```
import numpy as np
import cv2 as cv
import glob
import matplotlib.pyplot as plt
import open3d as o3d
from google.colab import drive
drive.mount('/content/drive')
```

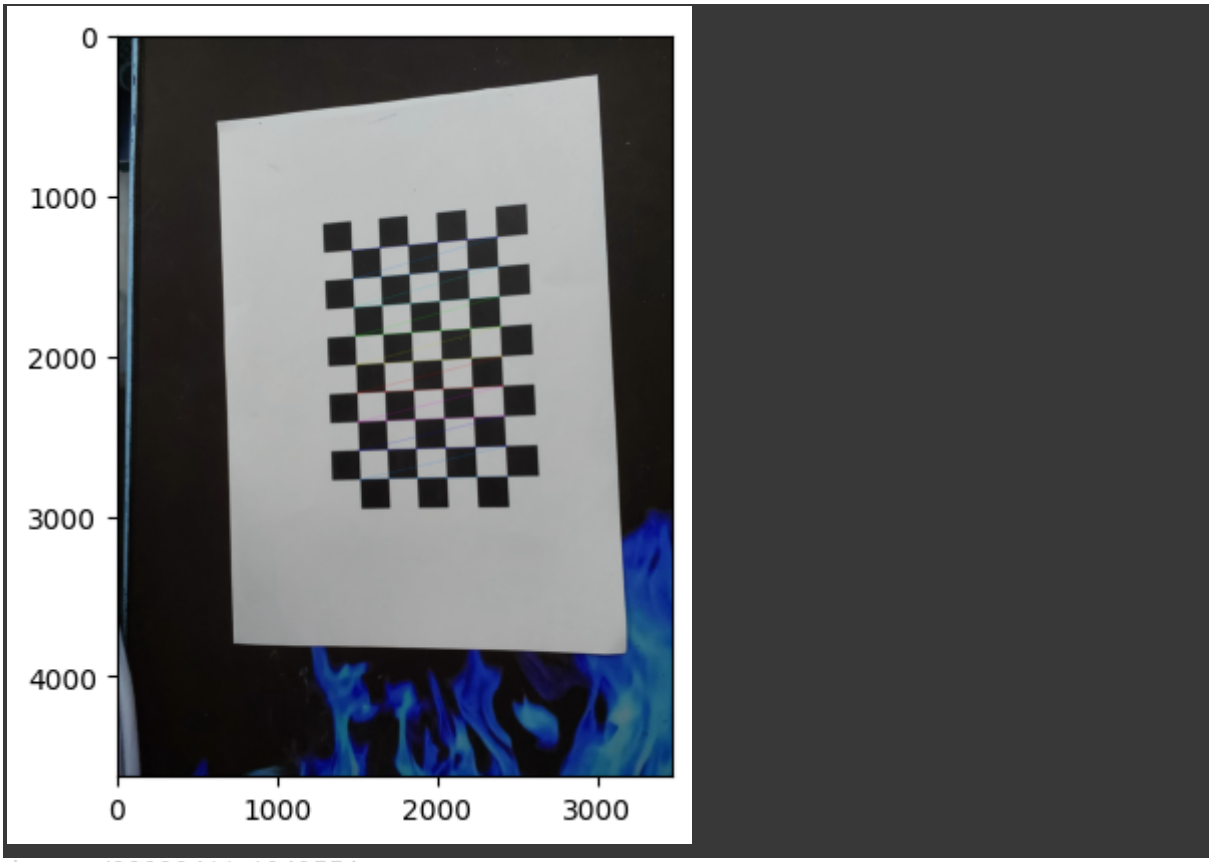
Importing the necessary libraries.

Q1

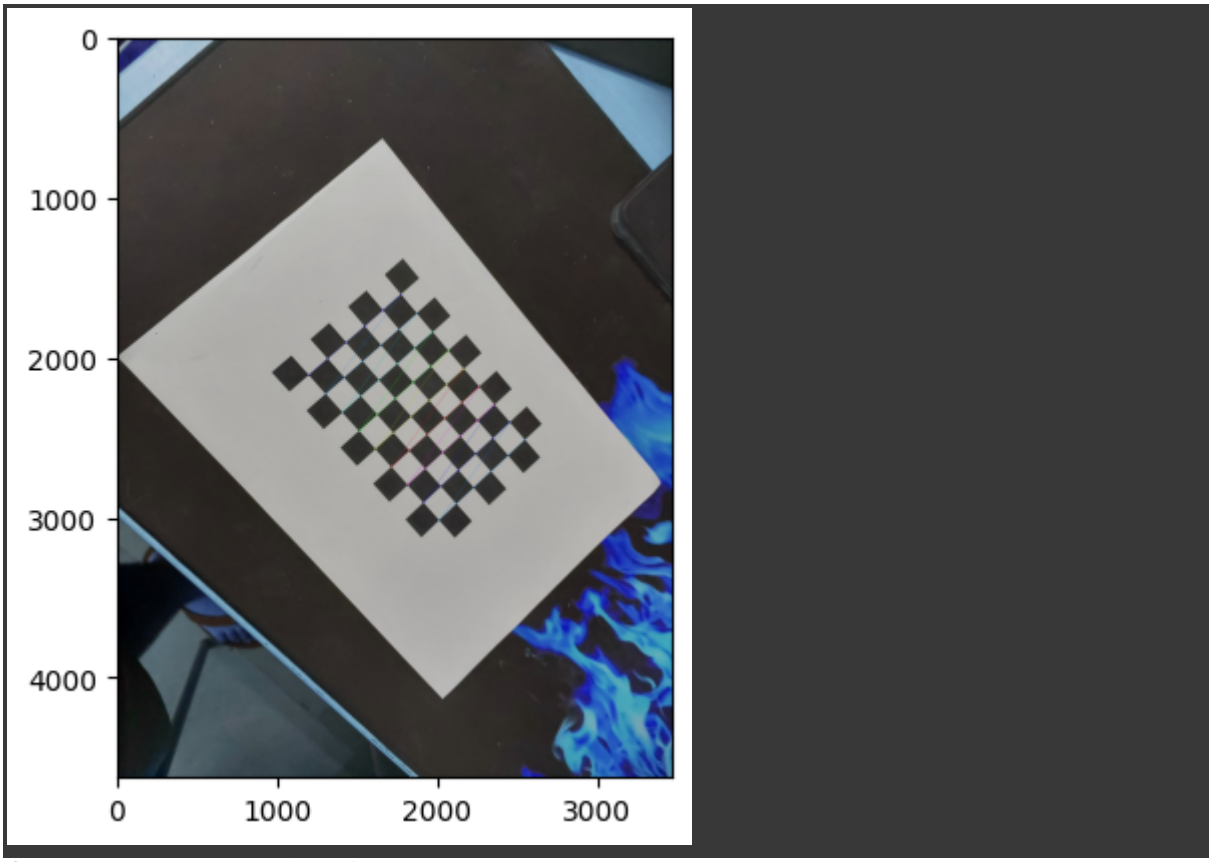
```
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30,
0.001)
objp = np.zeros((6*9,3), np.float32)
objp[:,:2] = np.mgrid[0:6,0:9].T.reshape(-1,2)
objpoints = []
imgpoints = []
n=1
images = glob.glob('data_cv/*.jpg')
for fname in images:
    print(fname)
    img = cv.imread(fname)
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    ret, corners = cv.findChessboardCorners(gray, (6,9), None)
    print("Image", n, "Processed")
    if ret == True:
        print("Corners found for image", n)
        objpoints.append(objp)
        corners2 = cv.cornerSubPix(gray, corners, (11,11), (-1,-1),
criteria)
        imgpoints.append(corners2)
        cv.drawChessboardCorners(img, (6,9), corners2, ret)
    plt.imshow(img)
    plt.show()
    n+=1
```

Defining the criterion, collecting the images from the data folder using glob.
Inside the loop, finding if there are chessboard corners or not. If found. Storing the points, then the corners using cornerSubpix(). Finally, highlighting the chessboard on the image.

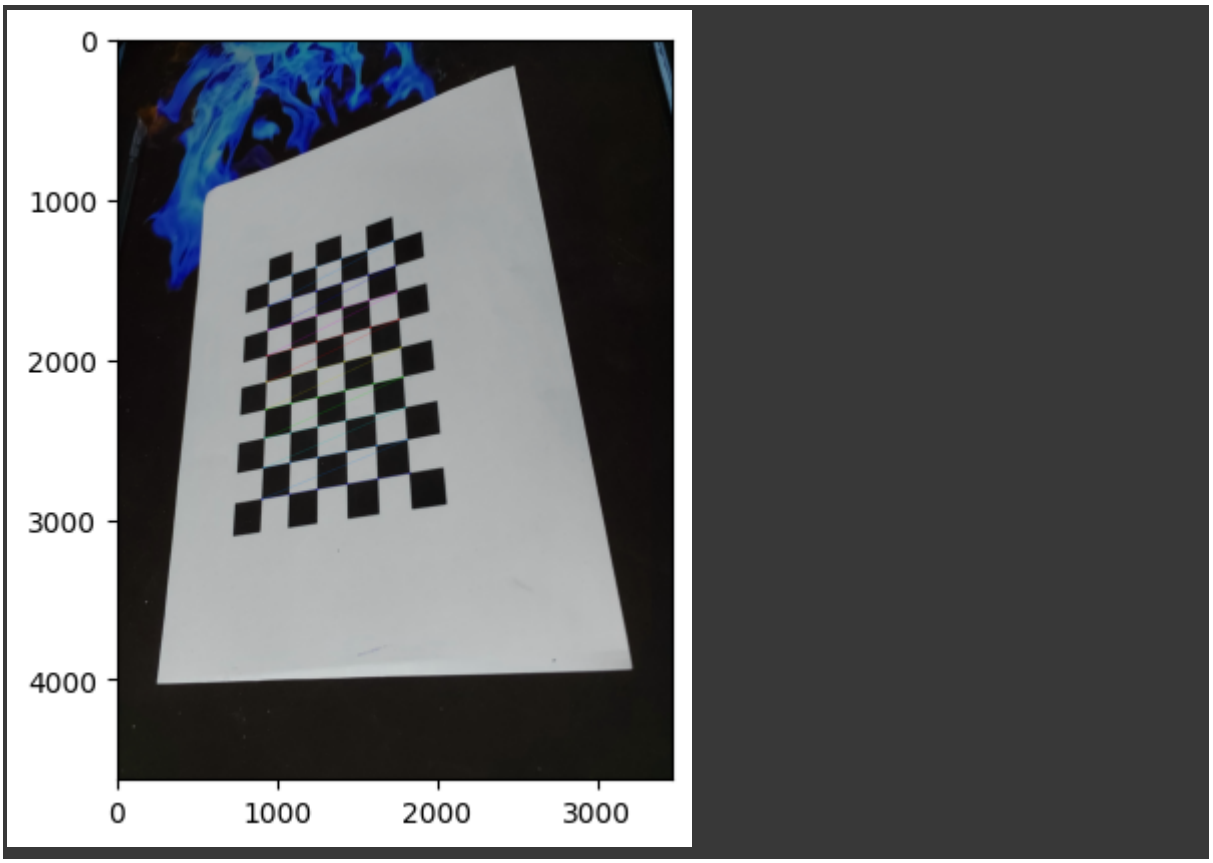
```
data_cv/20230411_184548.jpg
Image 1 Processed
Corners found for image 1
```



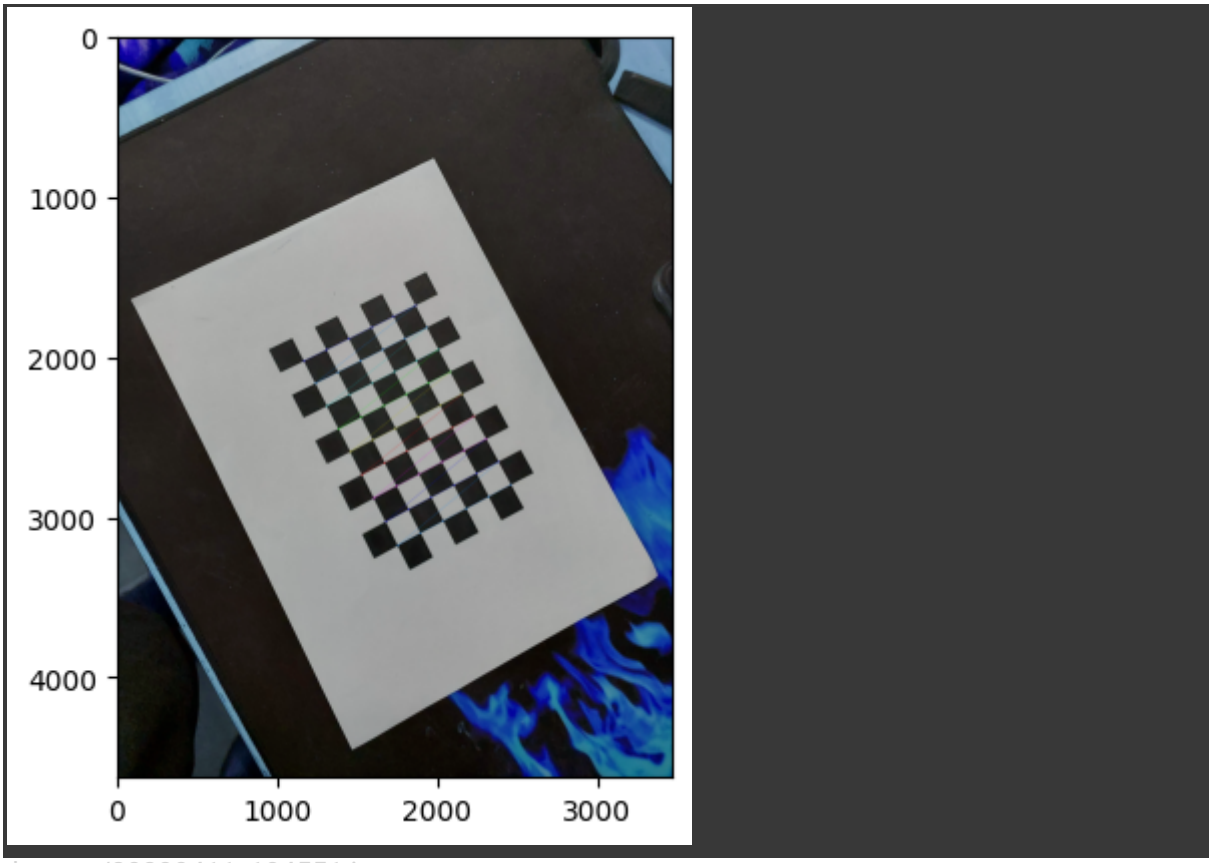
data_cv/20230411_184855.jpg
Image 2 Processed
Corners found for image 2



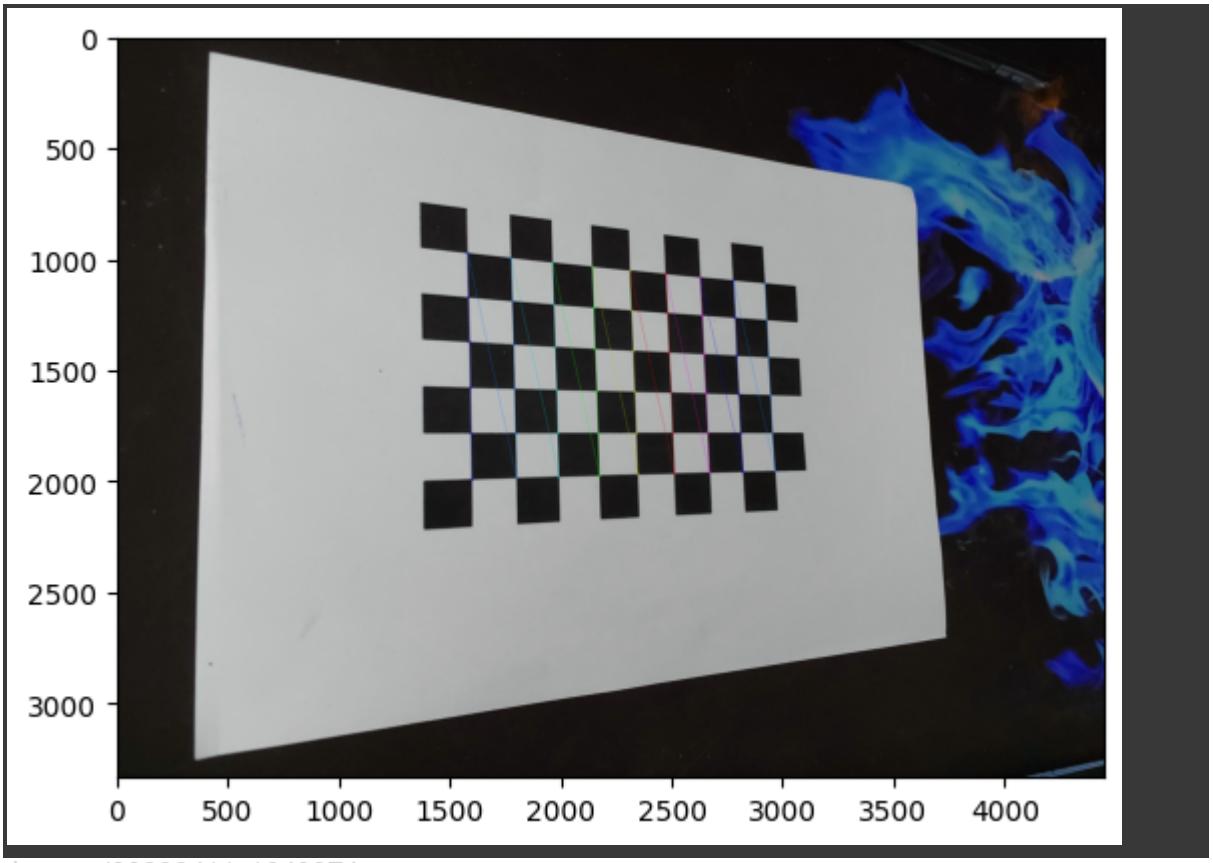
data_cv/20230411_184601.jpg
Image 3 Processed
Corners found for image 3



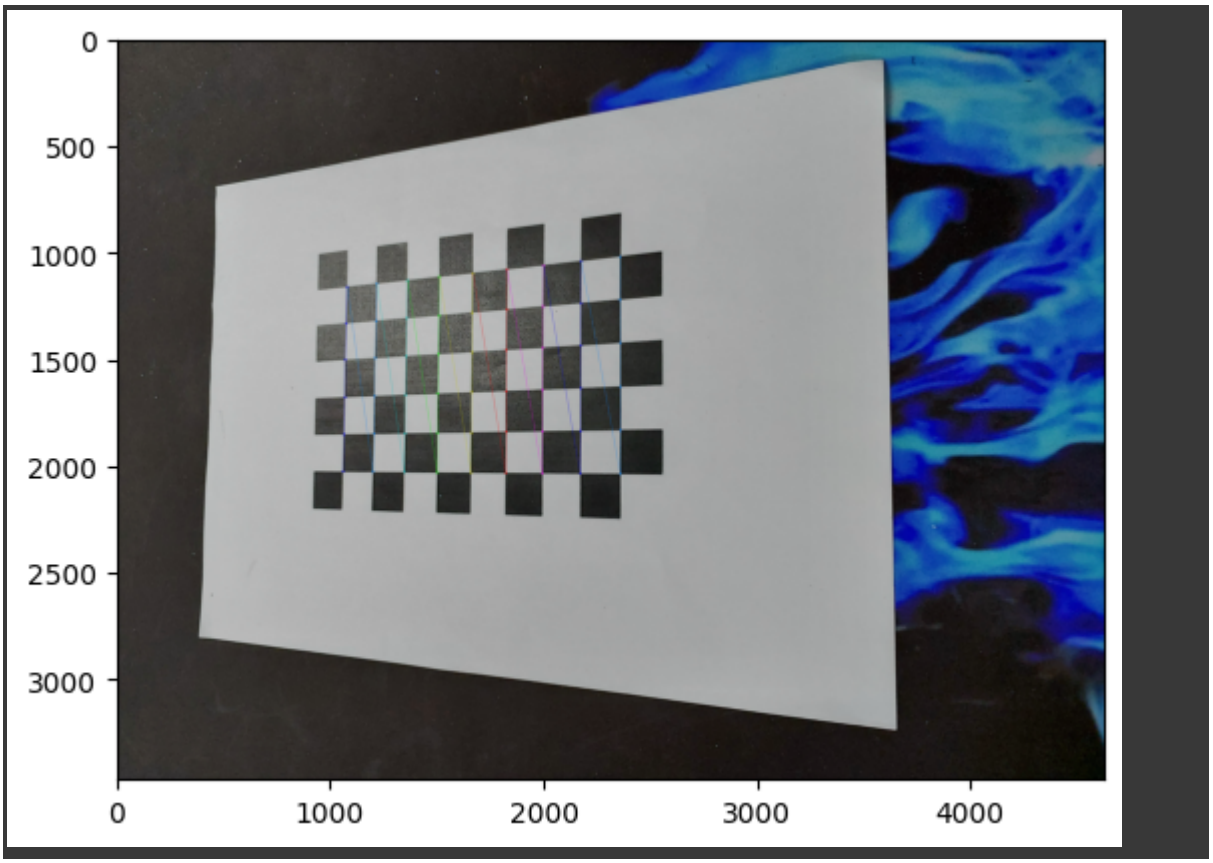
data_cv/20230411_184821.jpg
Image 4 Processed
Corners found for image 4



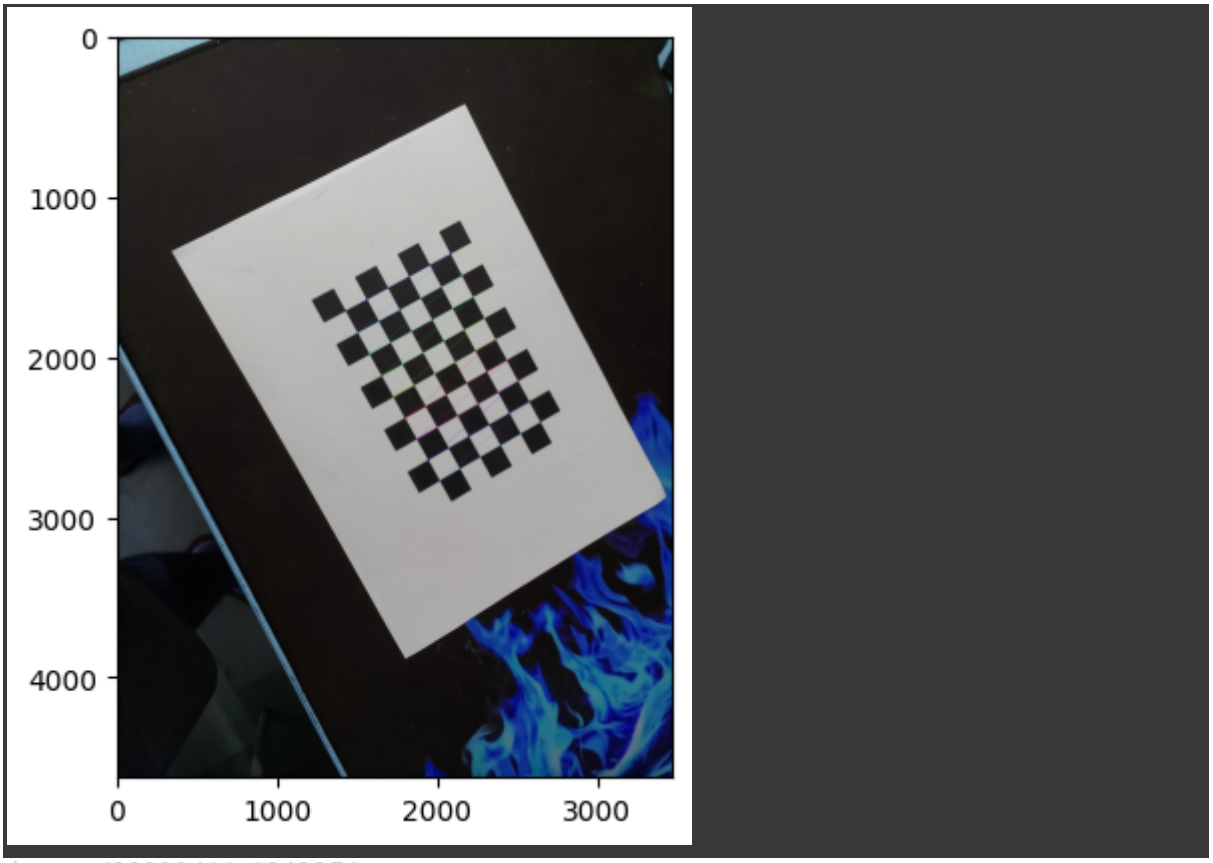
data_cv/20230411_184551.jpg
Image 5 Processed
Corners found for image 5



