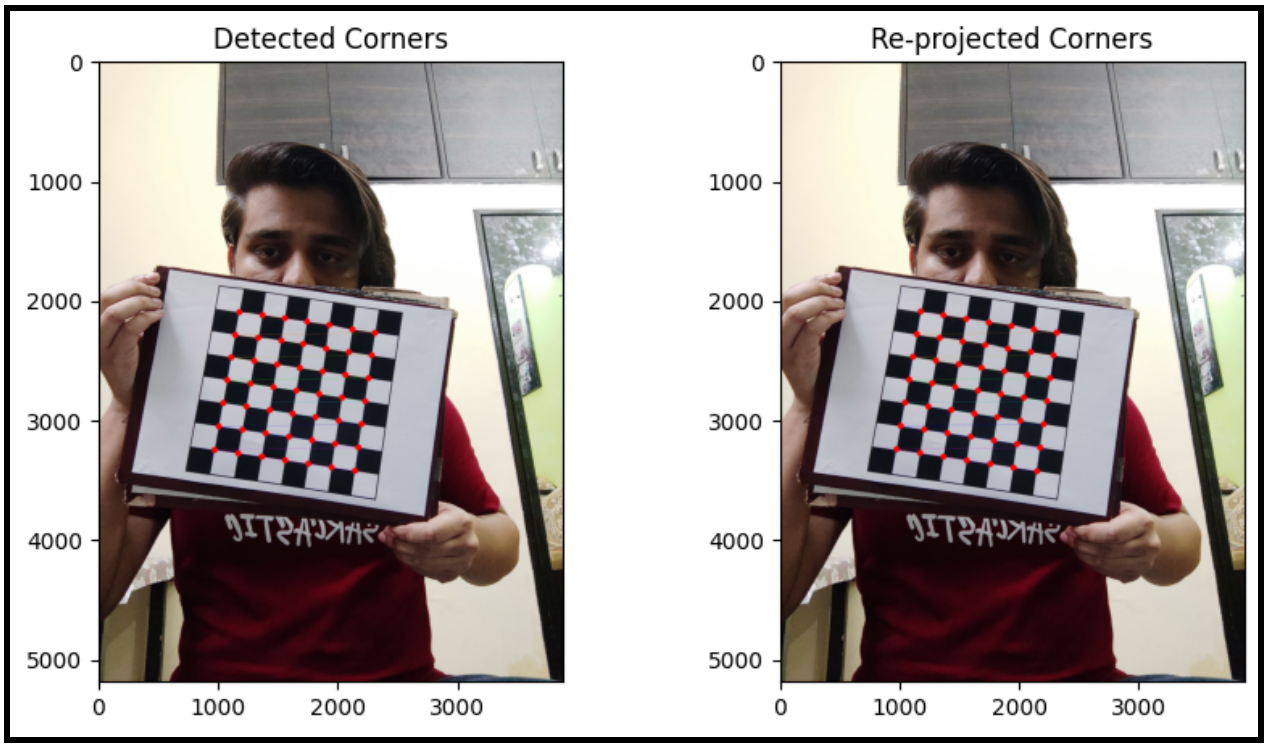
Finally, I create a figure with two subplots using the **plt.subplots()** function, with the first subplot showing the image with the detected corners, and the second subplot showing the image with the re-projected corners. I use the **imshow()** function to display the images in the subplots and set titles for each subplot using the set_title() method. I then display the plot using the **plt.show()** function.