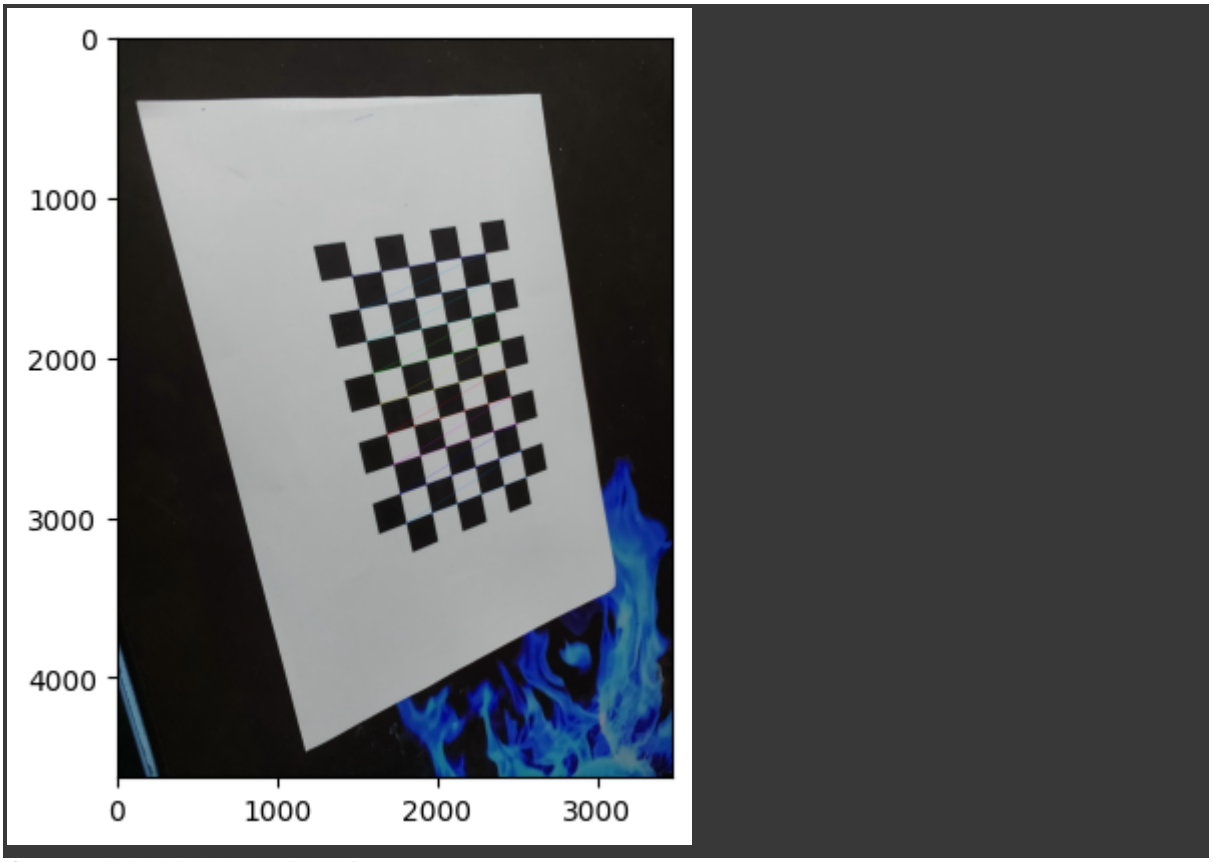
data_cv/20230411_184937.jpg
Image 6 Processed
Corners found for image 6



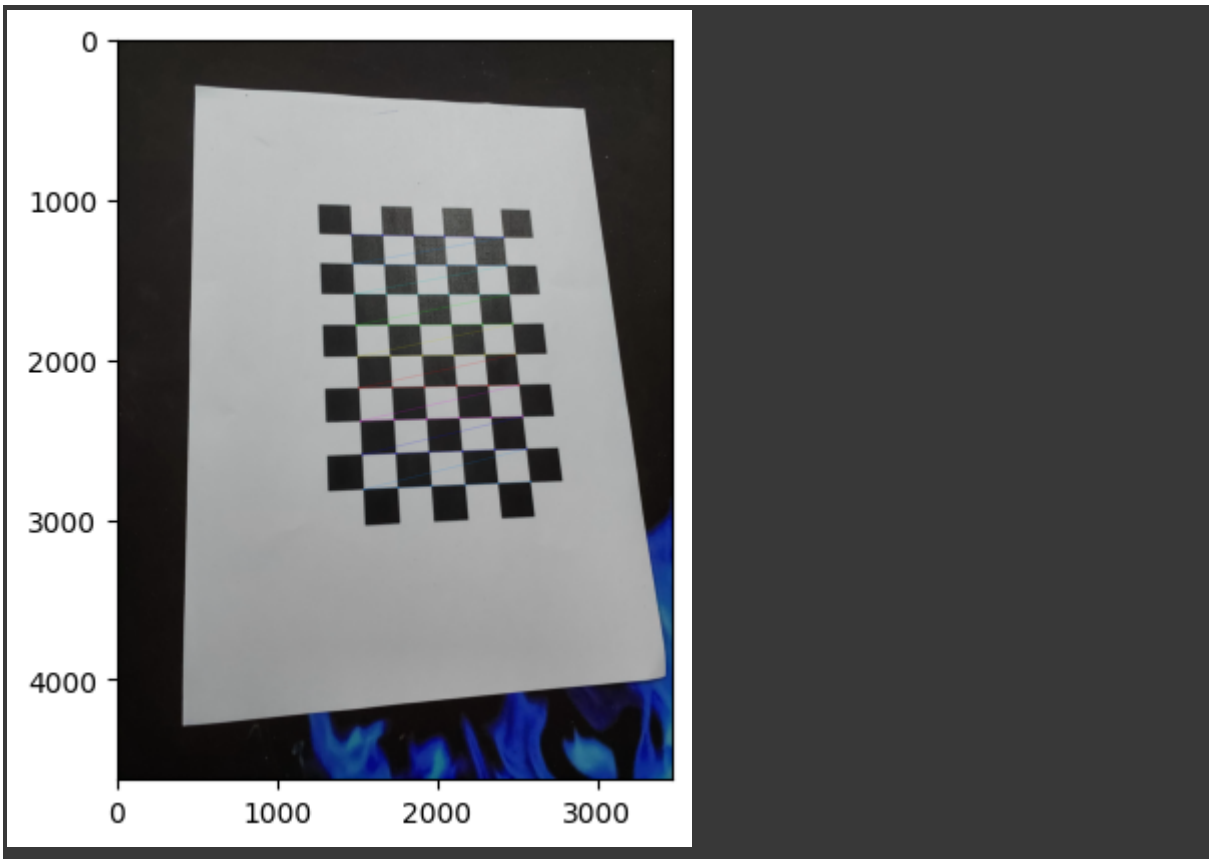
data_cv/20230411_184633.jpg
Image 7 Processed
Corners found for image 7



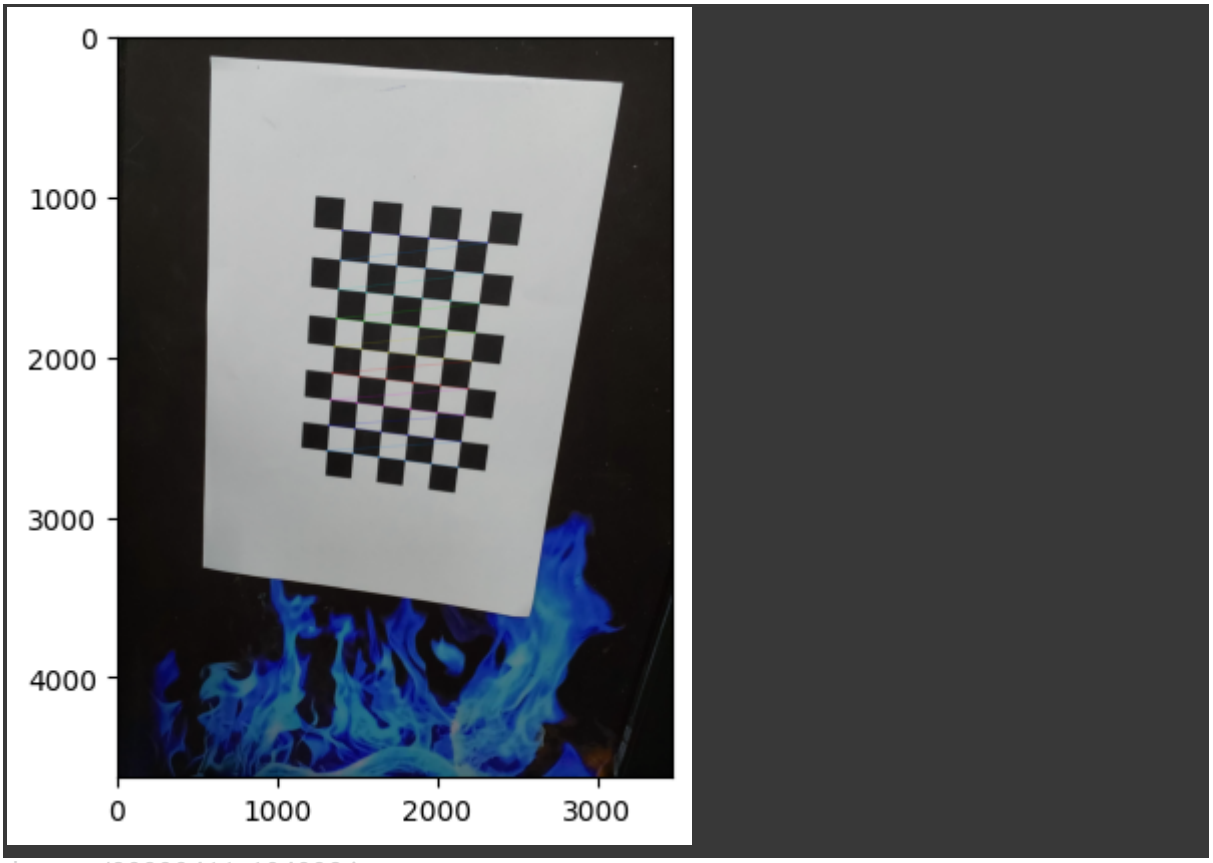
data_cv/20230411_184325.jpg
Image 8 Processed
Corners found for image 8



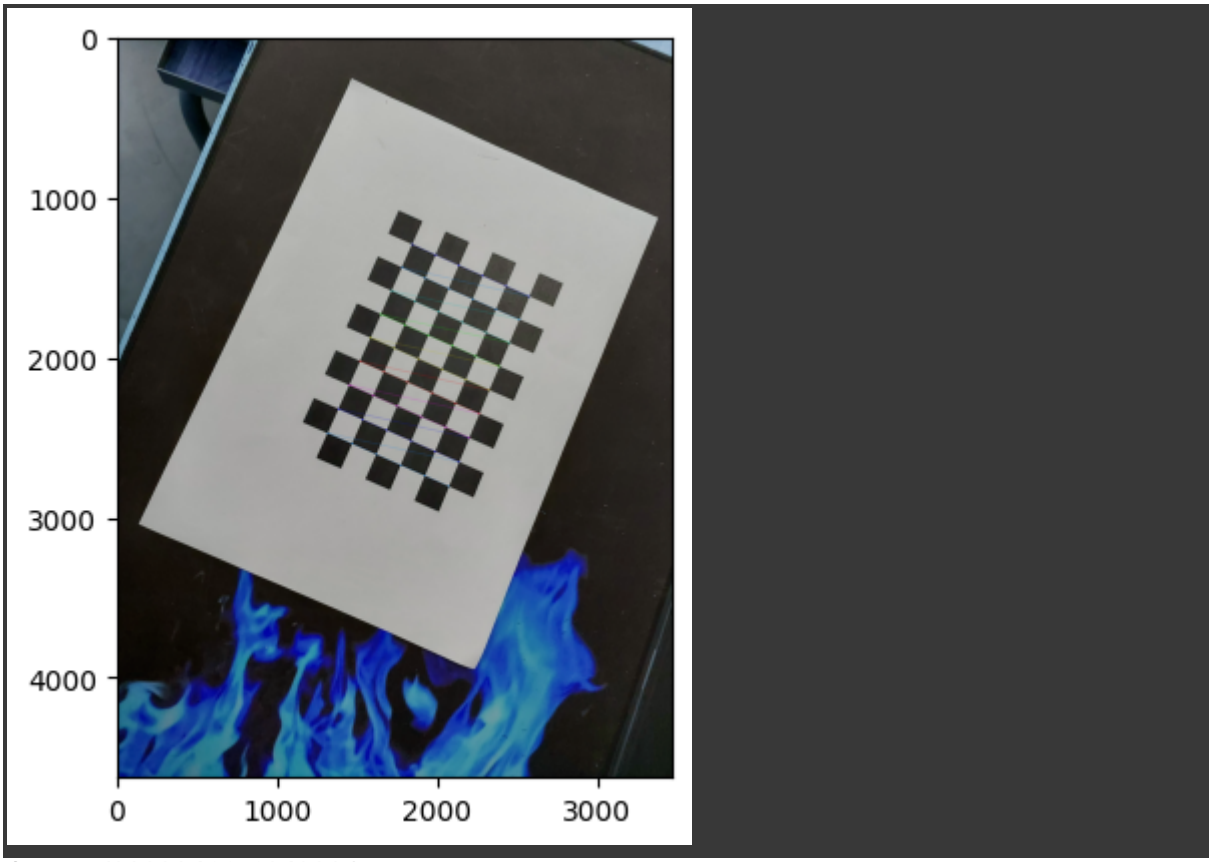
data_cv/20230411_184318.jpg
Image 9 Processed
Corners found for image 9



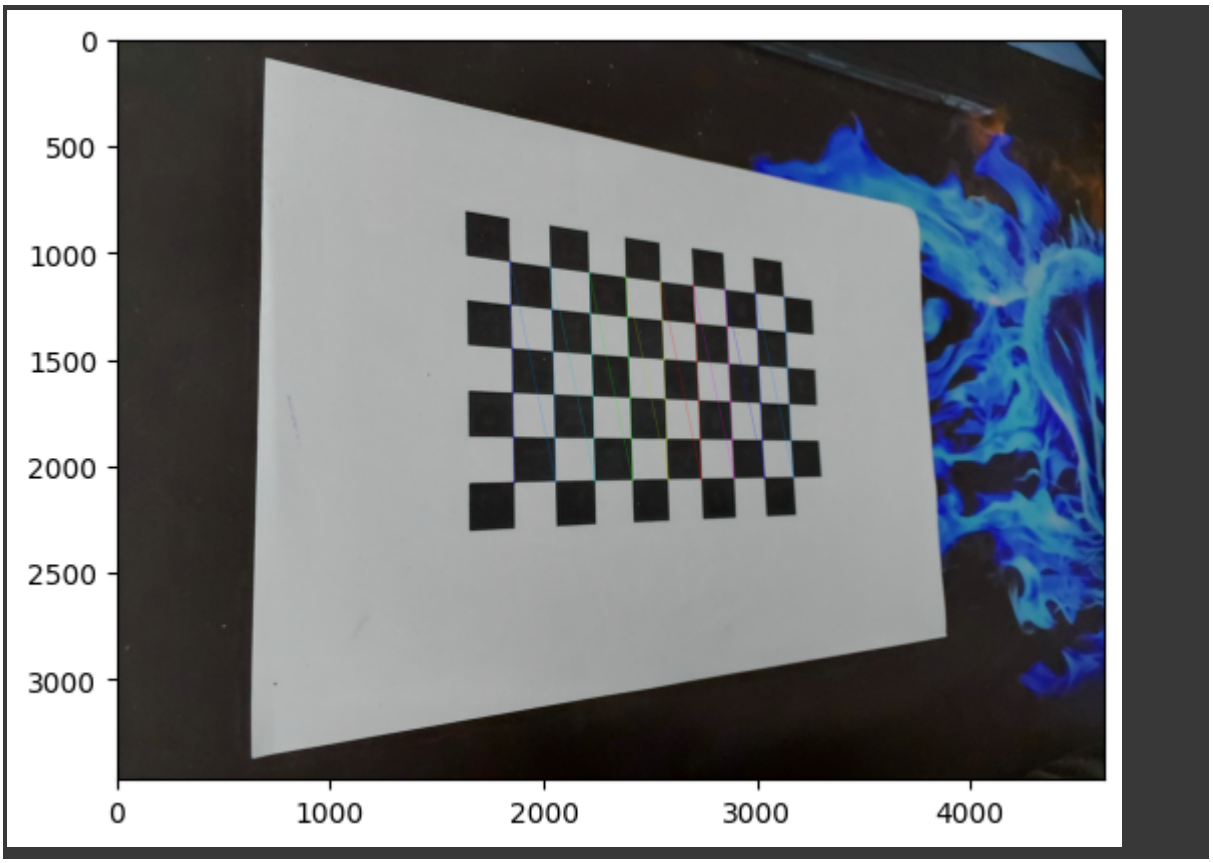
data_cv/20230411_184502.jpg
Image 10 Processed
Corners found for image 10



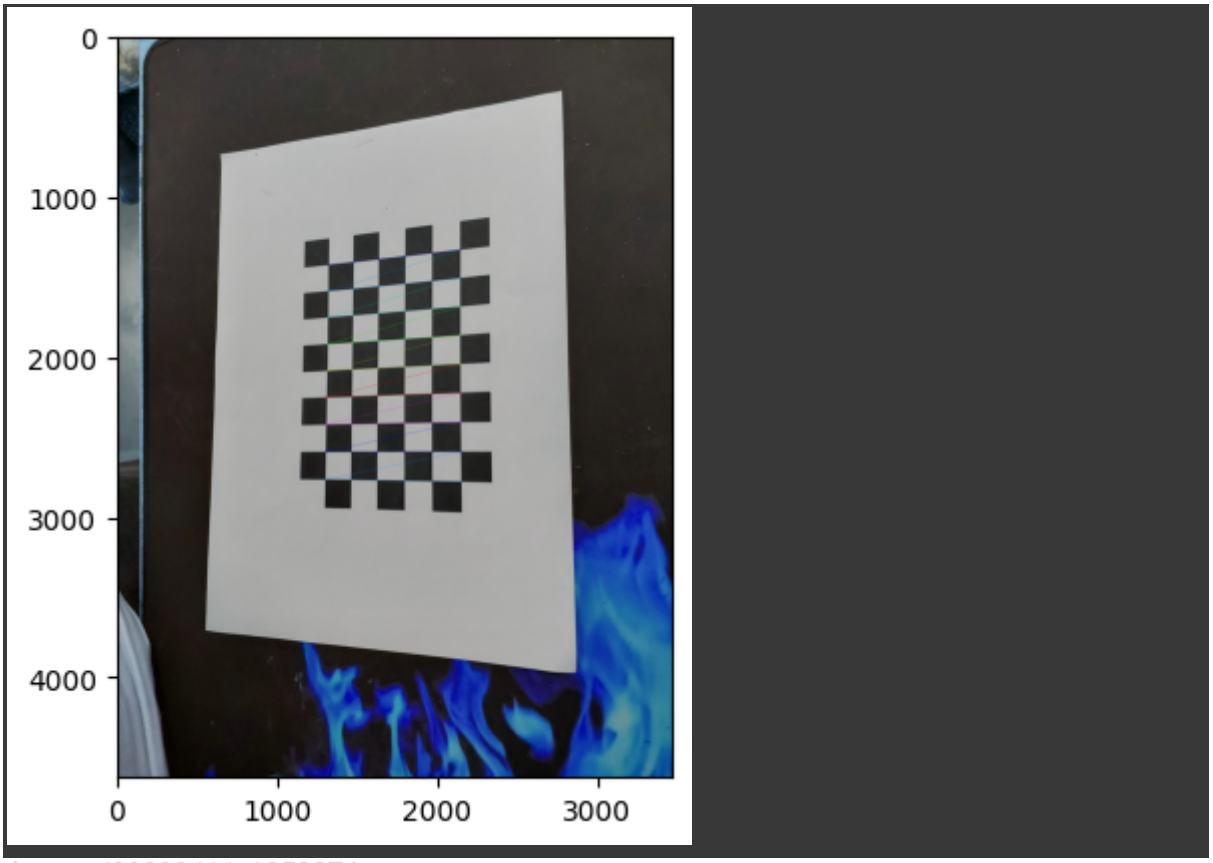
data_cv/20230411_184832.jpg
Image 11 Processed
Corners found for image 11



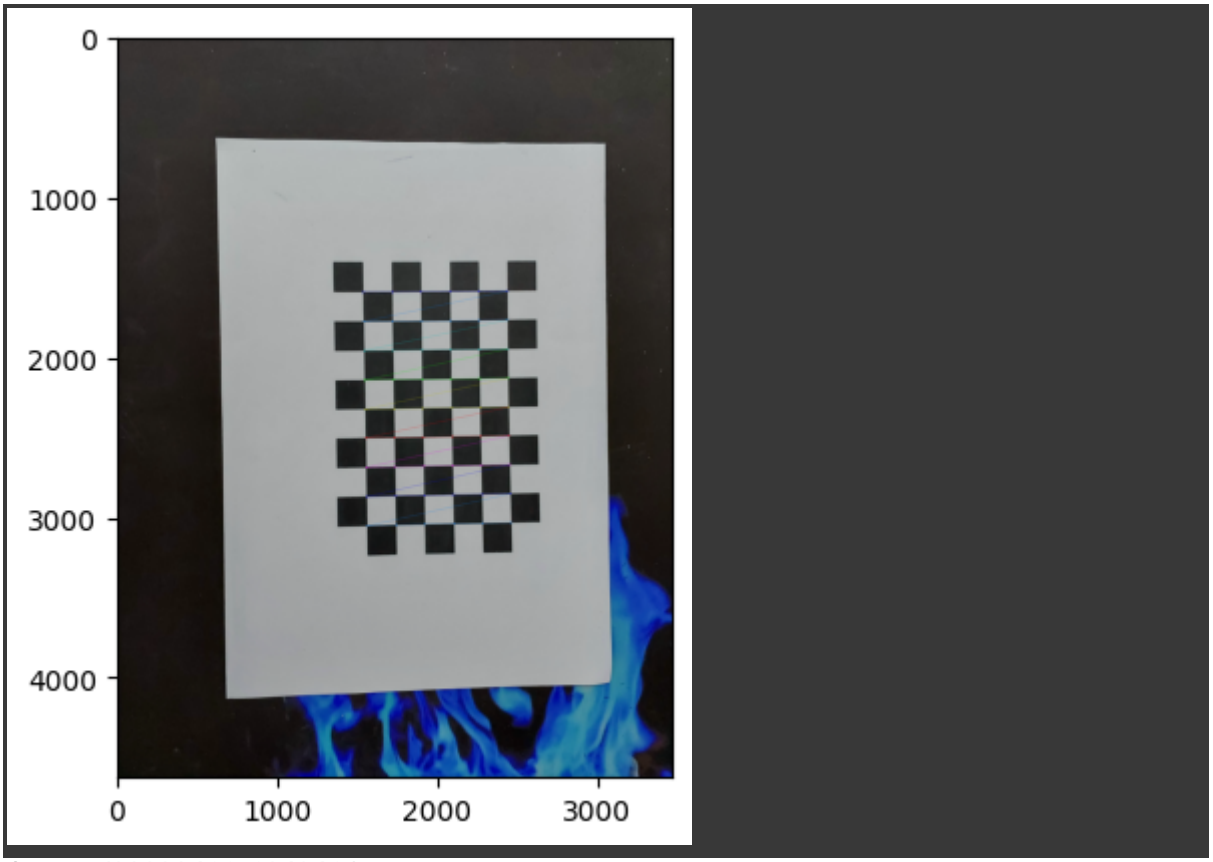
data_cv/20230411_184930.jpg
Image 12 Processed
Corners found for image 12



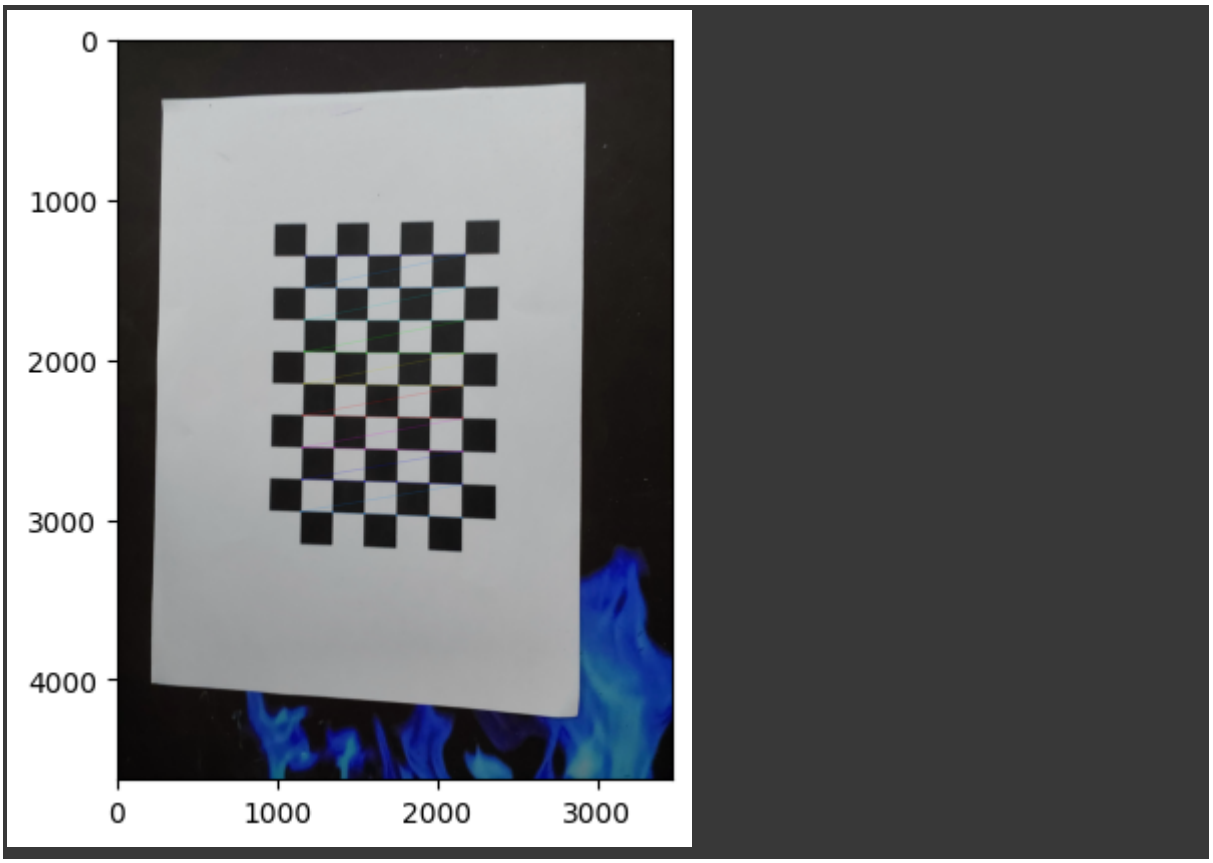
data_cv/20230411_184839.jpg
Image 13 Processed
Corners found for image 13



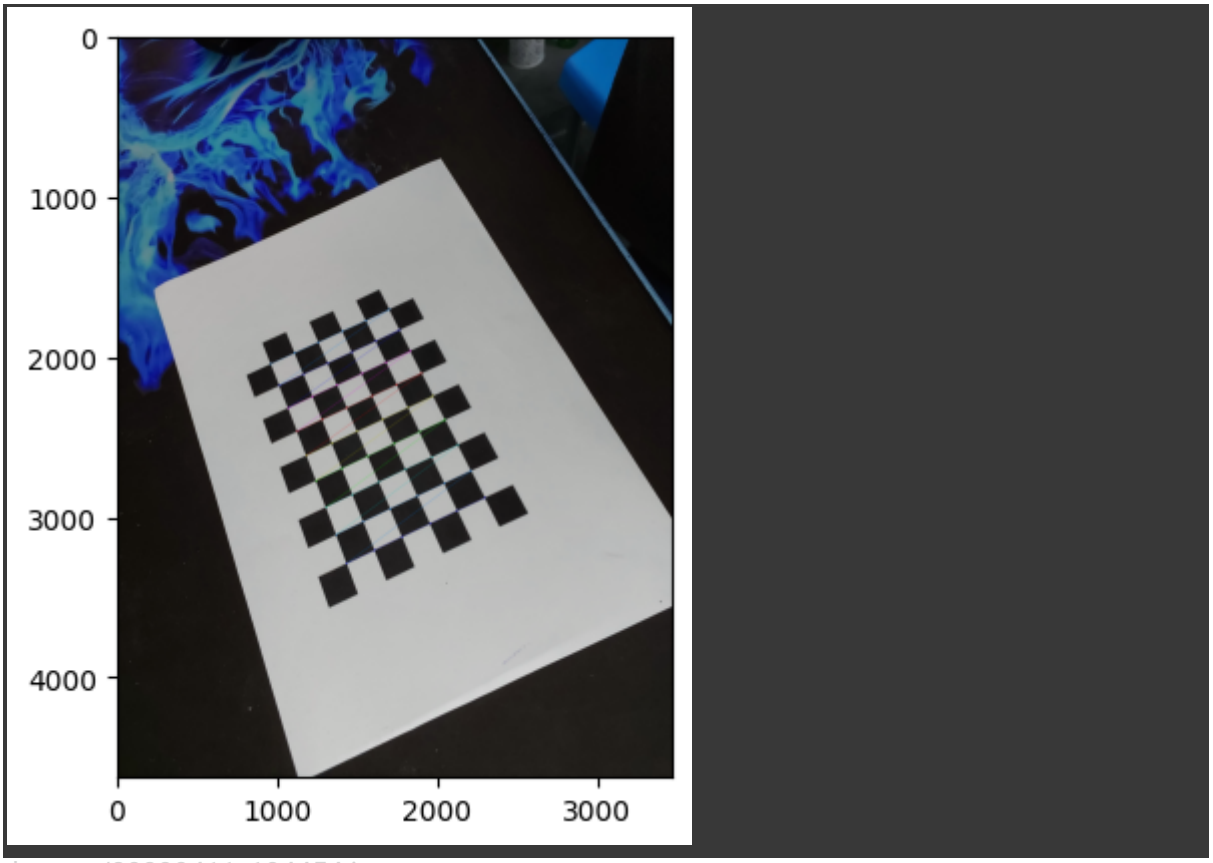
data_cv/20230411_185007.jpg
Image 14 Processed
Corners found for image 14



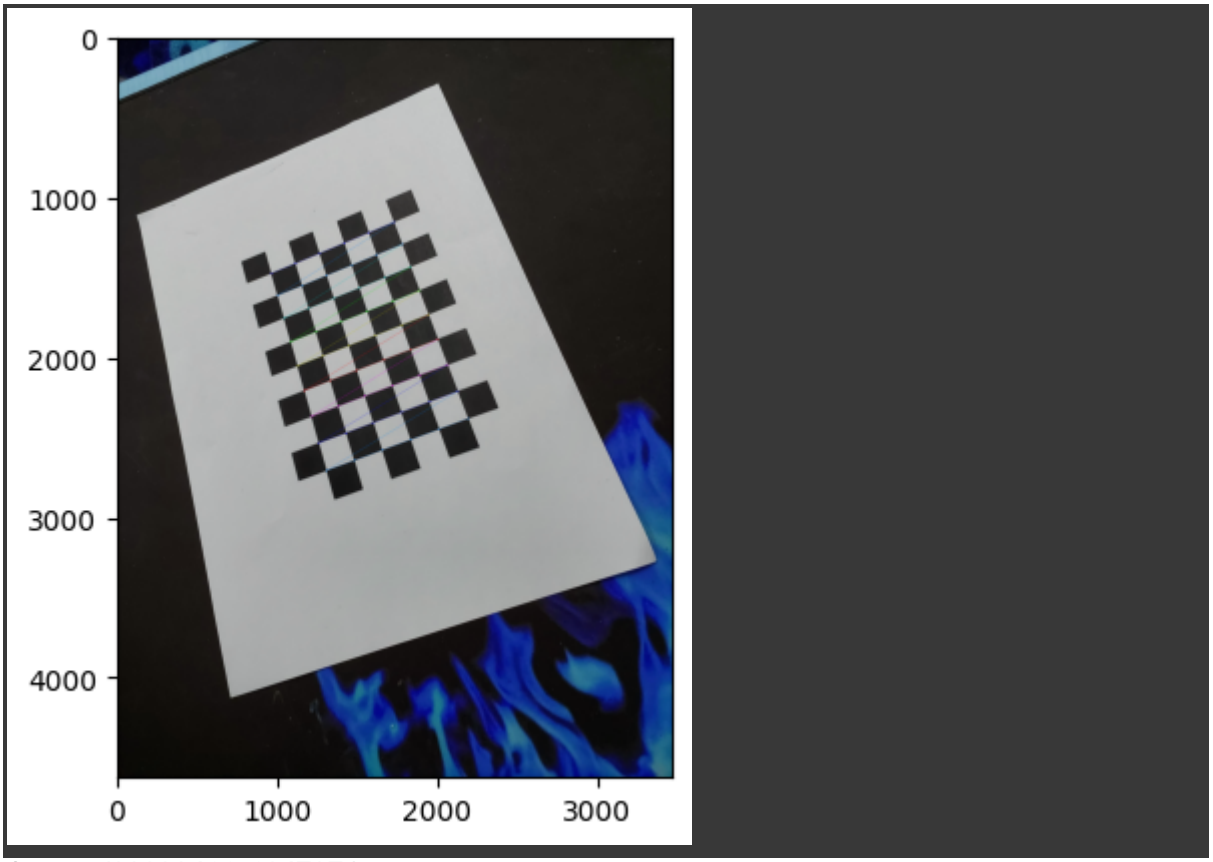
data_cv/20230411_184341.jpg
Image 15 Processed
Corners found for image 15



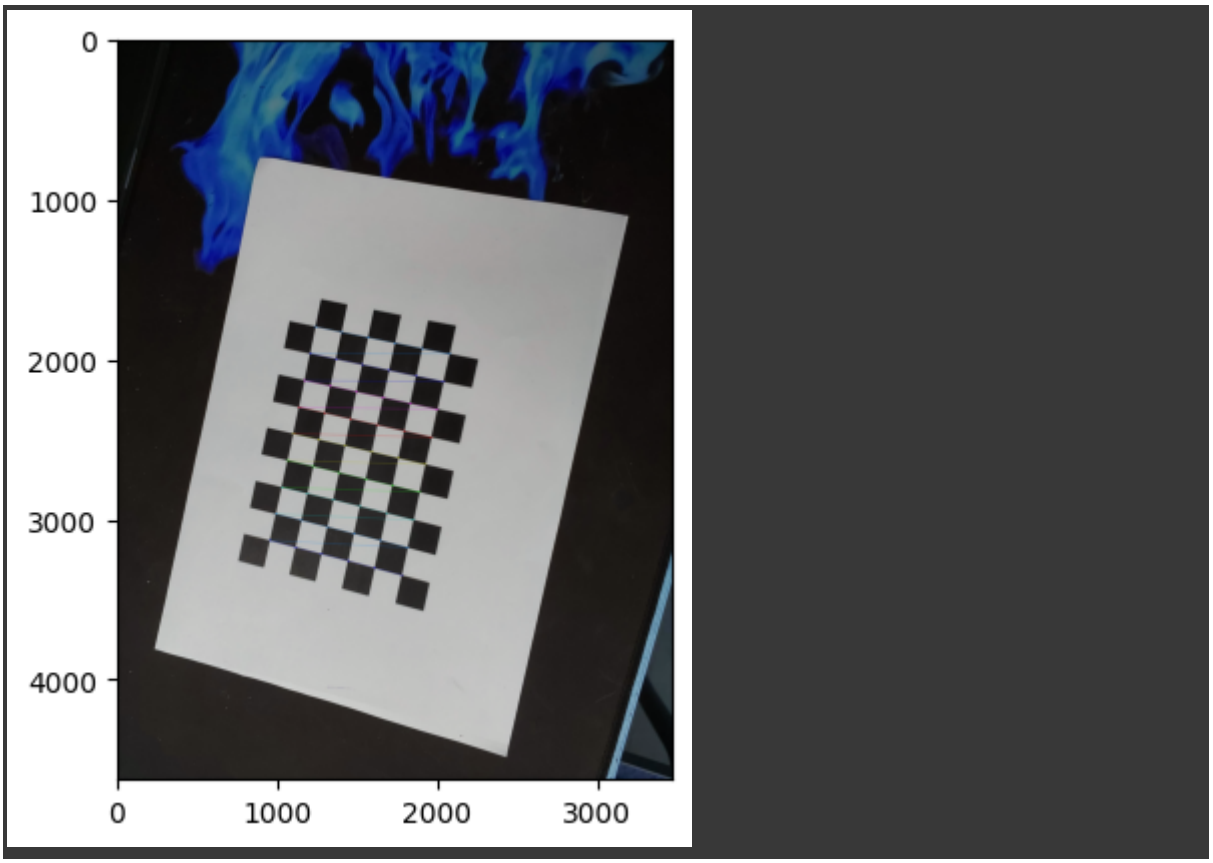
data_cv/20230411_184439.jpg
Image 16 Processed
Corners found for image 16



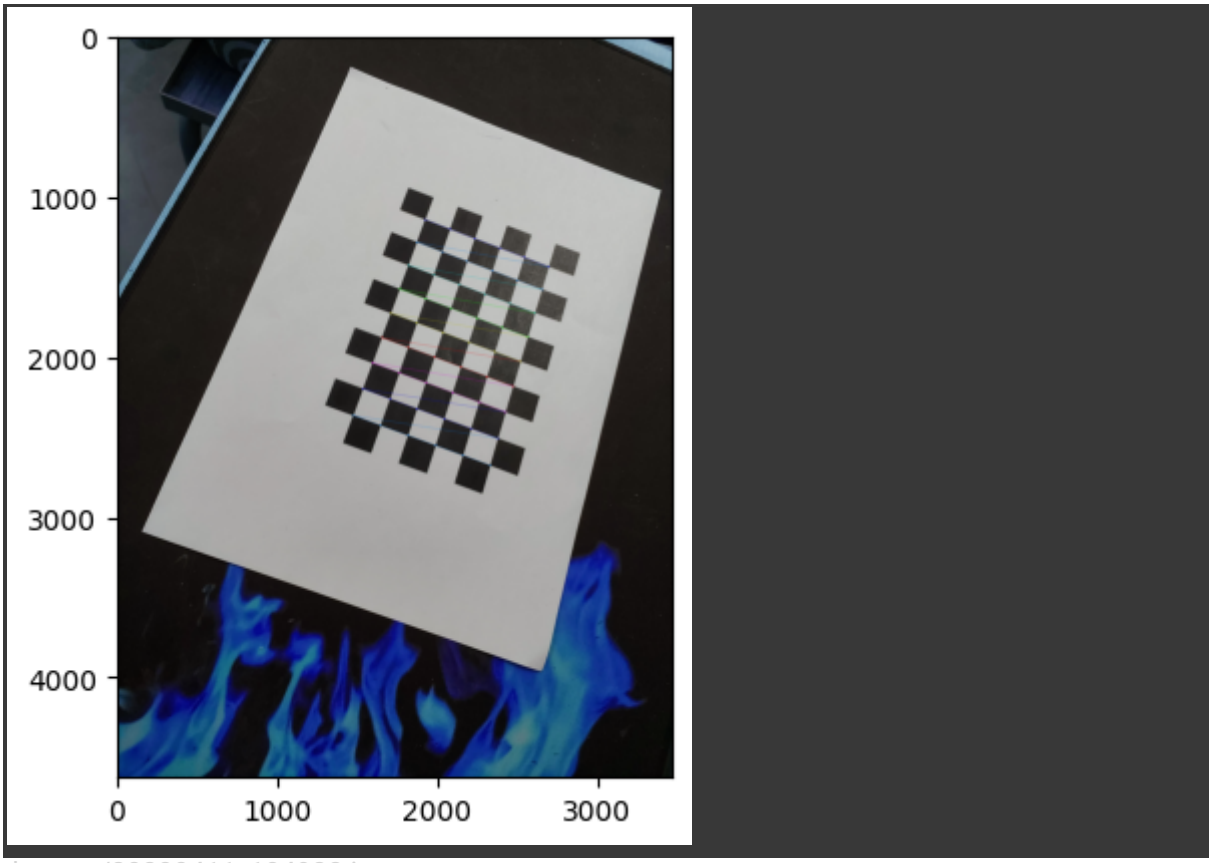
data_cv/20230411_184454.jpg
Image 17 Processed
Corners found for image 17



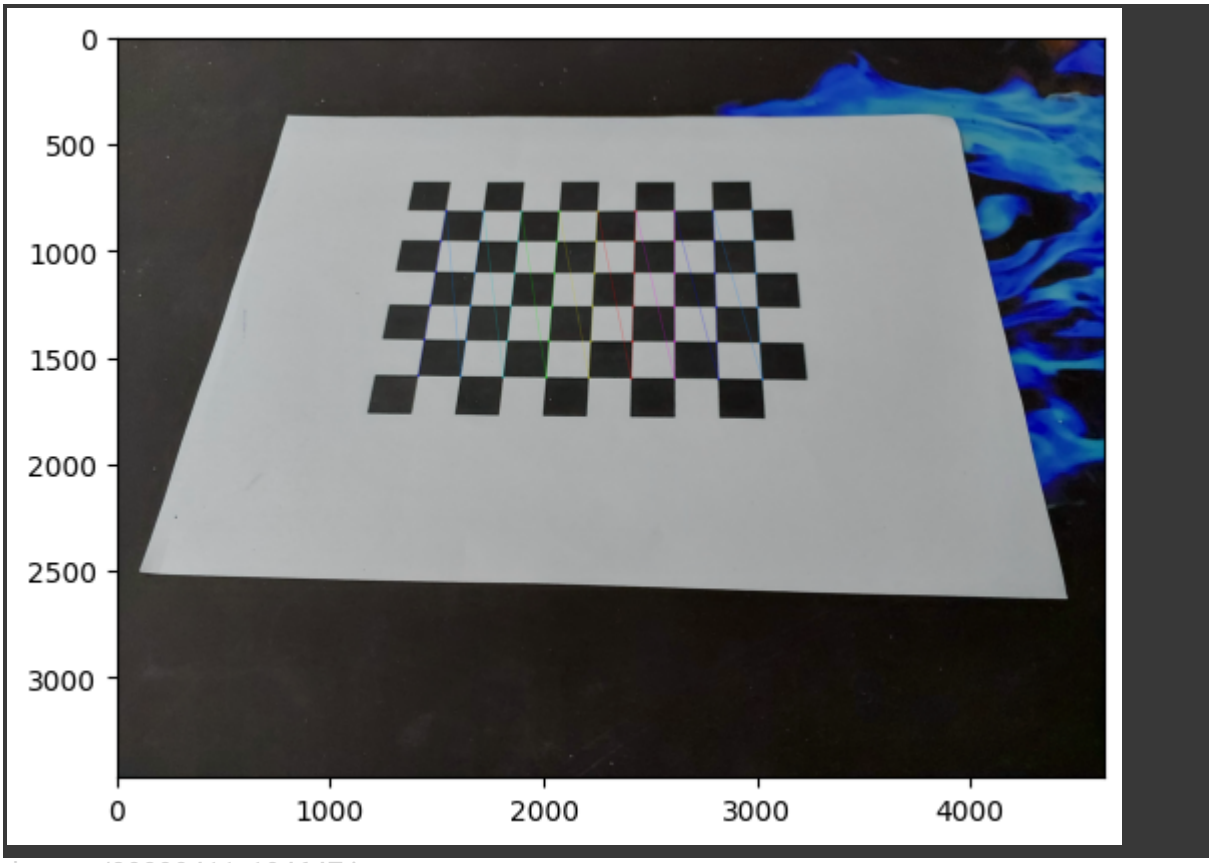
data_cv/20230411_184707.jpg
Image 18 Processed
Corners found for image 18