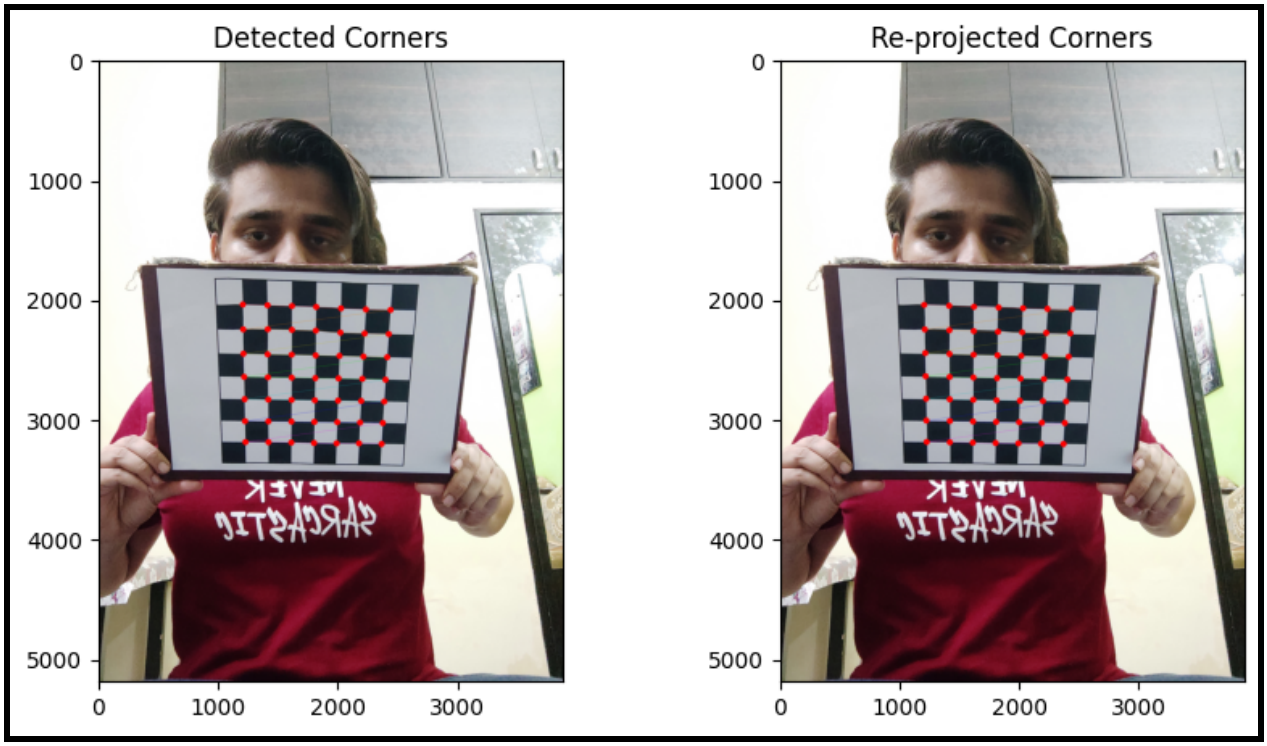
The calibration procedure' accuracy is measured by the reprojection error, which shows how well the predicted intrinsic and extrinsic parameters suit the chessboard pattern's corners. It is calculated as the average distance between the image's reprojected corners and detected corners.