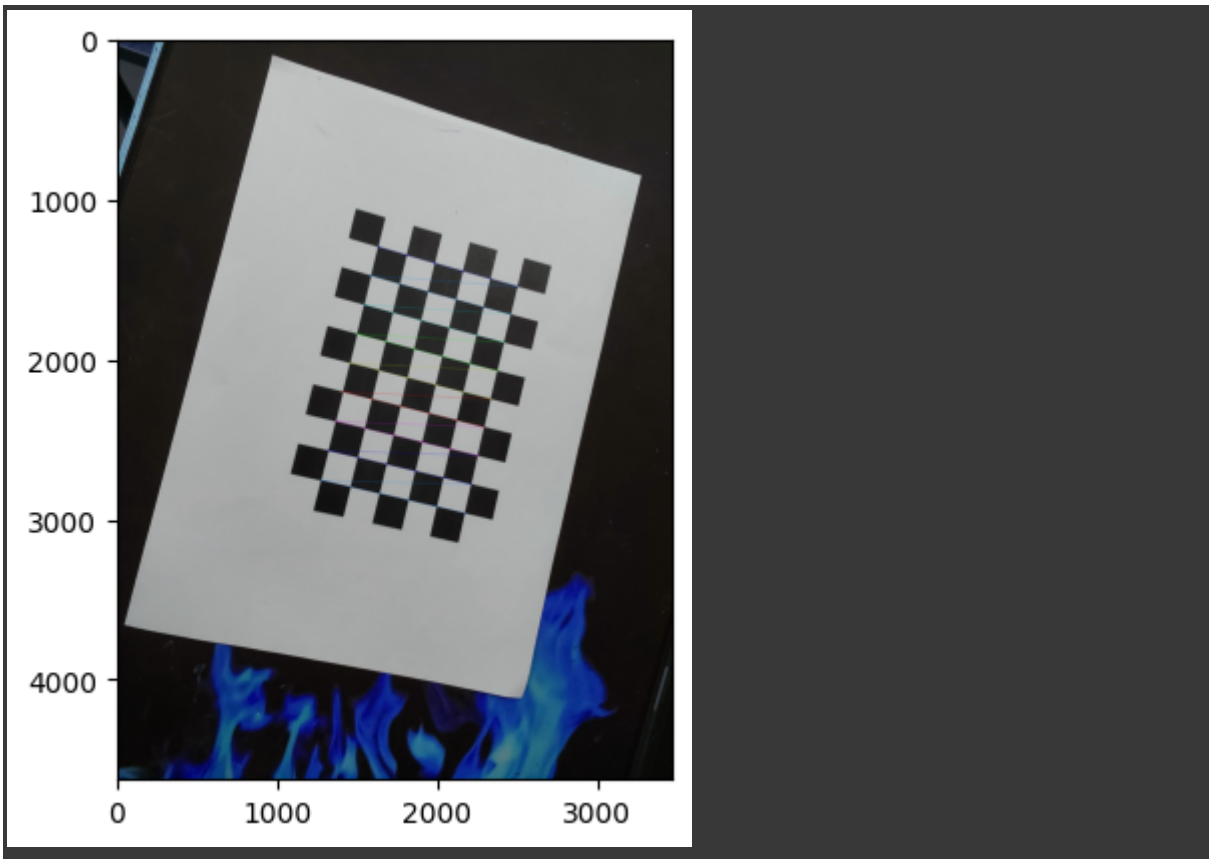
data_cv/20230411_184620.jpg
Image 19 Processed
Corners found for image 19



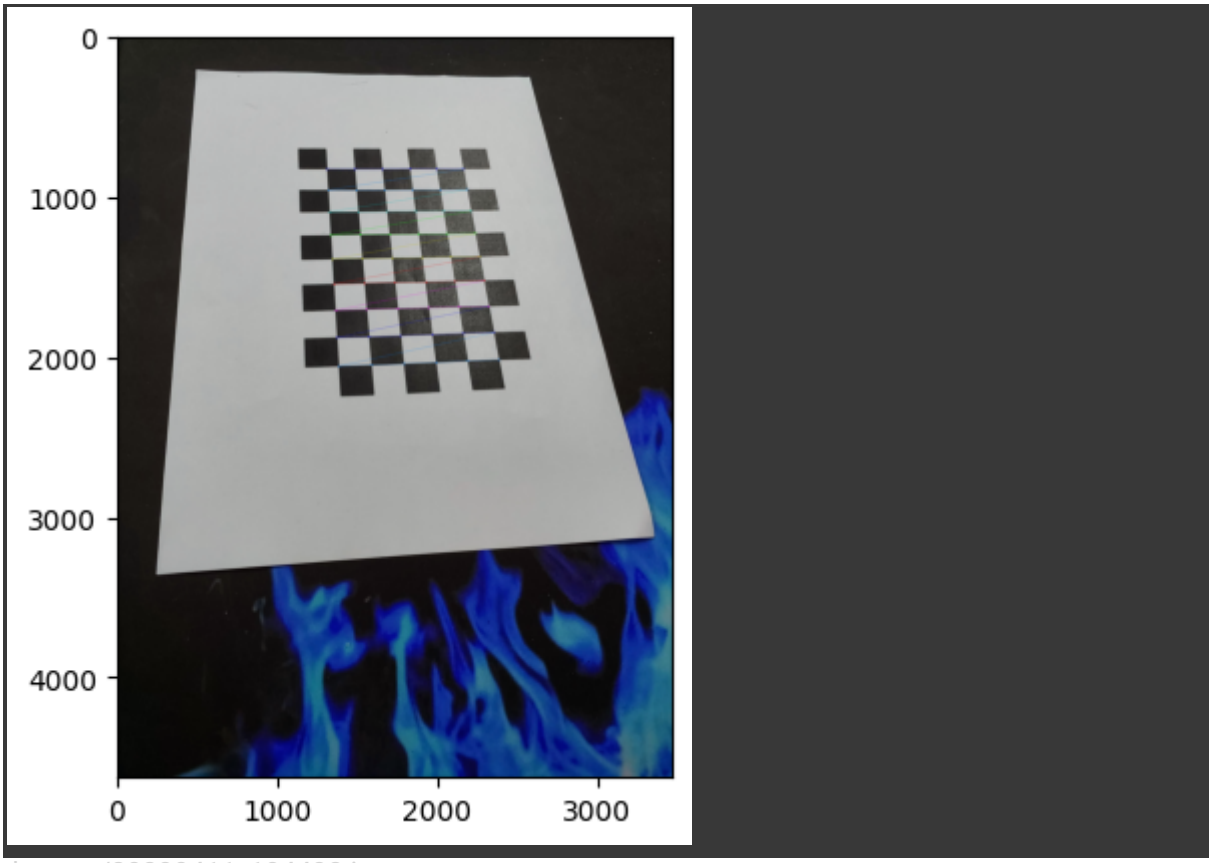
data_cv/20230411_184922.jpg
Image 20 Processed
Corners found for image 20



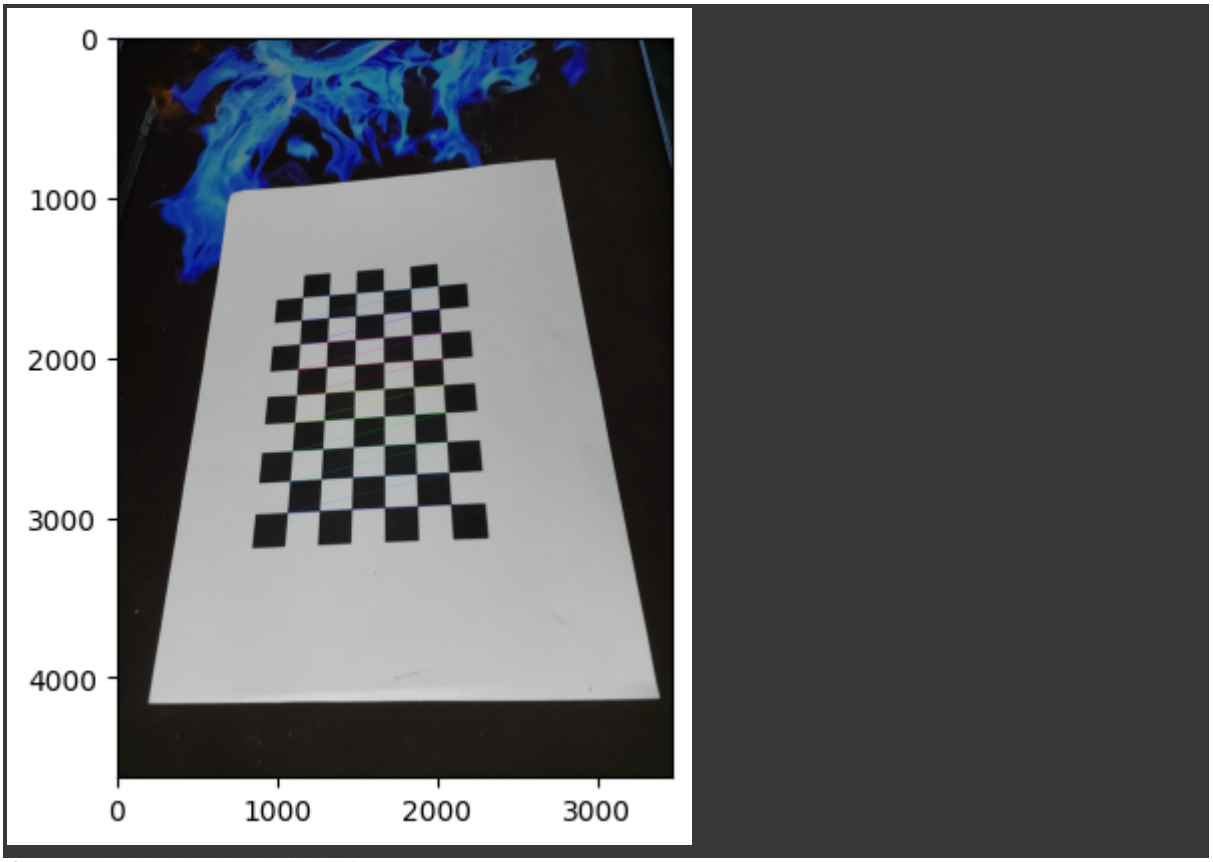
data_cv/20230411_184647.jpg
Image 21 Processed
Corners found for image 21



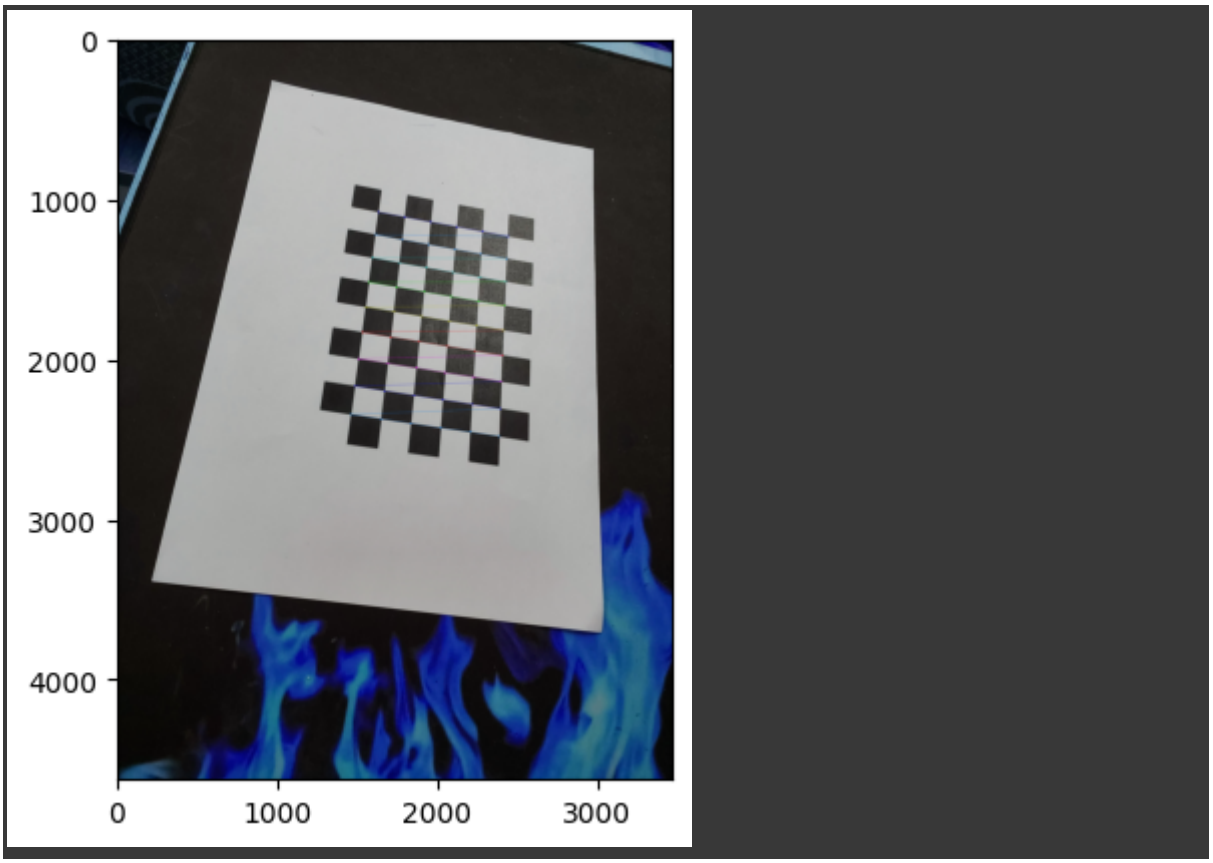
data_cv/20230411_184540.jpg
Image 22 Processed
Corners found for image 22



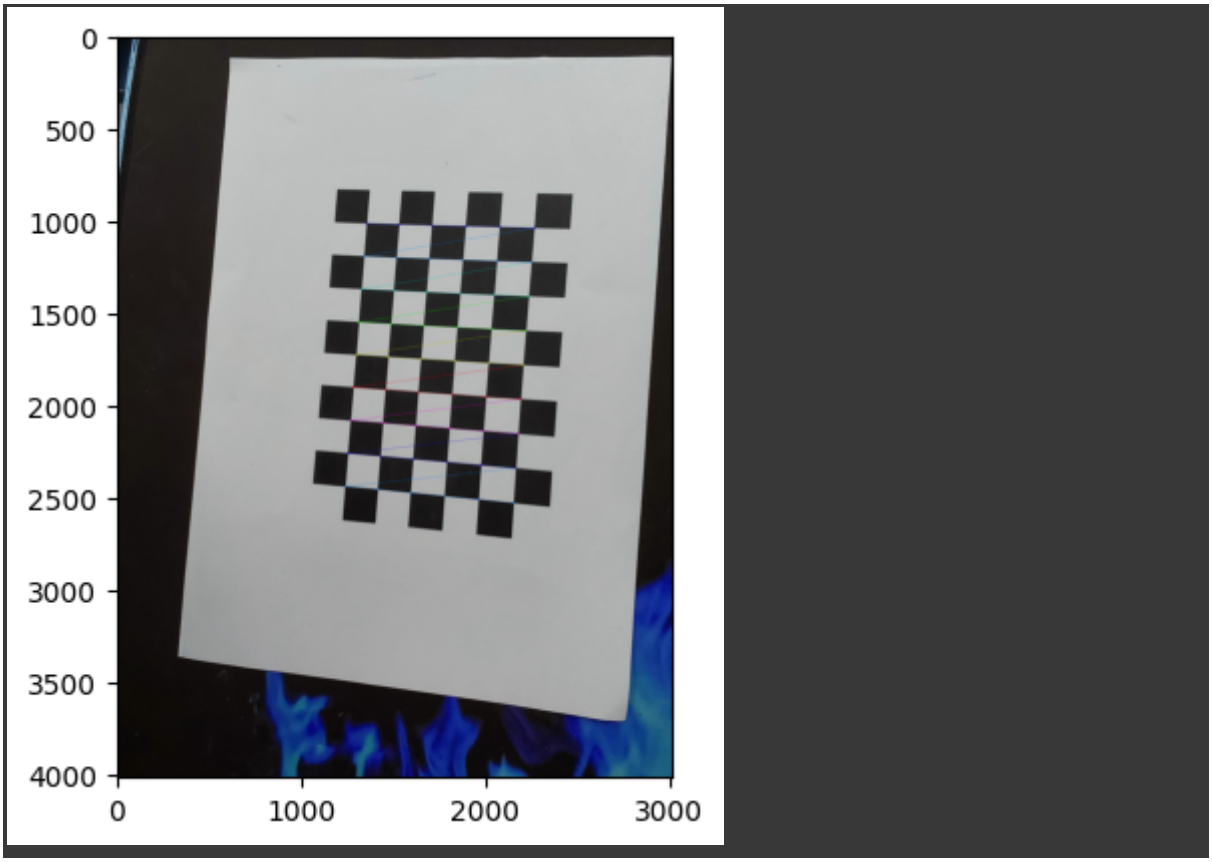
data_cv/20230411_184432.jpg
Image 23 Processed
Corners found for image 23



data_cv/20230411_184447.jpg
Image 24 Processed
Corners found for image 24



data_cv/20230411_184561.jpg
Image 25 Processed
Corners found for image 25



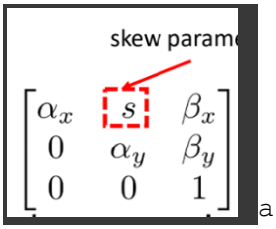
Q1) Report the estimated intrinsic camera parameters, i.e., focal length(s), skew parameter and principal point along with error estimates if available.

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

calirateCamera() returns the intrinsic parameters matrix, distortion, rotation vectors & translational vectors.

Matrix :

```
Intrinsic Parameters
[[3.47728720e+03 0.00000000e+00 1.72992761e+03]
 [0.00000000e+00 3.48716594e+03 2.22040130e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
```



alpha-x = 3477.28, alpha-y = 3487.16, s = 0, beta-x = 1729.9 and beta-y = 2220.4

Q2) Report the estimated extrinsic camera parameters, i.e., rotation matrix and translation vector for each of the selected images.

```
print("Extrinsic Parameters")
print("Rotation Vectors:")
print(rvecs)
print("Translation vectors:")
print(tvecs)
```

Printing the rotation vectors and translational vectors coming from the above code

```
Extrinsic Parameters
```

```
Rotation Vectors:
```

```
(array([[-0.05027587], [ 0.17073214],[-0.04252389]]),
array([[ 0.03342594],[-0.15821607],[-0.7237723 ]]),
array([[0.60239815],[0.57367012],[2.88497106]]),
array([[-0.12331685],[-0.08837152],[-0.47477047]]),
array([[ 0.20626741],[-0.24664723],[-1.58909282]]),
array([[-0.3361067 ],[ 0.40768885],[-1.51072676]]),
array([[ 0.00644111],[-0.13599916],[-0.50680222]]),
array([[ 0.06639143],[-0.45364487],[-0.23228168]]),
array([[-0.24432684],[-0.143595  ],[-0.0405986 ]]),
array([[0.20043661],[0.11538154],[0.10792777]]),
array([[-0.11162908],[-0.01946365],[ 0.41449773]]),
array([[ 0.22003032],[-0.27965155],[-1.5754343 ]]),
array([[-0.12489593],[ 0.3187581 ],[-0.01478638]]),
array([[-0.02684408],[-0.0471118 ],[-0.01527382]]),
array([[-0.03099365],[ 0.12347546],[ 0.01472857]]),
array([[0.10321561],[0.54861239],[2.65102254]]),
array([[-0.33764319],[ 0.16570834],[-0.34167815]]),
array([[-0.25017302],[-0.03907519],[-2.92122094]]),
array([[-0.27447257],[-0.0773407 ],[ 0.34445276]]),
array([[-0.42008677],[-0.16250269],[-1.471585  ]]),
array([[-0.05034829],[-0.13479927],[ 0.24058623]]),
array([[-0.43686376],[-0.09942321],[-0.06975093]]),
array([[0.19393708],[0.70313176],[3.01183283]]),
array([[-0.36377771],[-0.12910361],[ 0.13056714]]),
array([[-0.05686576],[ 0.13450493],[ 0.07808674]]))
```

```
Translation vectors:
```

```
(array([[-1.50181909],[-4.88568291],[19.8743715 ]]),
array([[-3.43881242],[-0.65661542],[22.43031249]]),
array([[ 0.41757823],[ 2.24038608],[15.79538736]]),
array([[-3.59237294],[-1.17717713],[22.09588925]]),
array([[-0.5612641 ],[-1.19593537],[17.6916335 ]]),
array([[-3.98301187],[-1.21195652],[21.26651764]]),
array([[-2.00684735],[-3.25524595],[22.94855498]]),
array([[-1.28892541],[-3.49868346],[17.19011064]]),
array([[-1.36568658],[-5.00996812],[17.92407707]]),
array([[-1.82112467],[-5.54776208],[19.64625677]]),
array([[ 0.66371127],[-5.71558442],[22.03392439]]),
array([[ 0.71022122],[-0.75122432],[17.36944961]]),
array([[-2.48626984],[-4.73270441],[21.32581711]]),
array([[-1.09322134],[-3.40020906],[19.3534439 ]]),
array([[-2.80806373],[-4.30562529],[17.86323039]]),
array([[ 2.95589735],[ 3.48847126],[18.41986223]]),
array([[-4.67938   ],[-4.46284144],[21.47930765]]),
array([[ 0.25027859],[ 6.40894065],[19.72310773]]),
array([[ 1.1482532 ],[-6.41176883],[21.29209995]]),
array([[-1.57133307],[-3.24782693],[18.21165552]]),
array([[-0.56398567],[-4.90745504],[18.98103713]]),
array([[-2.41629659],[-7.96884862],[20.38820974]]),
array([[ 1.68330068],[ 3.36819297],[16.72992678]]),
array([[-0.60413811],[-6.73253228],[21.09863719]]),
array([[-2.09851332],[-6.68623057],[19.76620952]]))
```

Q3) Report the estimated radial distortion coefficients. Use the radial distortion coefficients to undistort 5 of the raw images and include them in your report.

Observe how straight lines at the corner of the images change upon application of the distortion coefficients. Comment briefly on this observation.

```
print("Distortion Coefficients")
print(dist)
```

Printing the distortion coefficient from the above code.