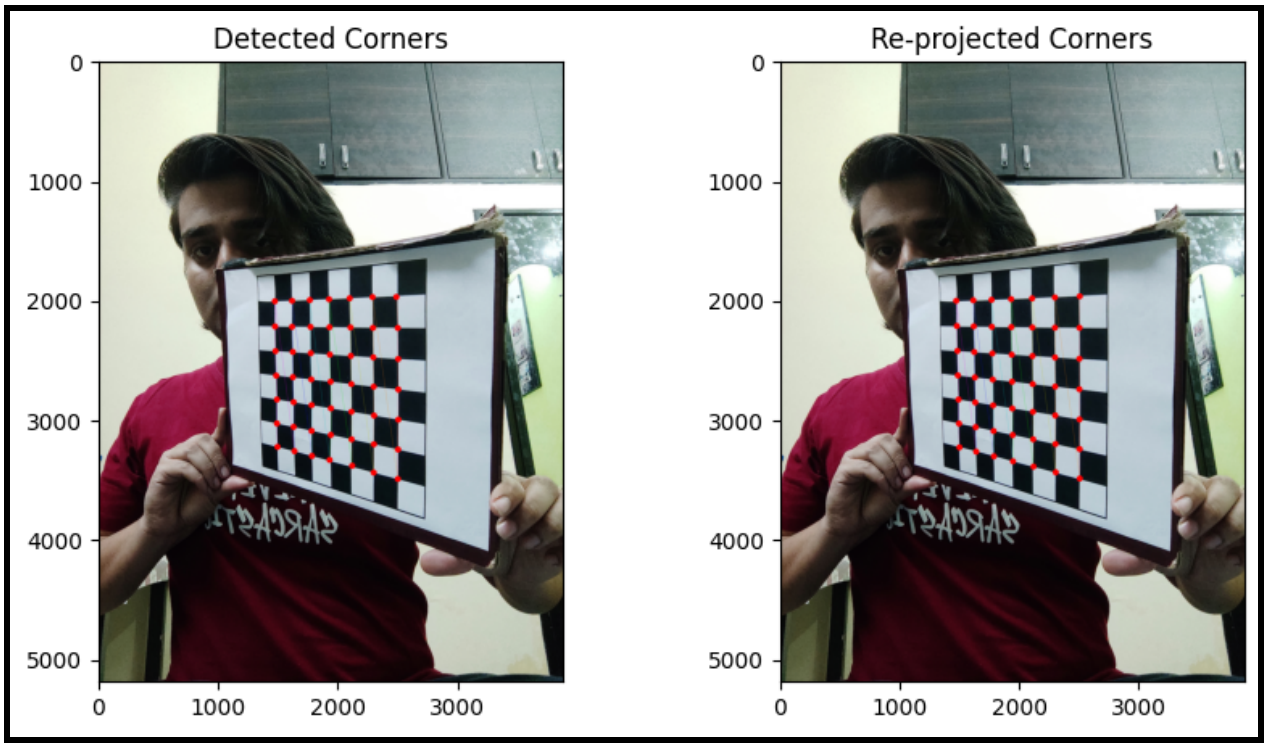
The **projectPoints()** function in OpenCV, which accepts the 3D world points of the chessboard corners, the estimated intrinsic and extrinsic camera parameters, and the distortion coefficients as input, is used to calculate the reprojection error.
The **projectPoints()** function uses the estimated parameters to project the 3D world points onto the 2D picture plane and then returns the matching 2D image points.
The Euclidean distance between their coordinates is used to compare the chessboard pattern's identified corners to the reprojected corners. The reprojection error is calculated as the sum of the distances between each corner of the image. The calibration process was more accurate if there was a better fit between the detected corners and the reprojected corners, which is indicated by a lower reprojection error.

Some of the outputs are shown below:




Detected Corners
Re-projected Corners
0
1000
2000
3000
4000
5000


Detected Corners
Re-projected Corners
0
1000
2000
3000
4000
5000



6.

In this code, I computed the checkerboard plane normals in the camera coordinate frame of reference. To do this, I loop through each object point and rotation vector using a **for** loop.

For each object point, I use the **cv2.Rodrigues()** function to convert the rotation vector to a rotation matrix R. I then compute the checkerboard plane normal in the camera coordinate frame by taking the dot product of R and a unit vector pointing in the z-direction [0, 0, 1].

I append the resulting normal vector to a list of **plane_normals** for each image. Once all the normals have been computed, I print them out using a for loop and the print() function.

Plane normals output example shown below:

```
The plane normals for each image are:

For image  1 :
[0.38337926 0.20683334 0.9001335 ]

For image  2 :
[-8.50578327e-04  1.61832906e-01  9.86817809e-01]

For image  3 :
[-0.0455157  -0.20992275  0.97665795]

For image  4 :
[ 0.66591358 -0.0320137   0.74534169]

For image  5 :
[0.45397502 0.15183171 0.87798281]

For image  6 :
[0.24094387 0.12715852 0.96217294]

For image  7 :
[-0.55840051 -0.0419454   0.82851038]

For image  8 :
[-0.53967705  0.42282239  0.72799033]

For image  9 :
[ 0.40941383 -0.18497022  0.89340155]

For image  10 :
[-0.41898367  0.51436868  0.74824966]
```

Answer 2

1.
For this part, I wrote a script that reads LIDAR scan data in the PCD format from a directory on the local file system.
The first step is to define the path to the source folder where the PCD files are stored. The script then uses the glob library to find all PCD files with a specific file extension and sorts them in alphabetical order. I then use a counter variable ctr to keep track of the number of LIDAR scans processed.

Next, for each PCD file, I use the **o3d.io.read_point_cloud** function from the Open3D library to load the point cloud data. I then apply the **RANSAC** algorithm to extract the planar LIDAR points from the loaded point cloud data. **RANSAC** fits a plane model to the point cloud data by randomly sampling points and calculating the inliers. I use the **segment_plane** function of Open3D to extract the planar points from the LIDAR scan data.