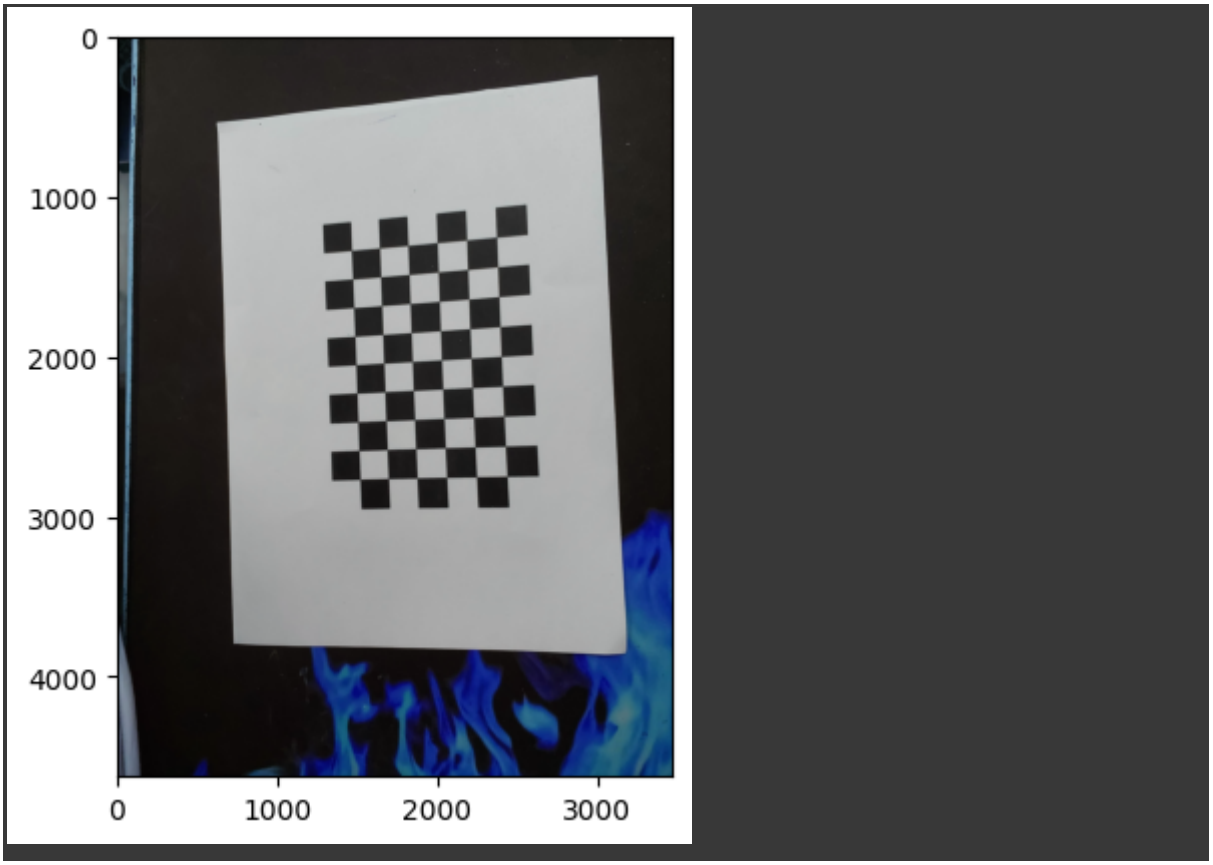
```
Distortion Coefficients
[[ 0.31187401 -2.8687774  -0.01079368 -0.00551166  5.38258322]]
```

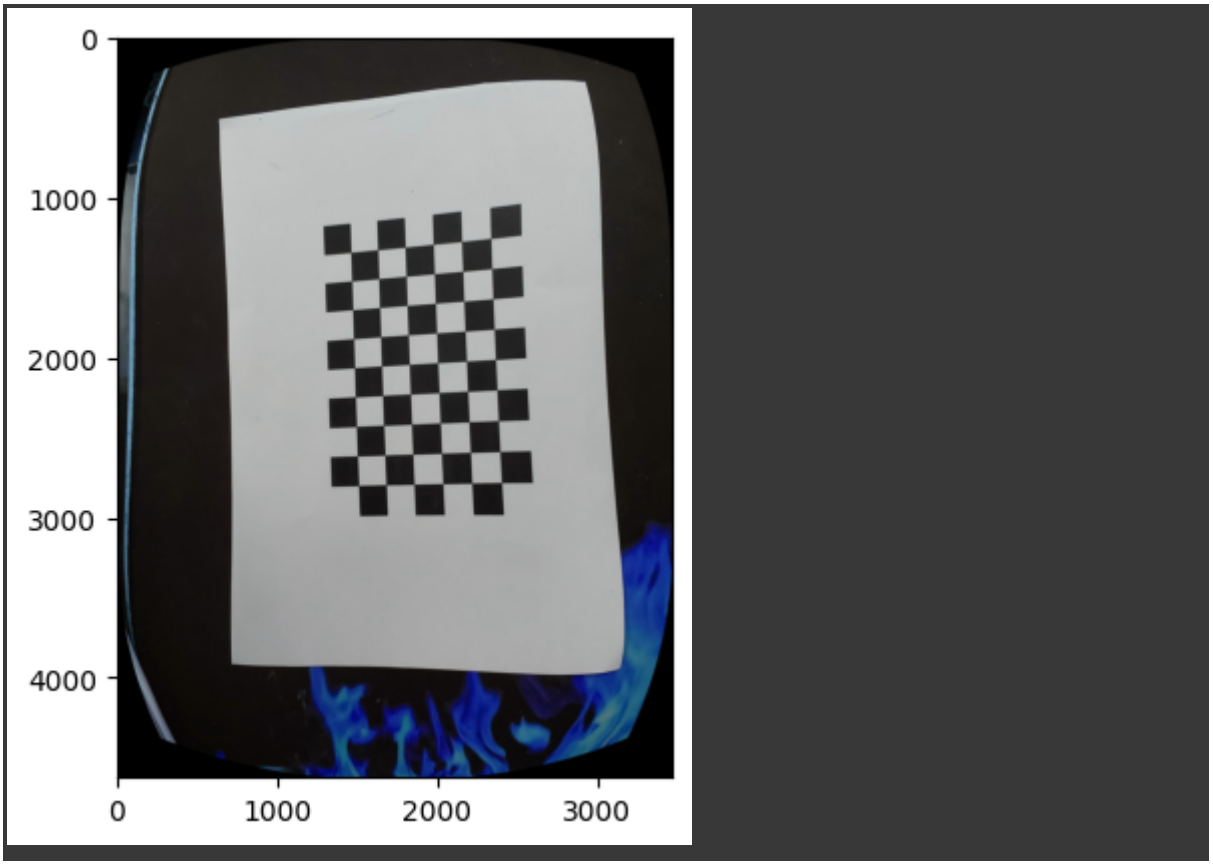
```
for i in range(5):
    print(images[i])
    img = cv.imread(images[i])
    h,  w = img.shape[:2]
    newcameramtx, roi = cv.getOptimalNewCameraMatrix(mtx, dist, (w,h),
1, (w,h))
```

```
    dst = cv.undistort(img, mtx, dist, None, newcameramtx)
    plt.imshow(img)
    plt.show()
    plt.imshow(dst)
    plt.show()
    print("\n--------------------------------\n")
```

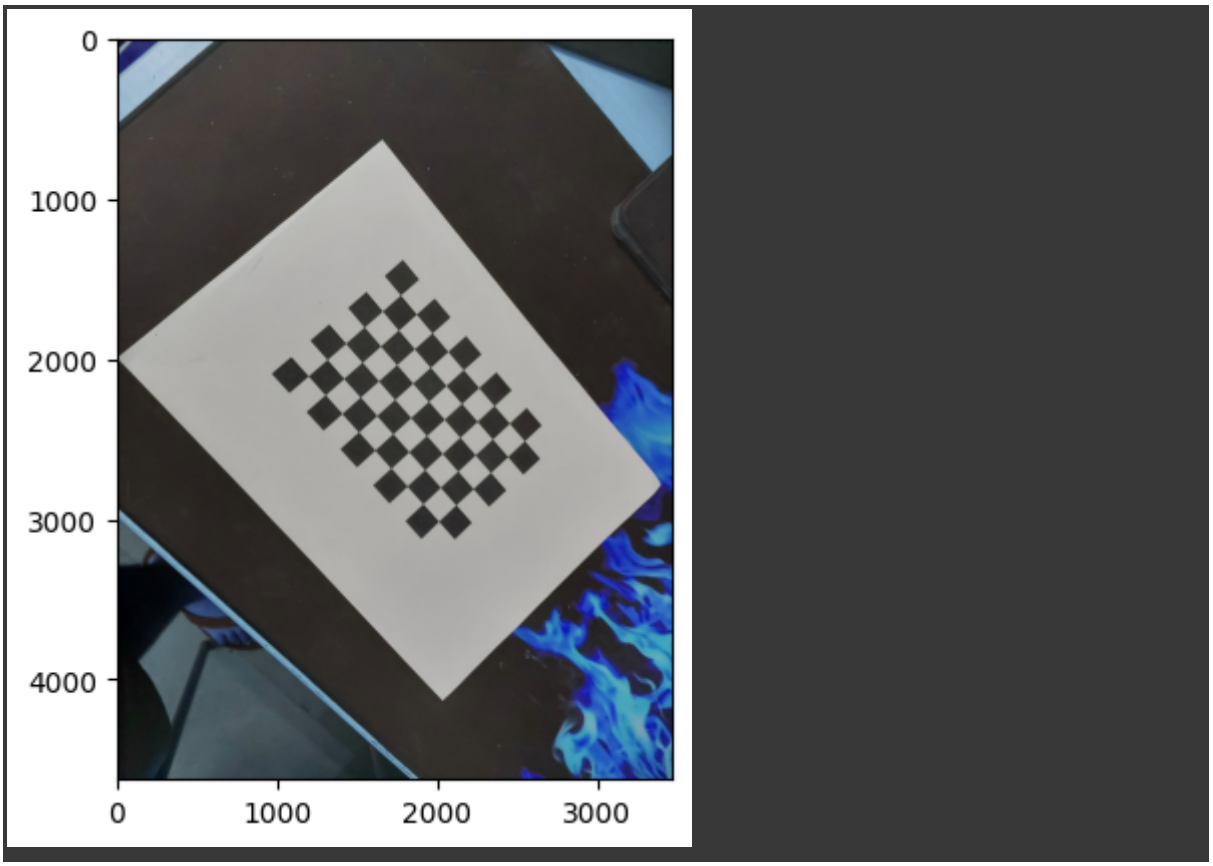
For the first 5 images, first read the images using imread(), next we get its height and width using shape. Next, we compute the new camera matrix and region of interest using the getOptimalNewCameraMatrix() function, which uses the previous camera matrix, distortion coefficients, and the calculated width and height.
Then we remove the distortion in the targeted image using the undistort() function, which uses the image, the previous matrix, distortion coefficients, and the new camera matrix.
Finally, we plot both the original and the undistorted images.

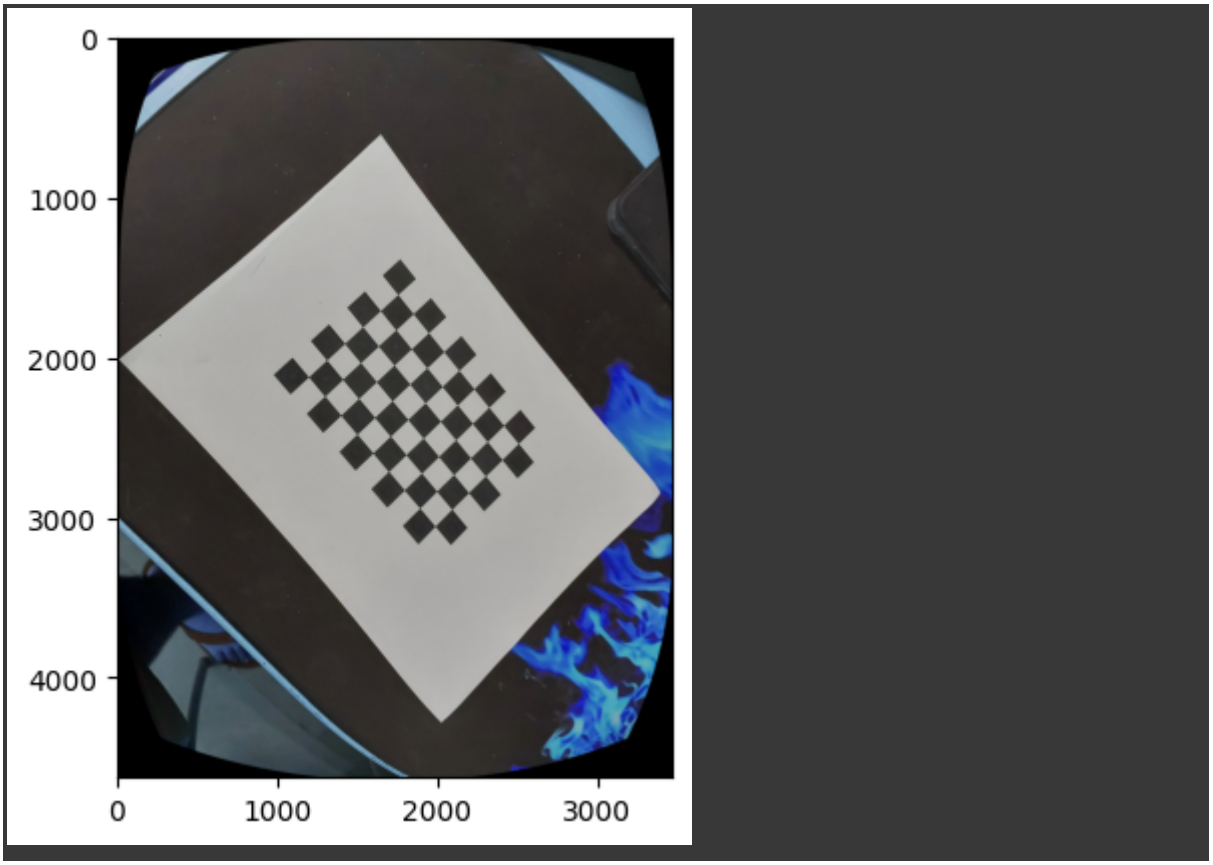
data_cv/20230411_184548.jpg



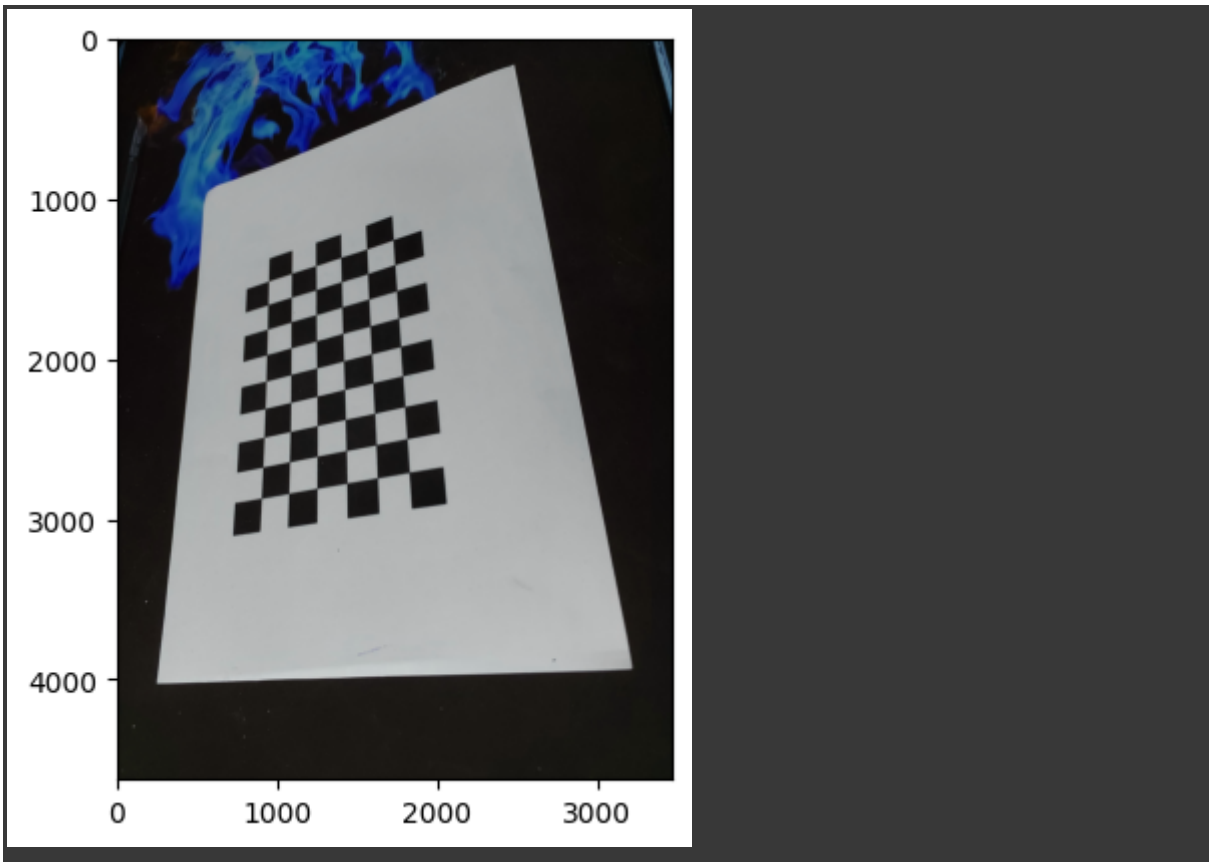


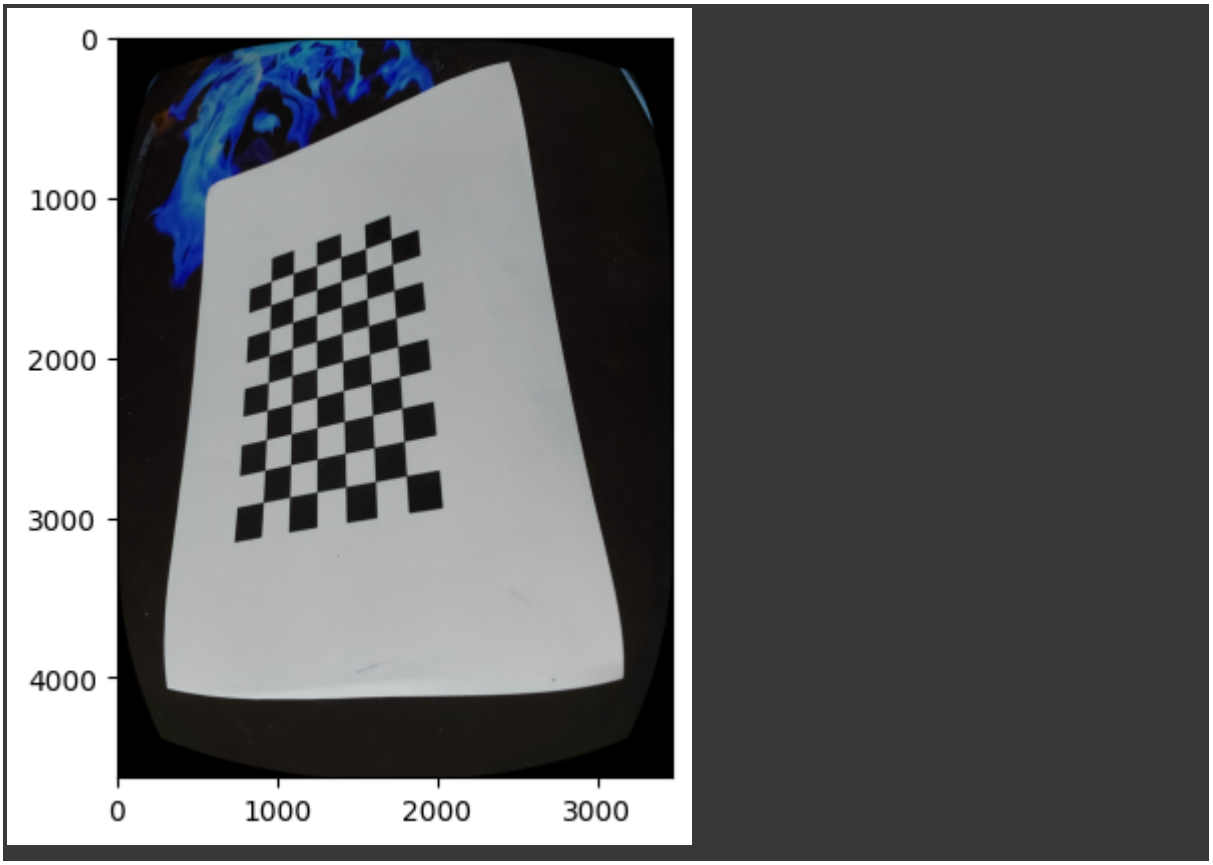
data_cv/20230411_184855.jpg



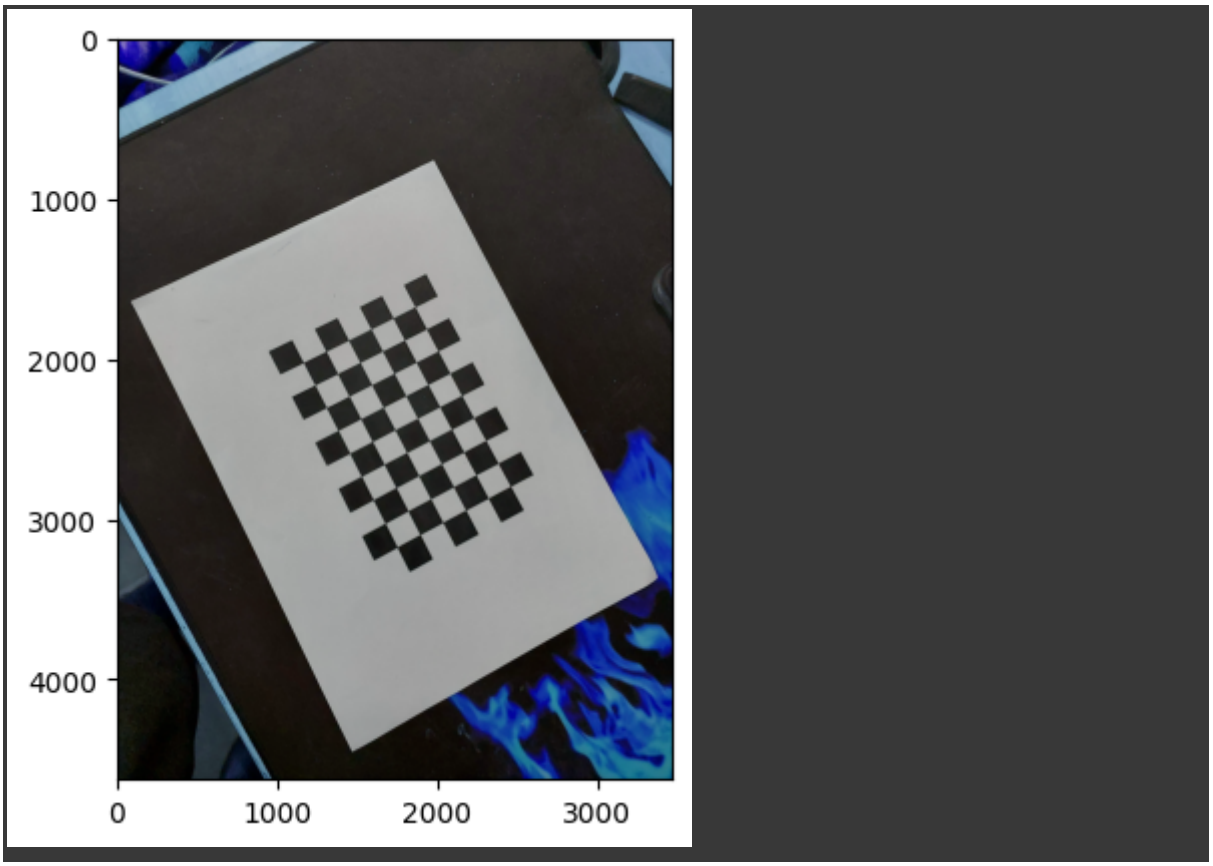


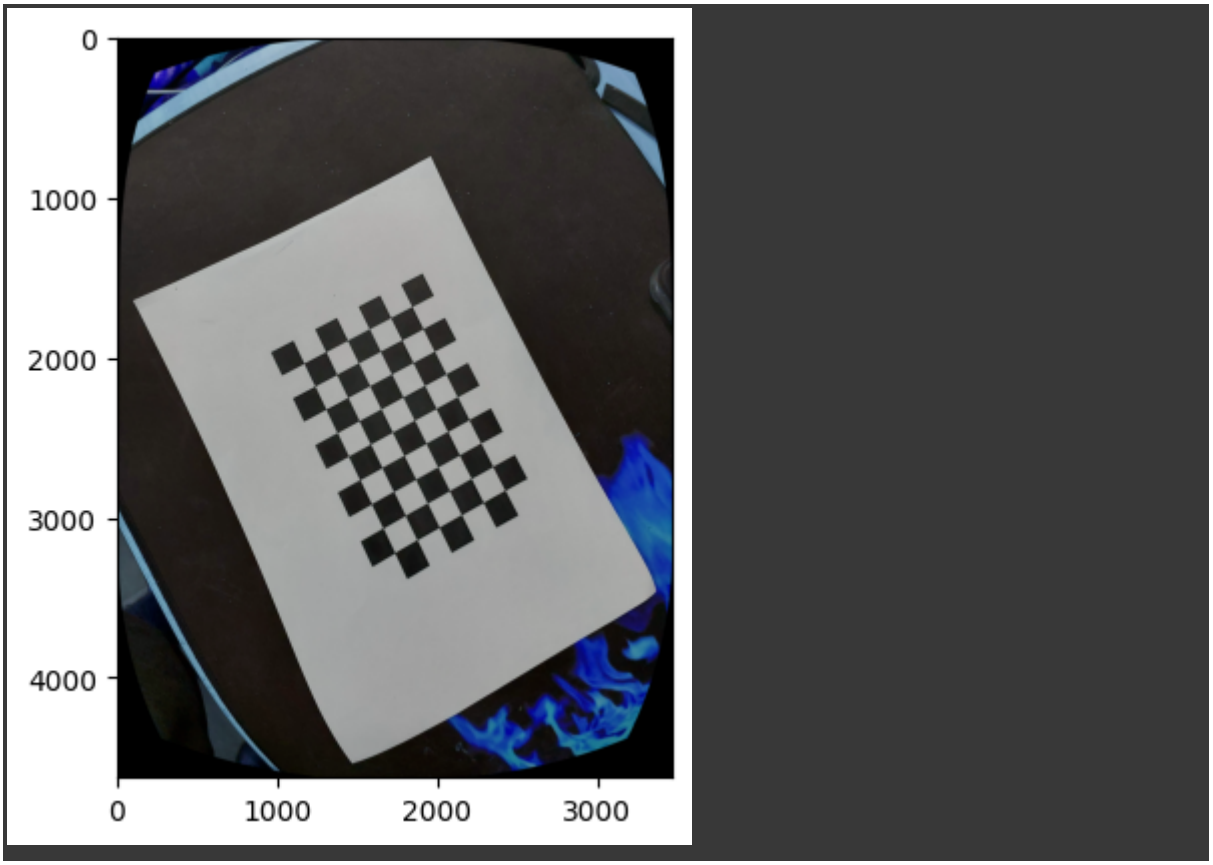
data_cv/20230411_184601.jpg



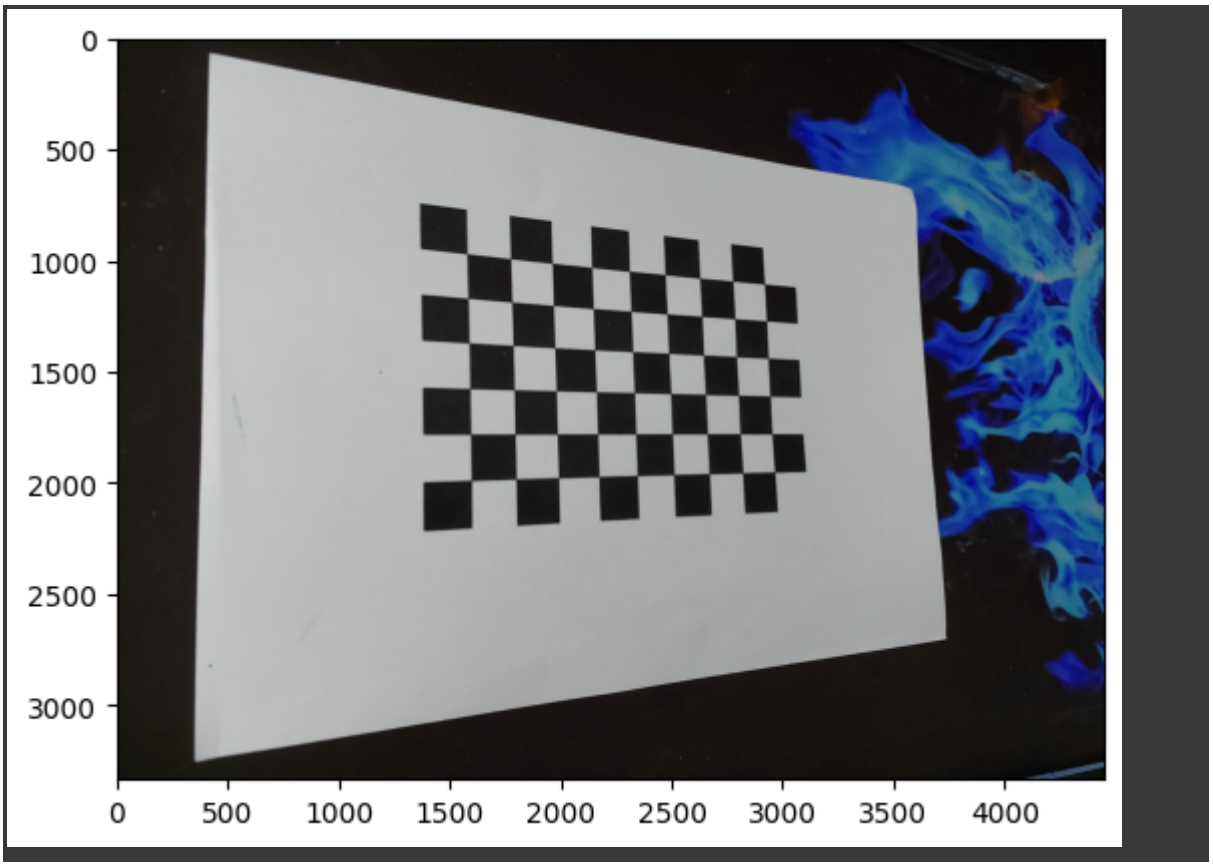


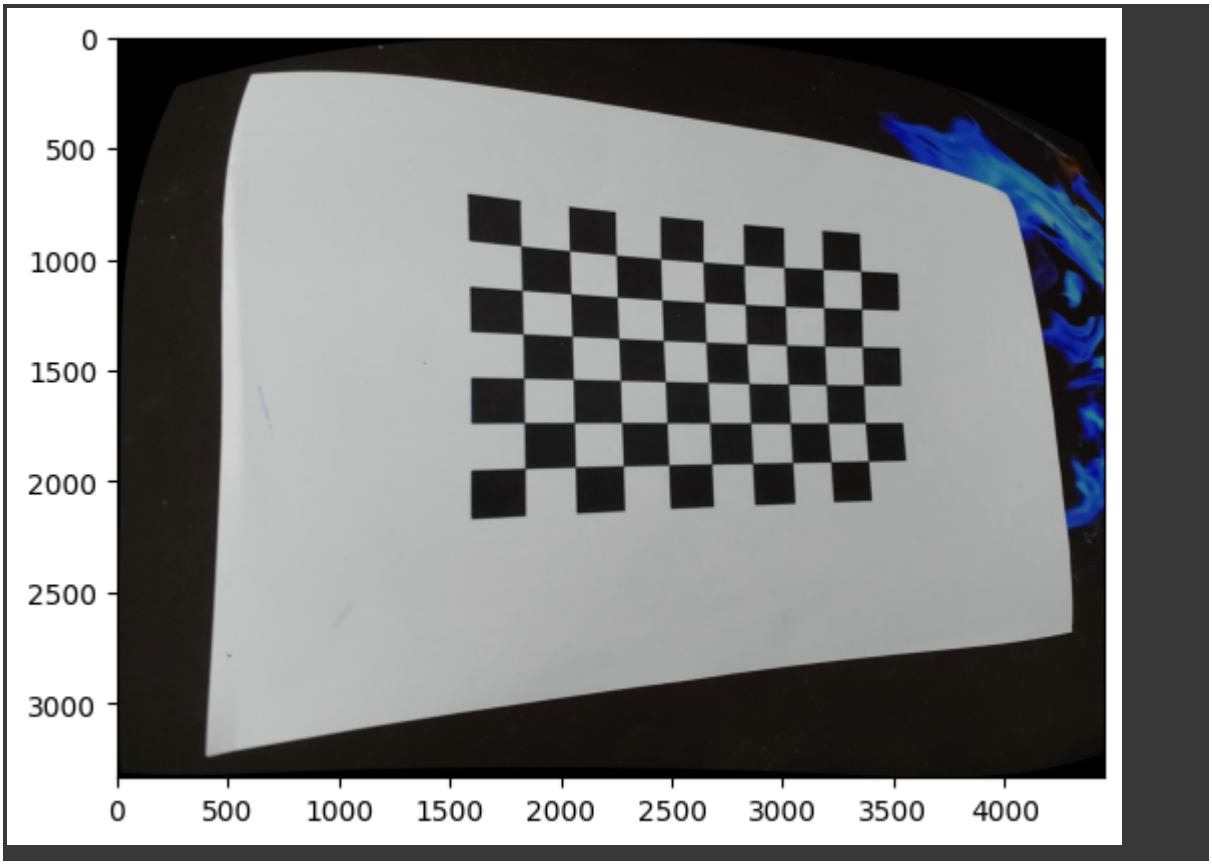
data_cv/20230411_184821.jpg





data_cv/20230411_184551.jpg





The straight lines in the corners of the checkerboard may appear to bend or curve when the distortion factors are applied. This occurs as a result of how the checkerboard is translated from the 3D environment to the 2D image plane depending on the distortion coefficients. Curved lines may appear because the distortion may cause the image to be stretched or compressed in some places so that the checkerboard can get rid of the minute curves and become a straight and parallel figure.

The straight lines that seemed bent in the distorted checkerboard can be rendered straight again by applying the ideal new camera matrix to the distorted checkerboard. This is due to the fact that the new camera matrix aids in correcting the lens distortion, producing a more realistic portrayal of the actual scene.

Finally, we can observe the lines at the checkerboard have become straighter and parallel to each other. But this stretching and compression of the checkerboard have caused the non important component of the image to bend and distort even further.

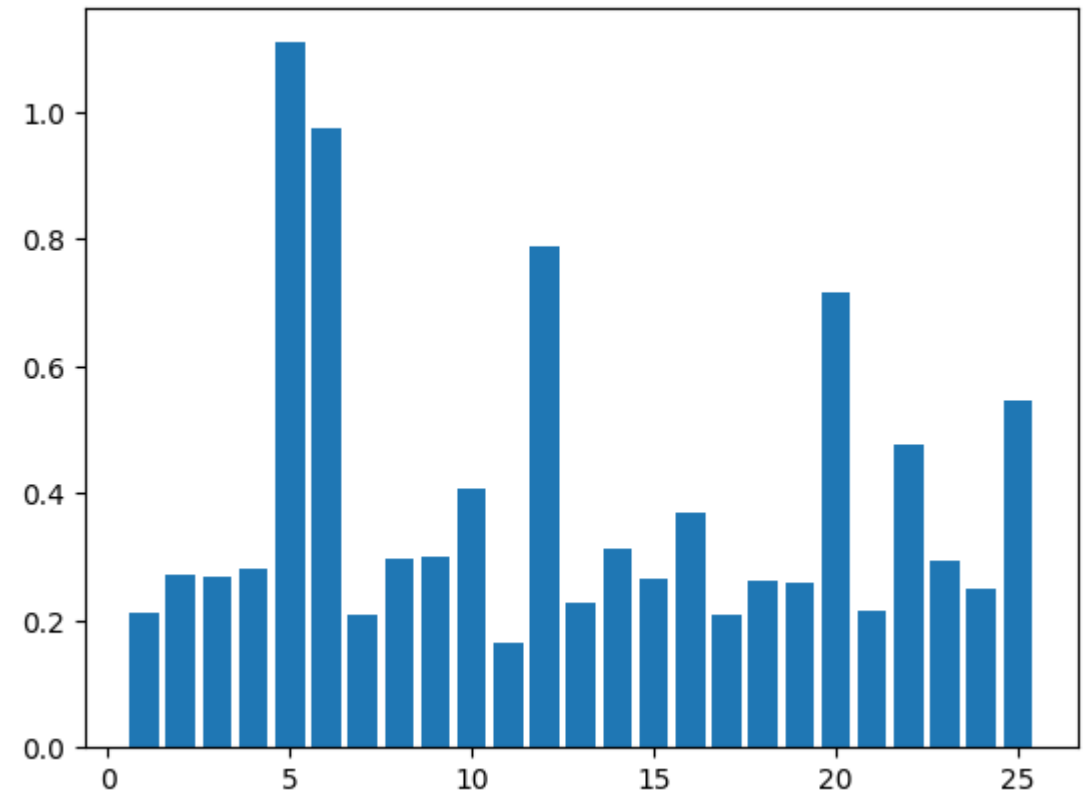
Q4) Compute and report the re-projection error using the intrinsic and extrinsic camera parameters for each of the 25 selected images. Plot the error using a bar chart. Also report the mean and standard deviation of the re-projection error.

```
error = []
new_2d = []
for i in range(len(objpoints)):
```

```
    imgpoints2, _ = cv.projectPoints(objpoints[i], rvecs[i], tvecs[i],
mtx, dist)
    error.append(cv.norm(imgpoints[i], imgpoints2,
cv.NORM_L2)/len(imgpoints2))
    new_2d.append(imgpoints2)
plt.bar(range(1, len(images)+1), error)
plt.show()
```

After initializing the error array and the new_2d points array. We loop for all images. Inside the loop, we use the projectPoints() function, which projects 3d points to 2d space using previously calculated object points, corresponding rotation and translation vectors, and the distortion coefficients. Next, we calculate the corresponding reprojection error using the Euclidean L2 Norm and append it to the error array while appending the reprojected 2d points to the new_2d array.

Finally, we plot the bar graph of the error array using the plt.bar() function.




```
print("Mean of Reprojection Error:", np.mean(error))
print("Standard Deviation of Reprojection Error:", np.std(error))
```

Next we find the mean and the standard deviation of the error array using numpy library.

```
Mean of Reprojection Error: 0.3877144148079726
Standard Deviation of Reprojection Error: 0.24504803502762892
```

Q5) Plot figures showing corners detected in the image along with the corners after the re-projection onto the image for all the 25 images. Comment on how is the reprojection error computed.

```
n=1
for fname in images:
    print(fname)
    img = cv.imread(fname)
    img1 = cv.imread(fname)
    cv.drawChessboardCorners(img, (6,9), imgpoints[n-1], ret)
    plt.imshow(img)
    plt.show()
    cv.drawChessboardCorners(img1, (6,9), new_2d[n-1], ret)
    print("\nImage", n, "Processed\n")
    plt.imshow(img1)
    plt.show()
    n+=1
    print("\n-------------------------------\n")
```

We will loop on all images. We would simply use the precalculated old image points from imgpoints array and then implement the drawChessboardCorners() on them. Next, use the precalculated new image points from the new_2d array, then implement the drawChessboardCorners() on them. Finally, we plot both images with their respective checkerboards.

A measurement of the calibration's precision is the reprojection error. It measures the difference between the detected image points and the associated reprojected points generated from the estimated camera settings.

The reprojection error between the projected points and the detected images is calculated by taking the Euclidean L2 Norm and diving it by the length of the image points array.

data_cv/20230411_184548.jpg

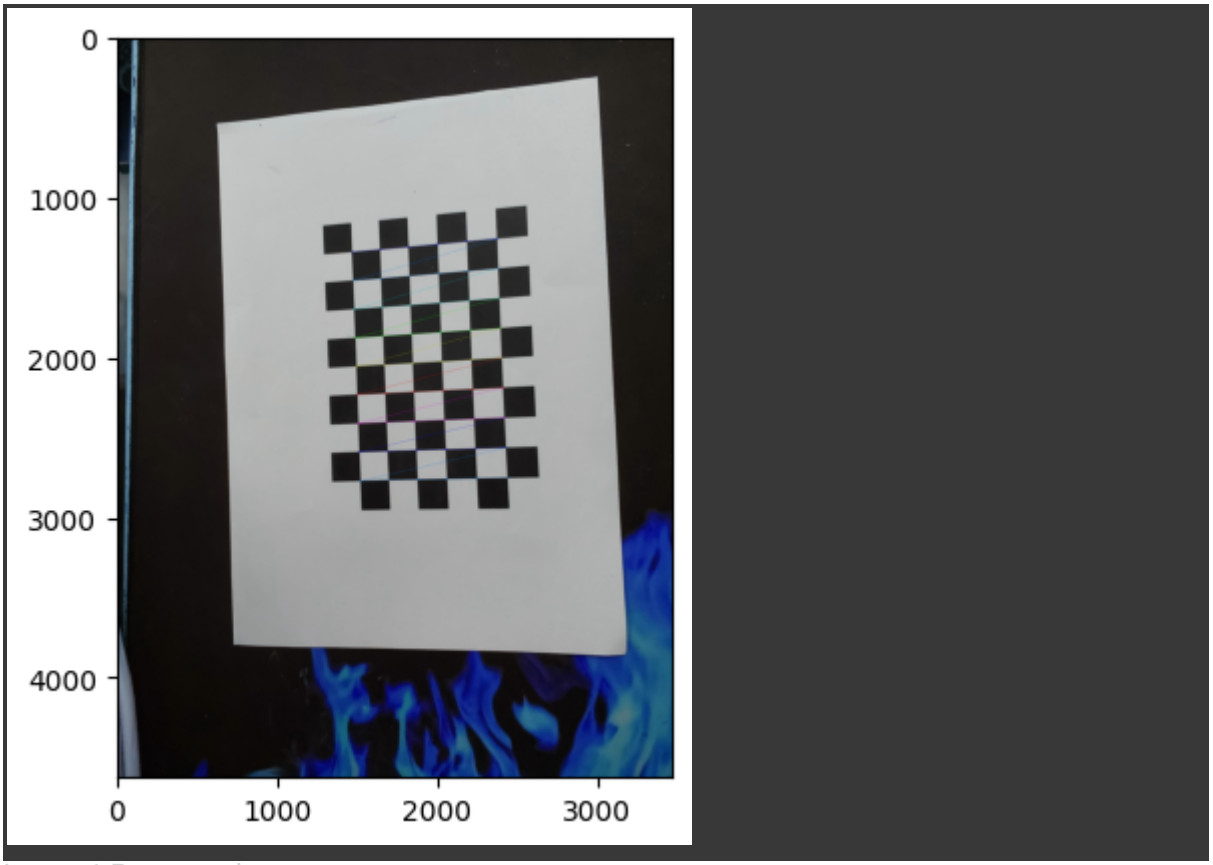
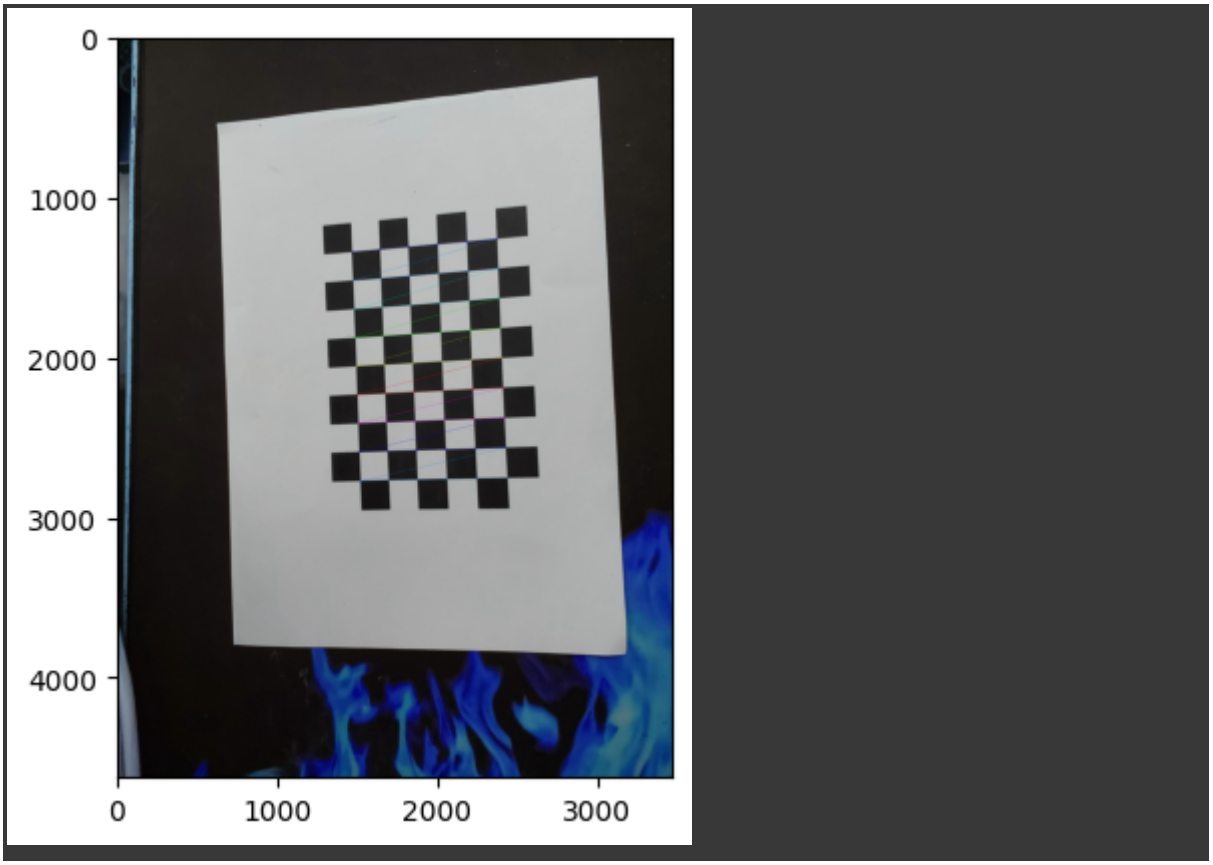