After extracting the planar points, I compute the centroid of these points by taking their mean. This is done using the NumPy **mean** function. I then compute the **singular value decomposition (SVD)** of the planar points, and the normal of the chessboard plane is obtained from the third column of the "**mtx"** matrix. The offset of the chessboard plane is computed by taking the dot product of the normal vector and the centroid of the planar points. I then append the computed normal and offset values to the respective lists for later use.

Finally, I print the chessboard plane normals and corresponding offsets for each LIDAR scan. This is done to verify that the plane detection algorithm is working correctly. The counter variable ctr is incremented after processing each LIDAR scan.

2.

Derivation

Some definitions to begin with:

1. **cam_plane_normal**
   It is the plane normal of the chessboard in the camera frame of reference
2. **lidar_plane_normal**
   It is the plane normal of the chessboard in the lidar frame of reference
3. **matrix_lidar_camera**
   It is the rotation matrix transforming a vector from the LIDAR space to the camera space.
4. **vector_lidar_camera**
   It is the vector translating a vector from the LIDAR frame of reference to the camera frame of reference.
5. **camera_normals**
   The matrix containing all the plane normal of the chessboard in the camera frame of reference
6. **lidar_normals**
   The matrix containing all the plane normal of the chessboard in the lidar frame of reference
7. **centroid**
   The centroid of all the LIDAR points on the chessboard plane.

Statement 1:
The chessboard normal in the camera frame of reference (**cam_plane_normal**), can be computed by performing the matrix multiplication of the rotation matrix, which transforms a vector from the LIDAR space to camera space (**matrix_lidar_camera**) with the chessboard normal in the lidar frame of reference (**lidar_plane_normal**).

Eqn:
cam_plane_normal = matrix_lidar_camera * lidar_plane_normal

Statement 2:
The equation for the **camera_normals** can be found using the formula:

Eqn:
camera_normals = lidar_normals * transpose (matrix_lidar_camera)

The matrices of **camera_normals** and **lidar_normals,** contain a list of chessboard plane normal vectors, and each chessboard plane normal vector contains their respective x,y, and z coordinates of their respective vectors.

Now let us consider a pair of normals (normal w.r.t camera and normal w.r.t lidar).
To achieve much accuracy in computing the rotation matrix, we need to compute it in such a way that the difference between the respective camera normal and lidar normal is as small as possible.

Hence, we will adopt the least-squares solution to compute the **matrix_lidar_camera.**
The equation to do that would be:

matrix_lidar_camera =
inverse(transpose(lidar_normals) * lidar_normals) * transpose(lidar_normals) * camera_normals

We can now also calculate the **vector_lidar_camera,** (vector representing the displacement of the origin between the LIDAR frame of reference and the origin of the camera frame of reference) using the **matrix_lidar_camera.**

The equation to do that would be

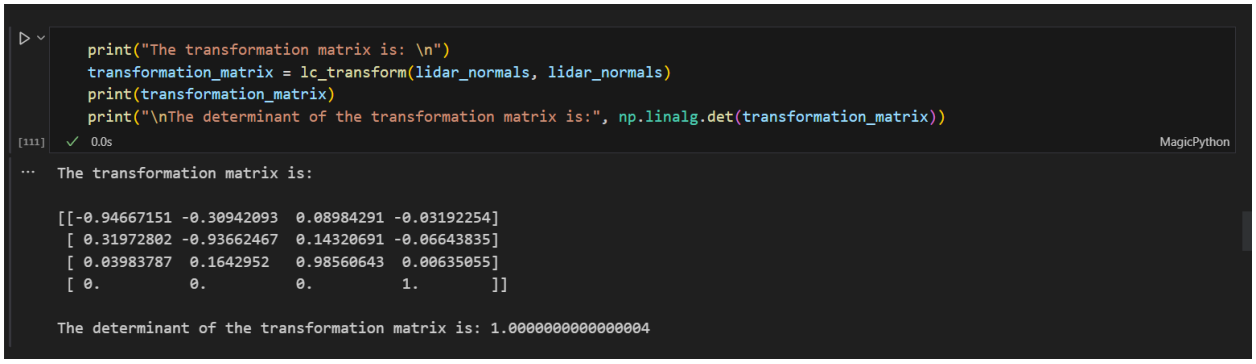
vector_lidar_camera = (-)matrix_lidar_camera * centroid

```
The camera to lidar transformation matrix is:

 [[ 0.05080857  0.02069283  0.11542587 -0.21501955]
 [ 0.21776785  0.03766529  0.17709986 -1.20374713]
 [ 0.07556496  0.33514312  0.23443432  0.19395206]]
```

3.