Image 1 Processed



data_cv/20230411_184855.jpg

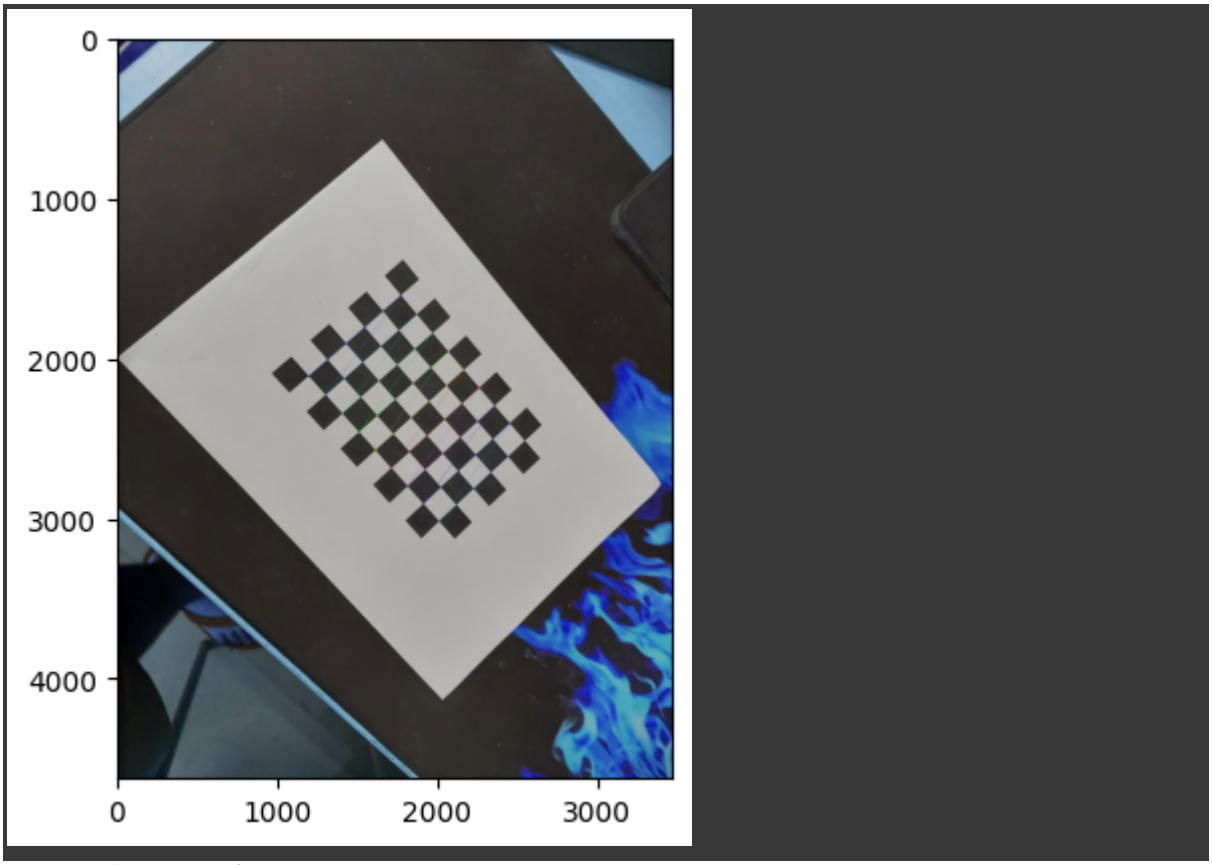
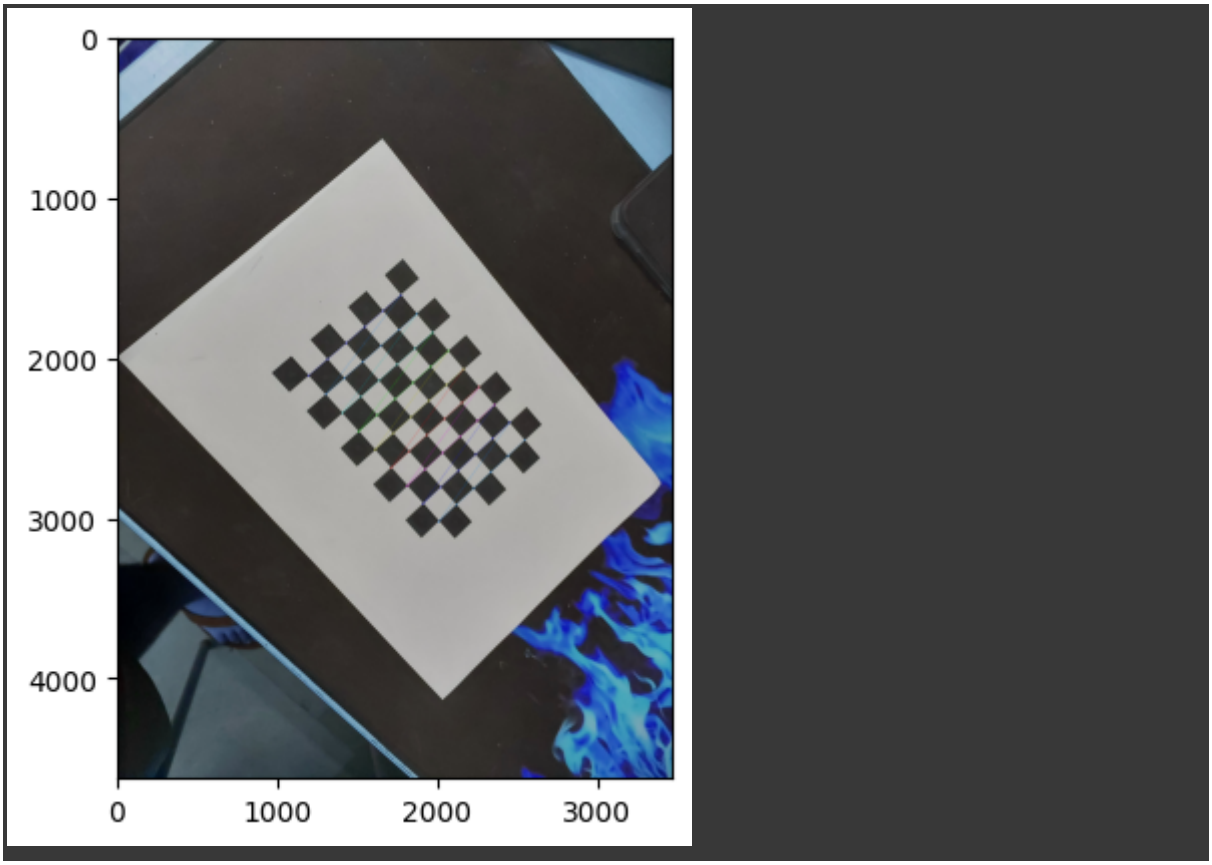


Image 2 Processed



data_cv/20230411_184601.jpg

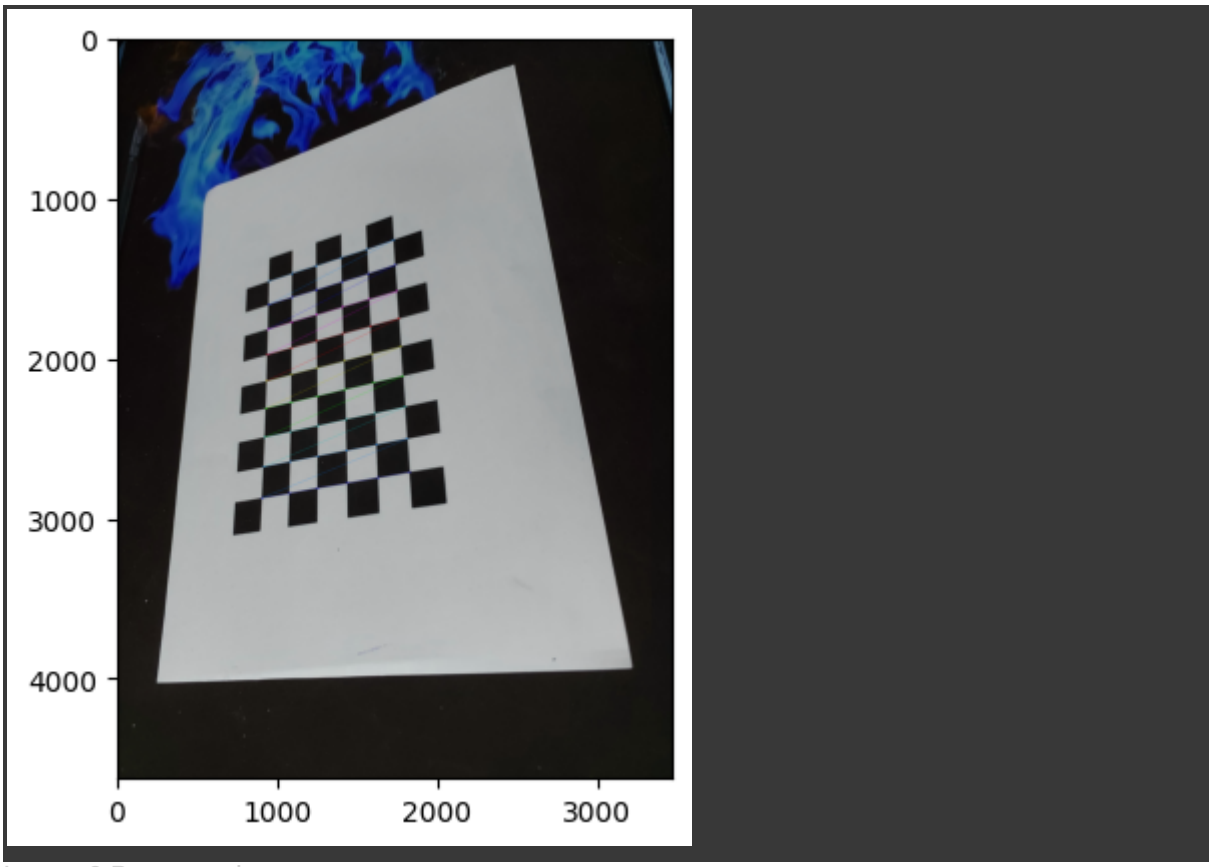
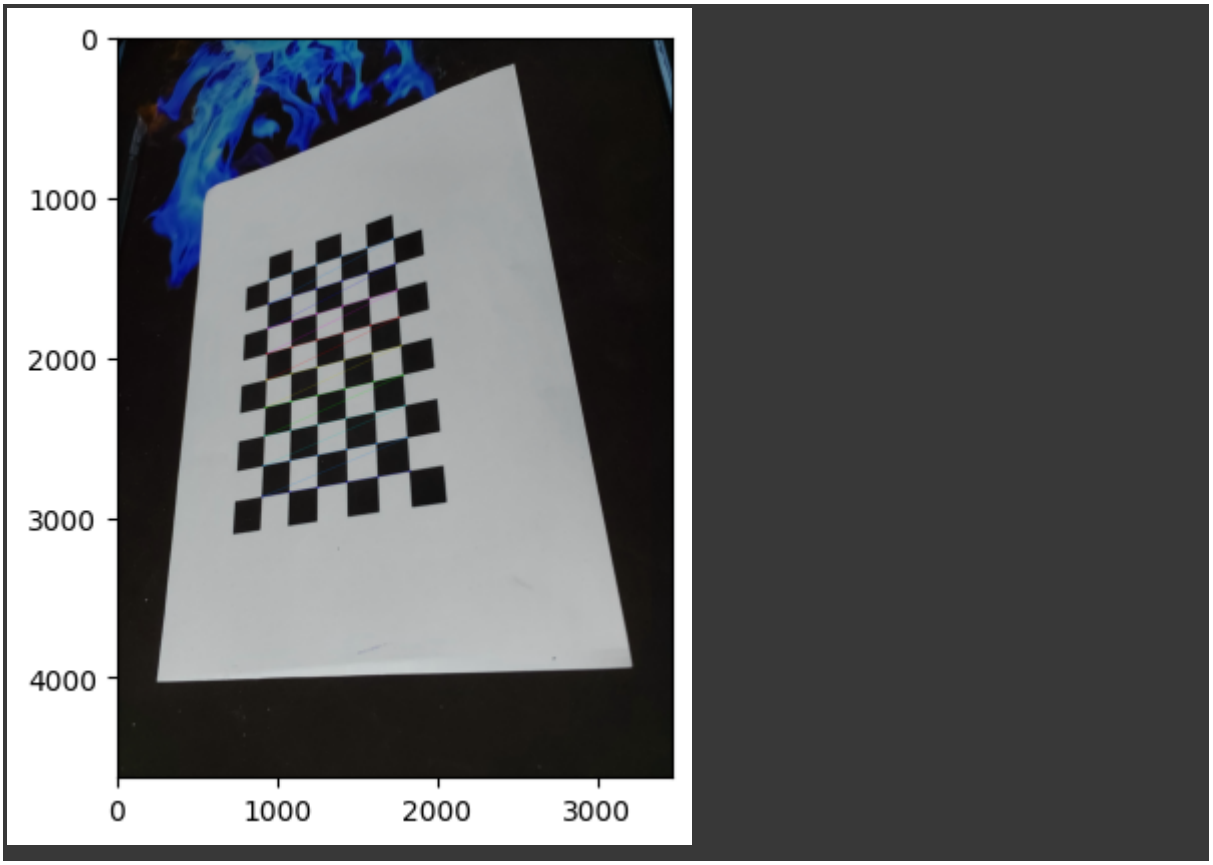


Image 3 Processed



data_cv/20230411_184821.jpg

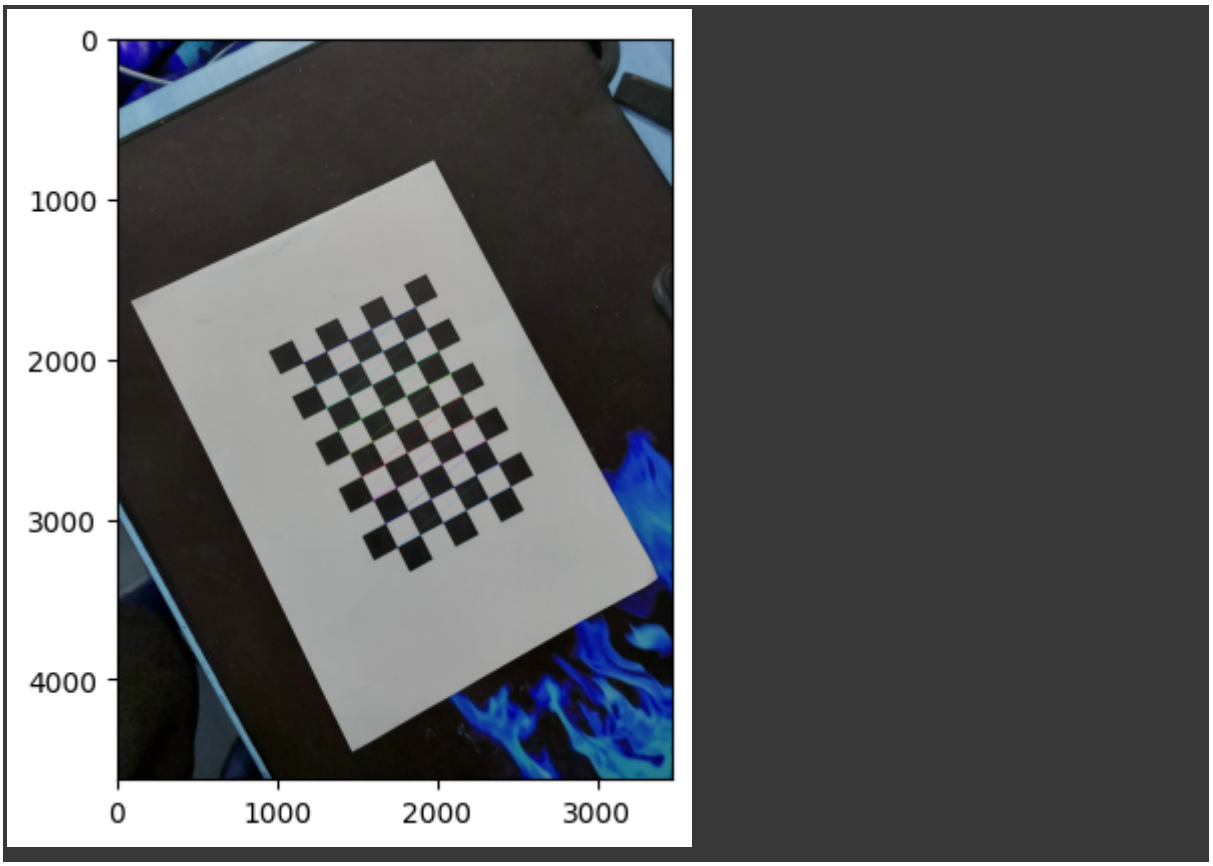
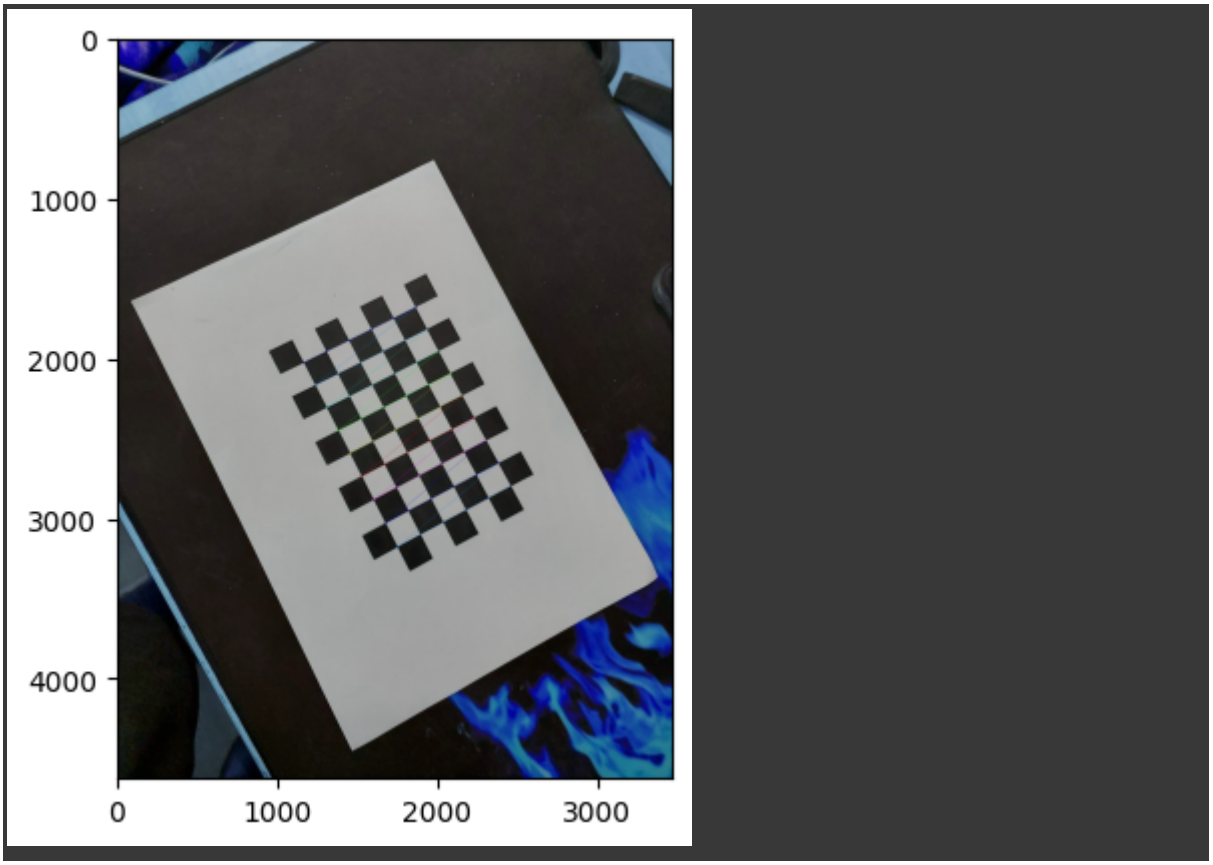


Image 4 Processed



---

data_cv/20230411_184551.jpg

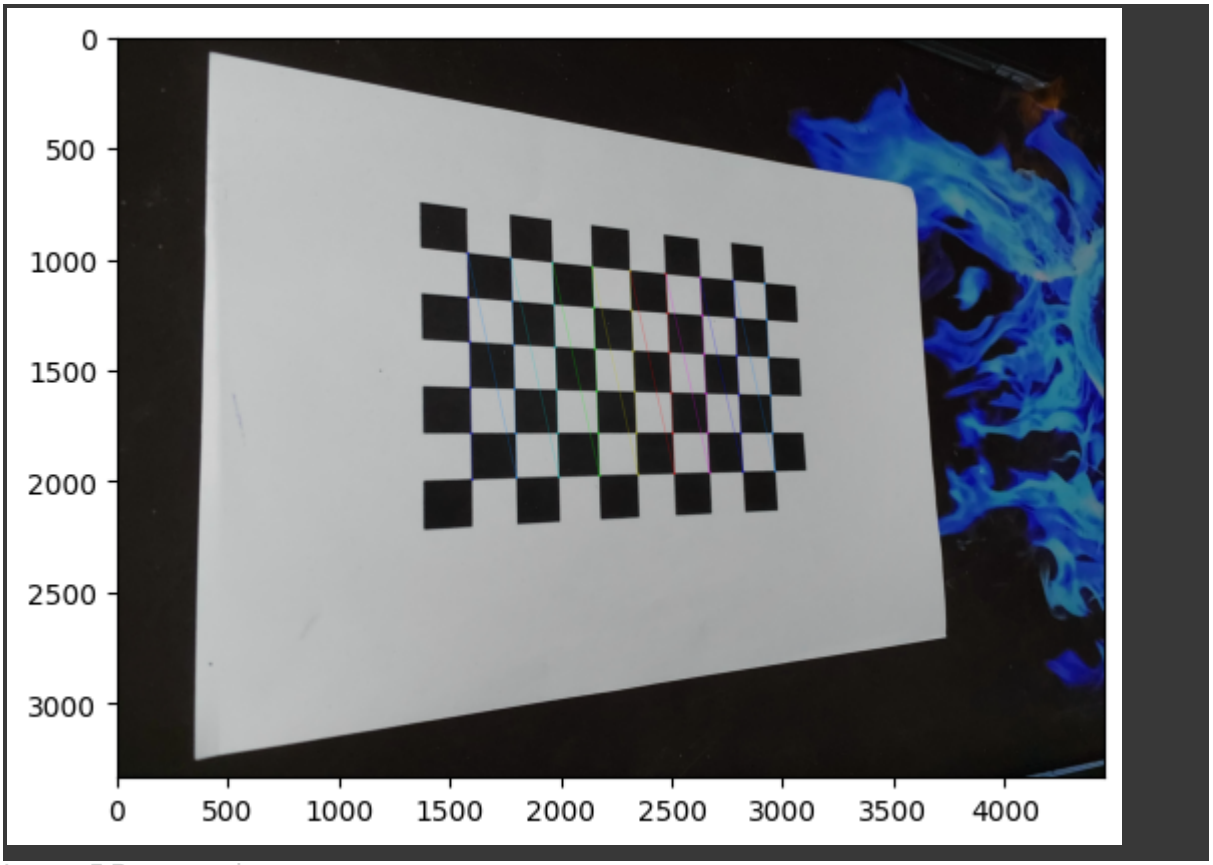
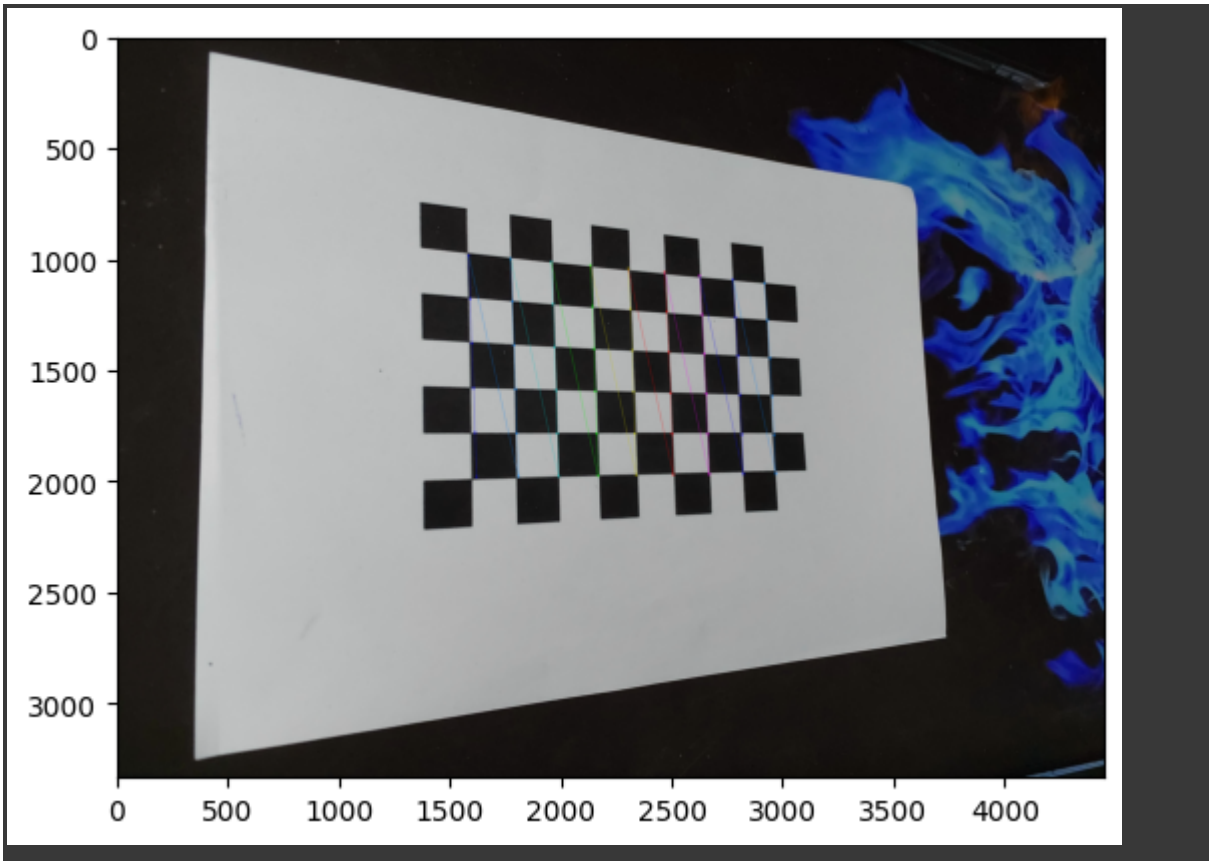