Using the above derivation, a function **lc_transform** is written, which estimates the transformation. The estimated transformation matrix from that is:



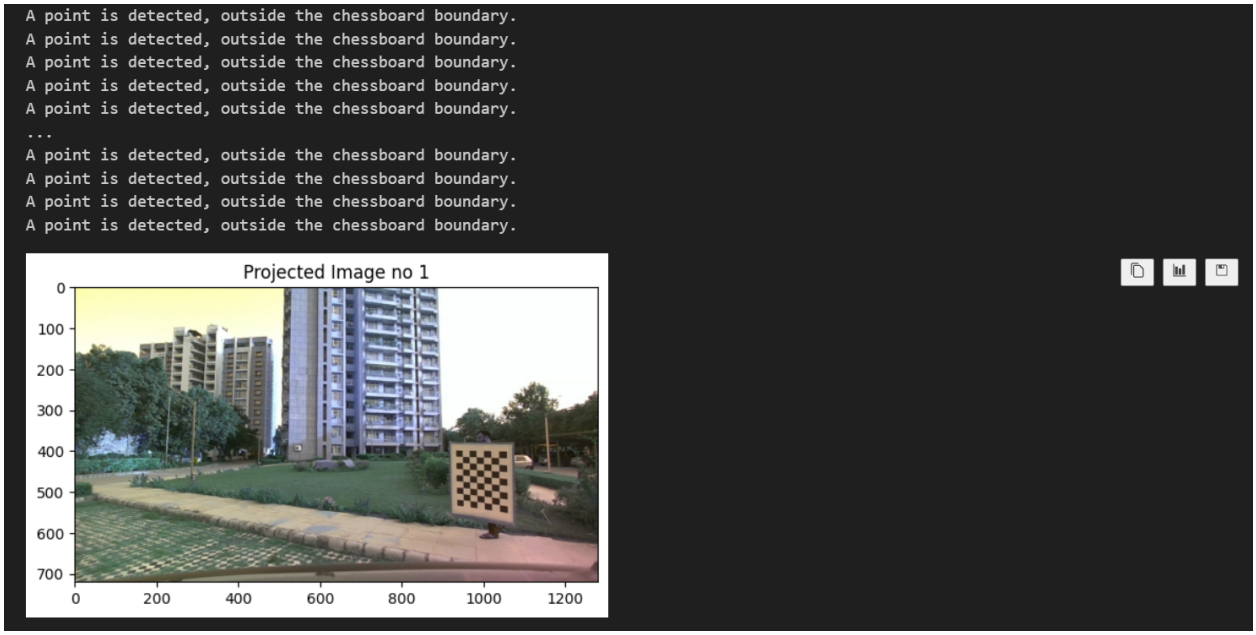
4.

In this part, I projected the LIDAR scans onto camera images by transforming the LIDAR point clouds into the camera frame of reference. I first defined the paths for the camera images and LIDAR scans. Then, I used the glob module to retrieve all the images and scans from their respective directories and sort them in ascending order.

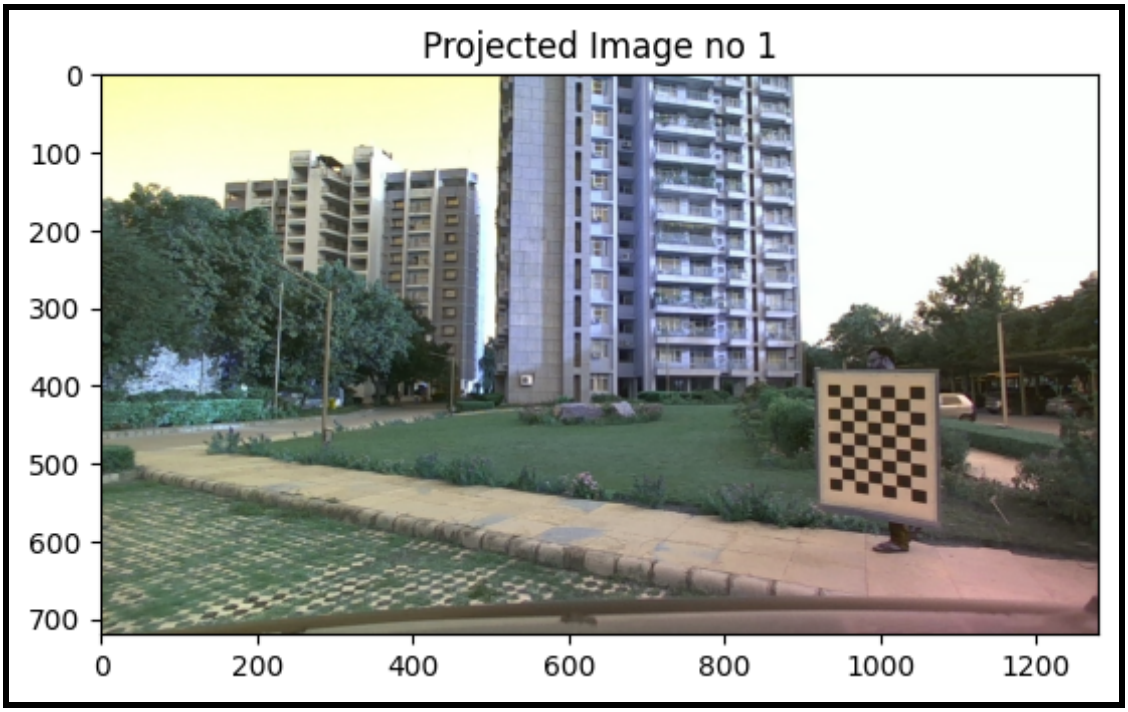
Next, I created a loop that iterates through each image in the camera folder. For each iteration, I loaded the corresponding camera image and LIDAR scan using OpenCV and Open3D, respectively. The LIDAR scan was then transformed using the transformation matrix which I computed in the p[revious part to align it with the camera frame of reference. The transformed point cloud was then converted to a numpy array.

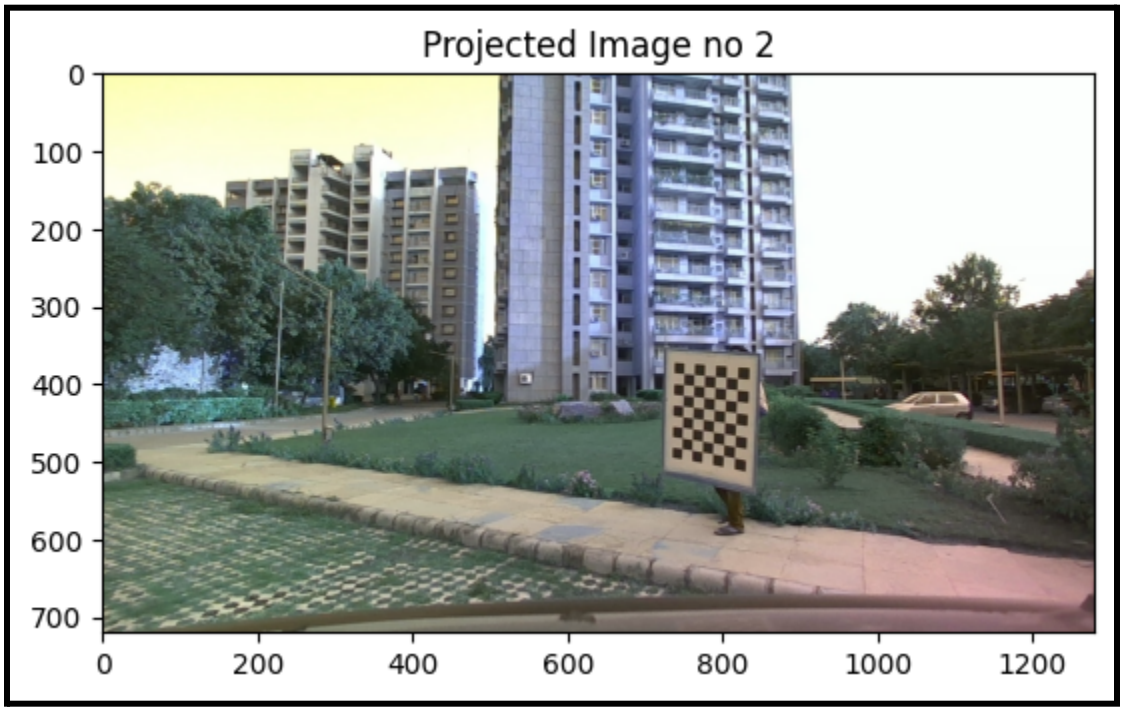
Using the camera intrinsic matrix and distortion coefficients, I projected the LIDAR points onto the 2D image plane. If any point was detected outside the chessboard boundary, a message was printed to the console. Finally, I displayed the image with the projected LIDAR points using the imshow() function from Matplotlib.

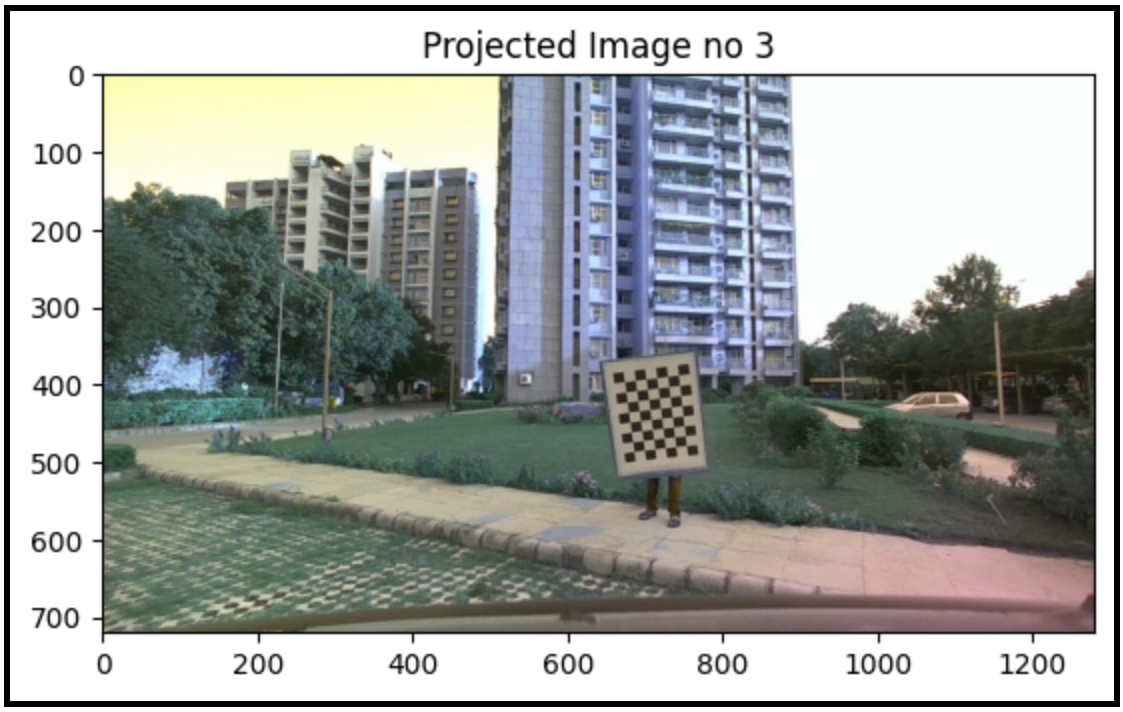
NO,
All the points were not within the checkerboard pattern's boundary in each image, we can see the output also.



Some of the examples of the projected images are shown below:


Projected Image no 1
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200


Projected Image no 2
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200



5.

In this code, I calculated the cosine distances between the camera normal and the transformed LIDAR normals. I started by initializing an empty list called cosine_dis.

I then created a loop that iterates through each camera normal in the camera_normals list. For each iteration, I extracted the rotation matrix from the transformation matrix and applied it to the corresponding LIDAR normal to transform it to the camera frame of reference.

Using the dot product formula, I calculated the cosine distance between the camera normal and the transformed LIDAR normal. The formula involves taking the dot product of the two vectors and dividing it by the product of their magnitudes. The absolute value of this value gives the cosine distance. I added this value to the cosine_dis list.

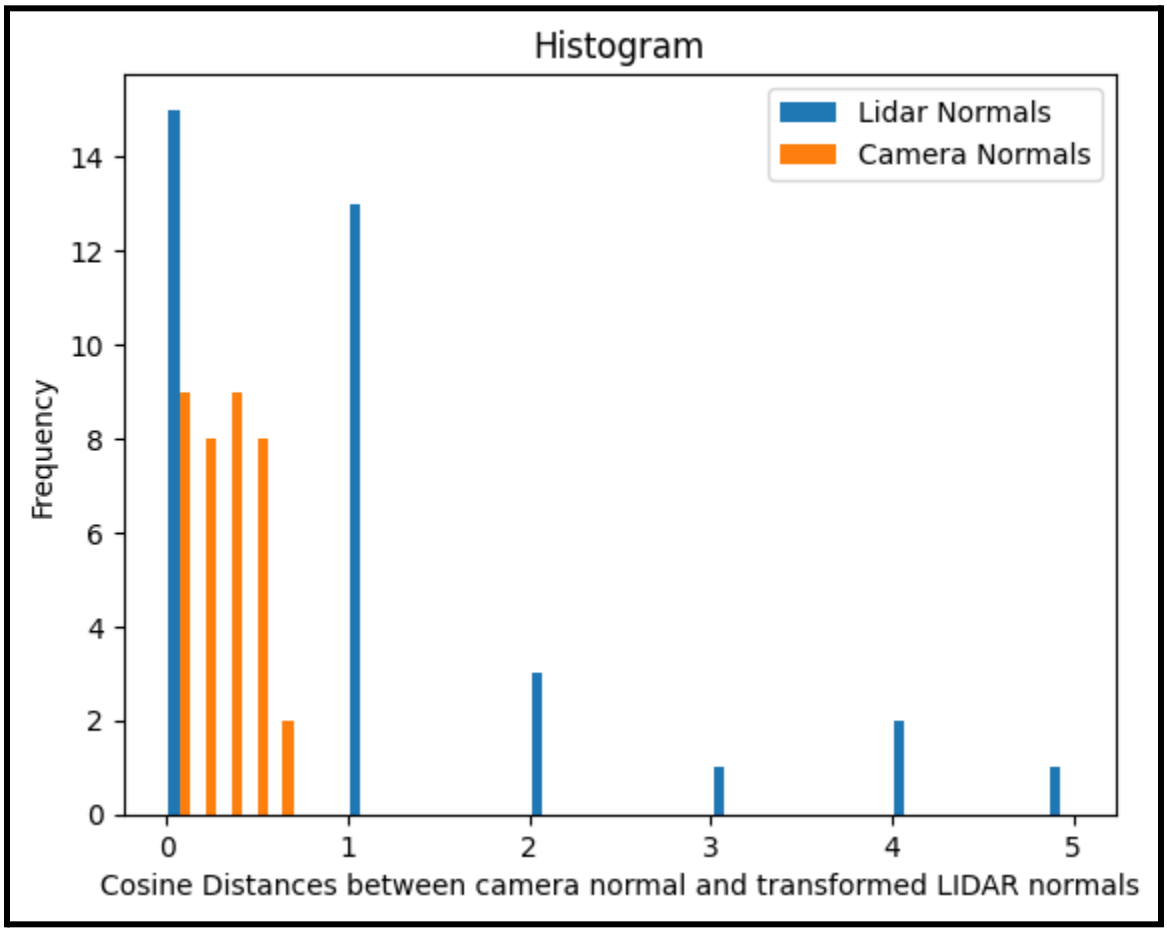
Finally, I printed the cosine distances for each pair of camera and LIDAR normals. The output is a list of floating-point values representing the cosine distances.

List of cosine distances

```
The cosine distances between the camera normal and the transformed LIDAR normals are:

0.3062611806465847
0.3068405528001571
0.08085787172753486
0.31375891517460625
0.27663379291139356
0.09150323949582069
0.5890793720585774
0.45160839171304507
0.060337929438914334
0.15302439956254194
0.30384796938358166
0.25538759179283094
0.14264071885955754
0.22569207773557365
0.25773992750419605
0.10408348903100254
0.135333837784945
0.2299800679452777
0.21956368740266796
0.48763828458178965
0.01115013950446425
0.2911300272426168
0.30378714621181624
0.3049485101525349
0.00230345653669401
0.30309031953450705
0.3316172549378362
0.15911284998401498
0.054009146401270774
0.17950633200599594
0.15064339406797567
```

Histogram of errors



The average and standard deviation