Image 5 Processed



data_cv/20230411_184937.jpg

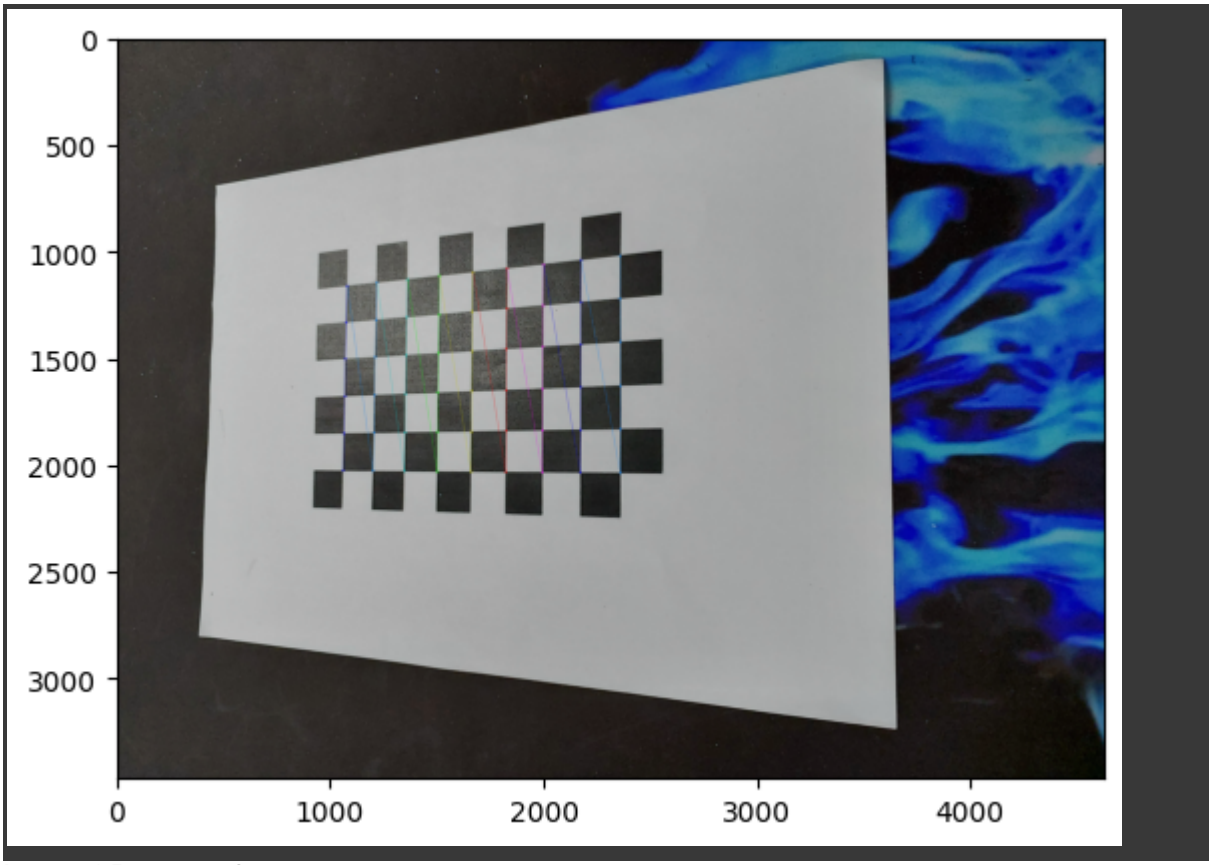
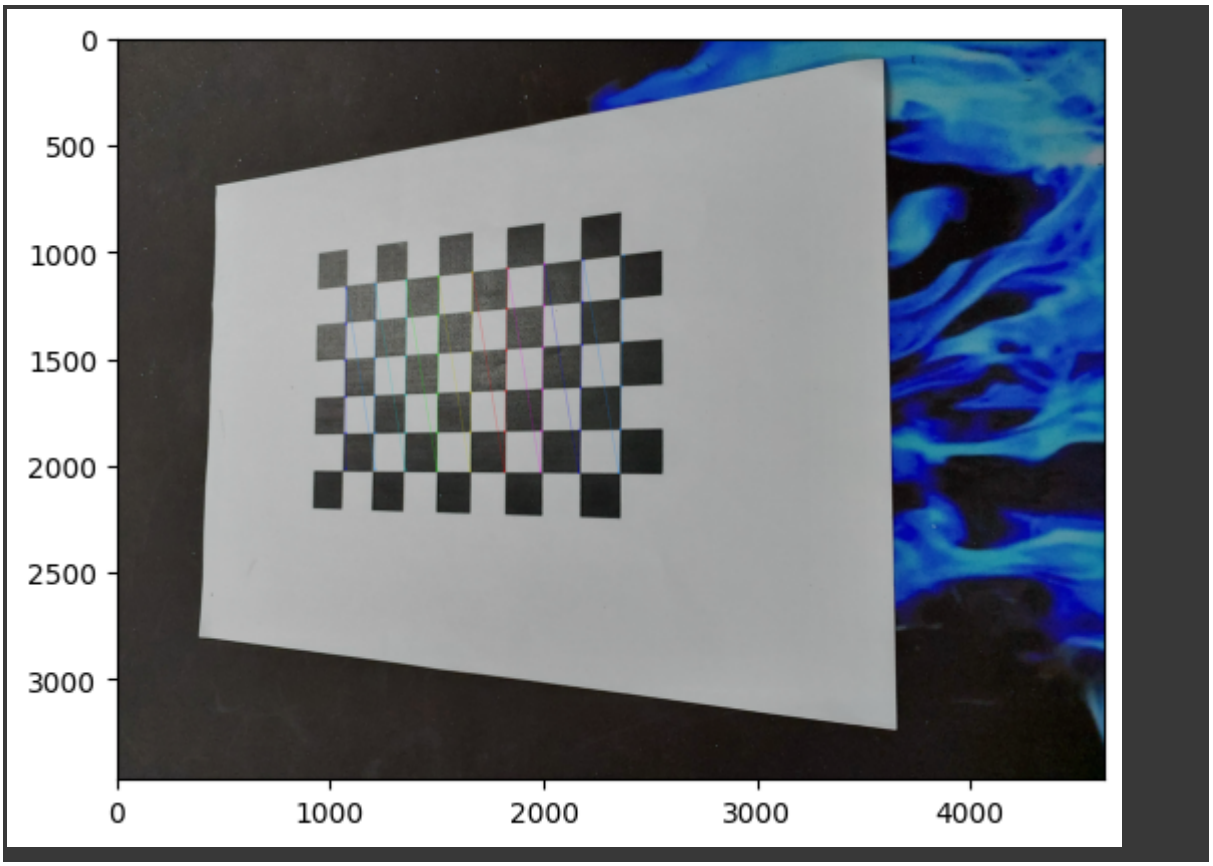


Image 6 Processed



---

data_cv/20230411_184633.jpg

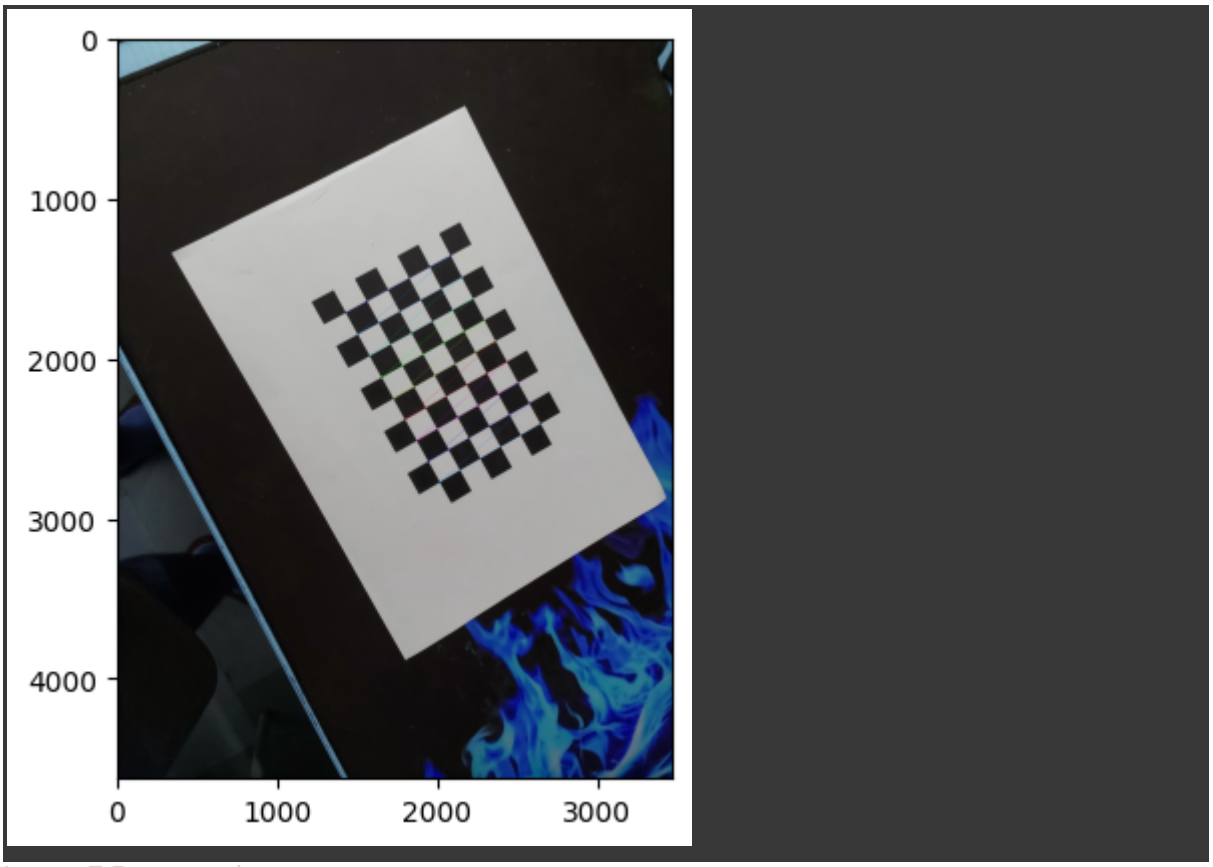
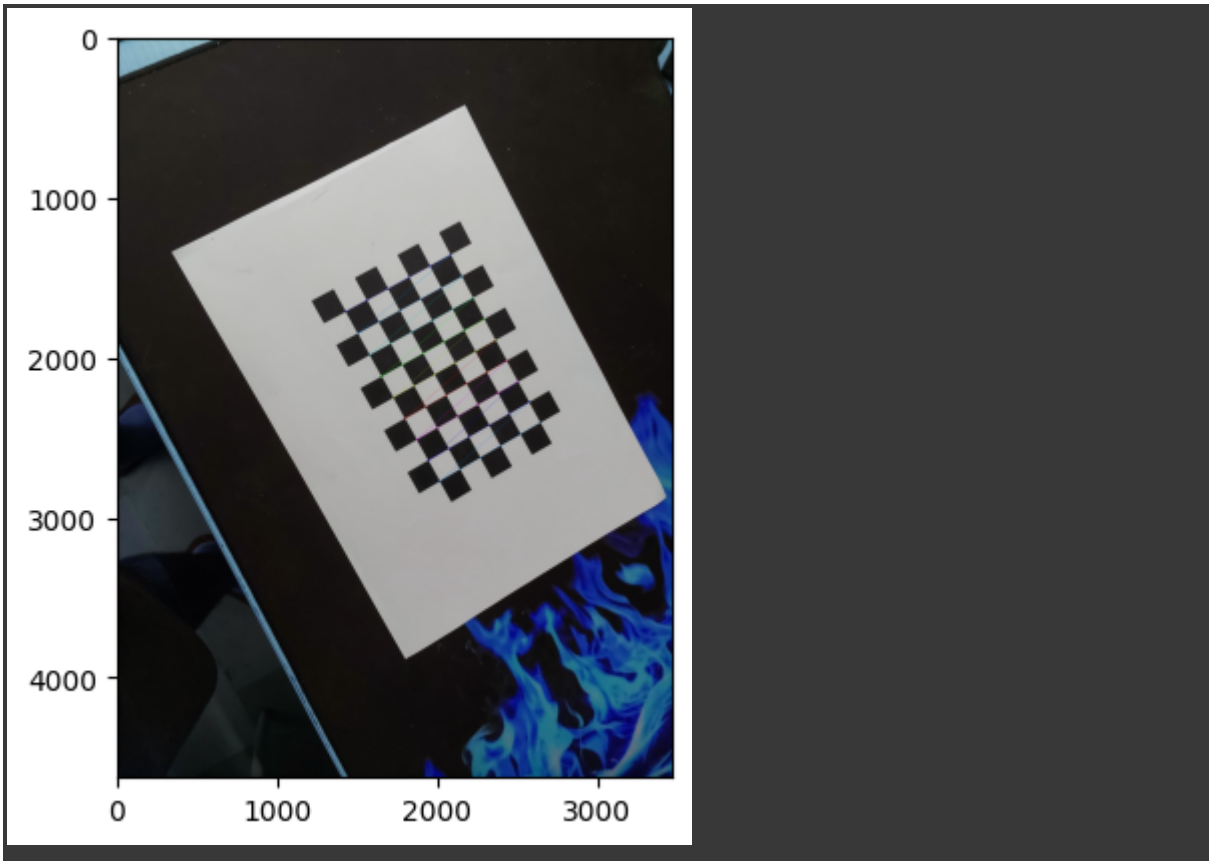


Image 7 Processed



data_cv/20230411_184325.jpg

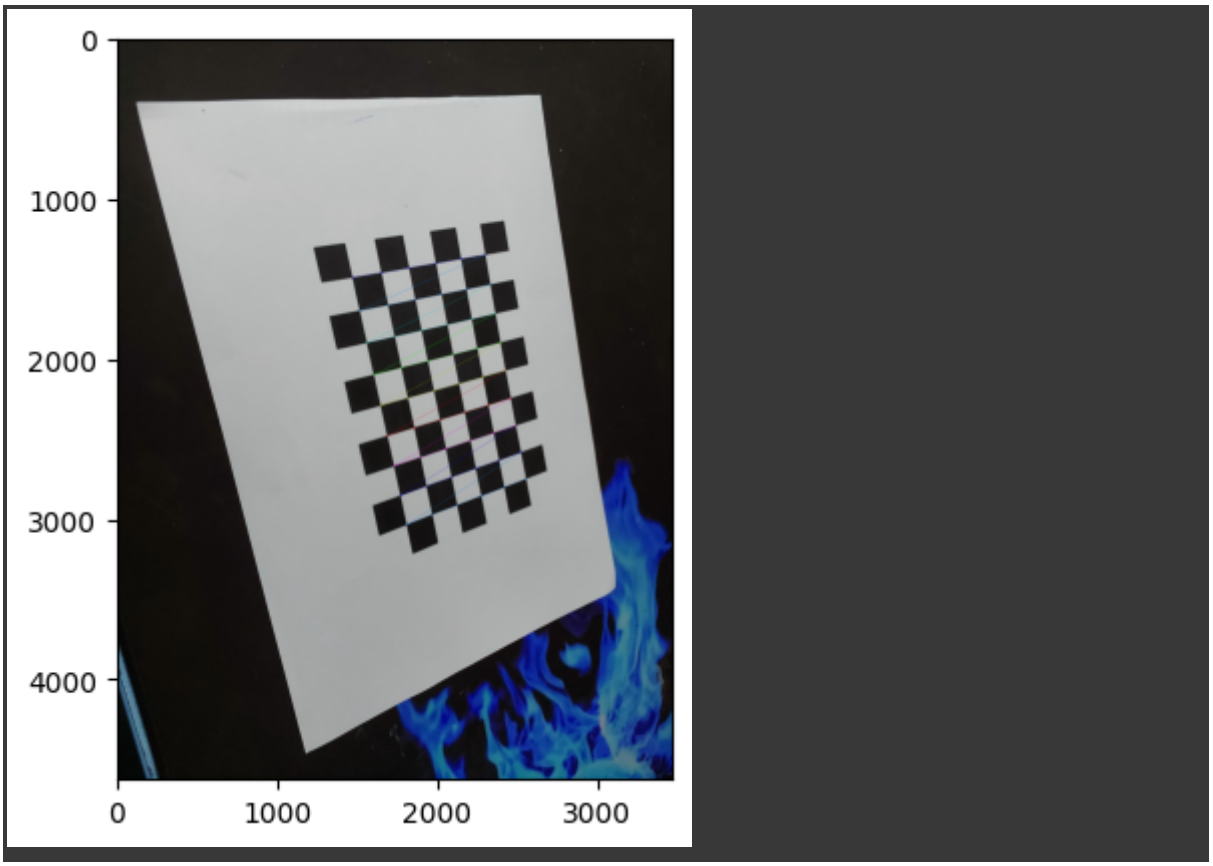
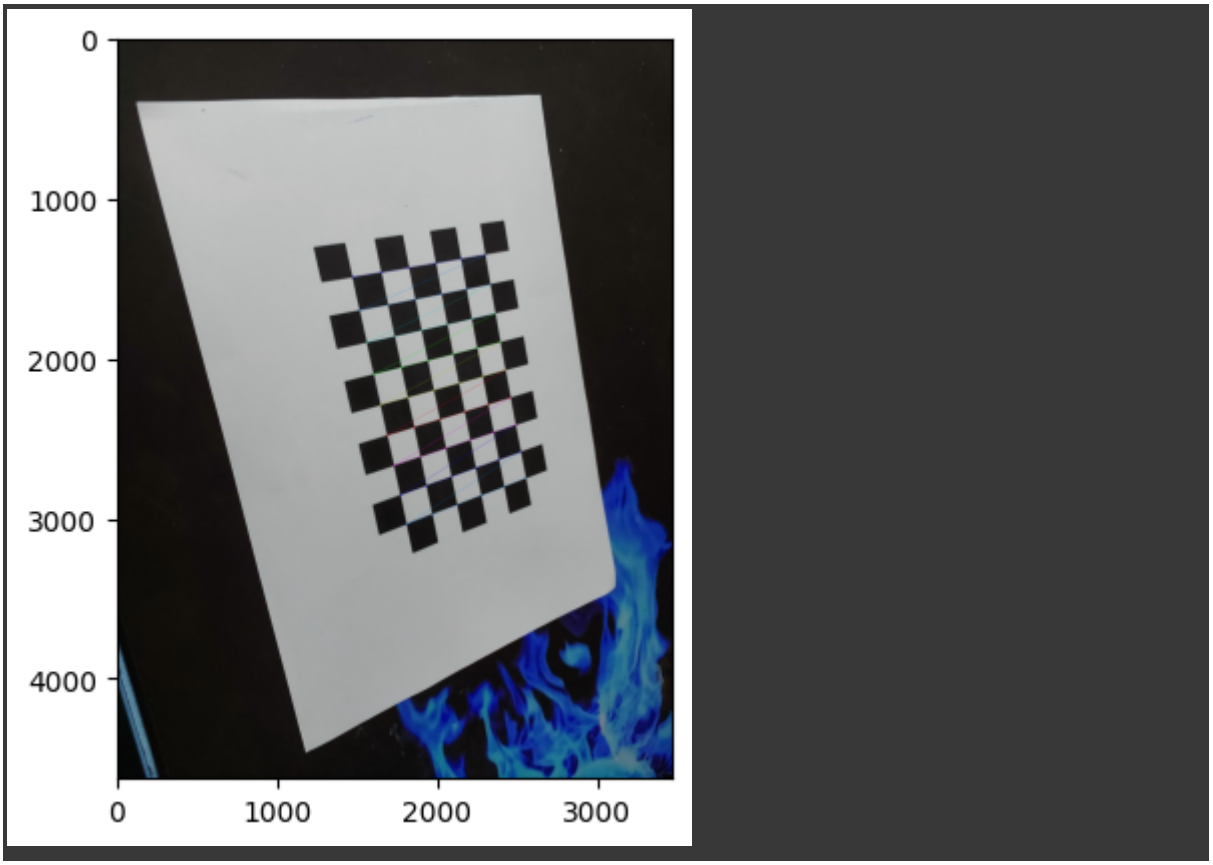


Image 8 Processed



---

data_cv/20230411_184318.jpg

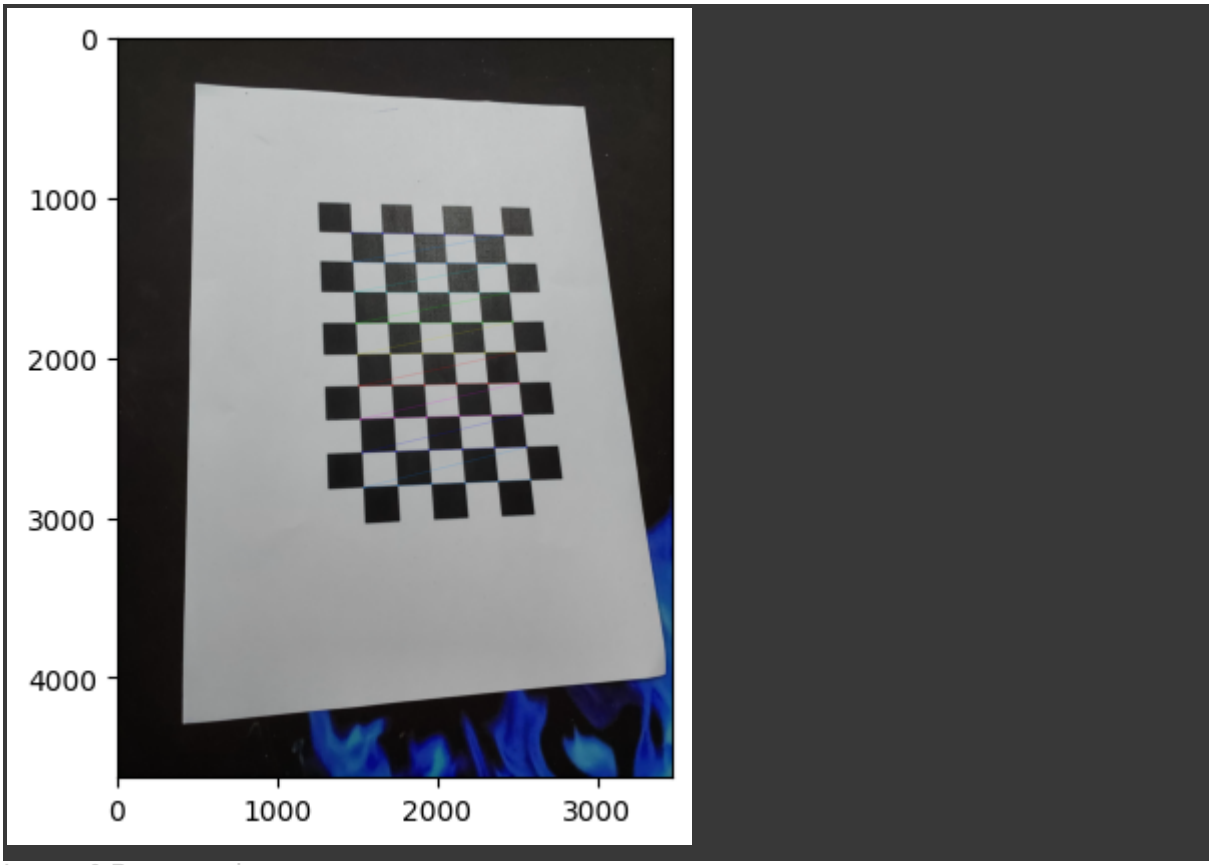
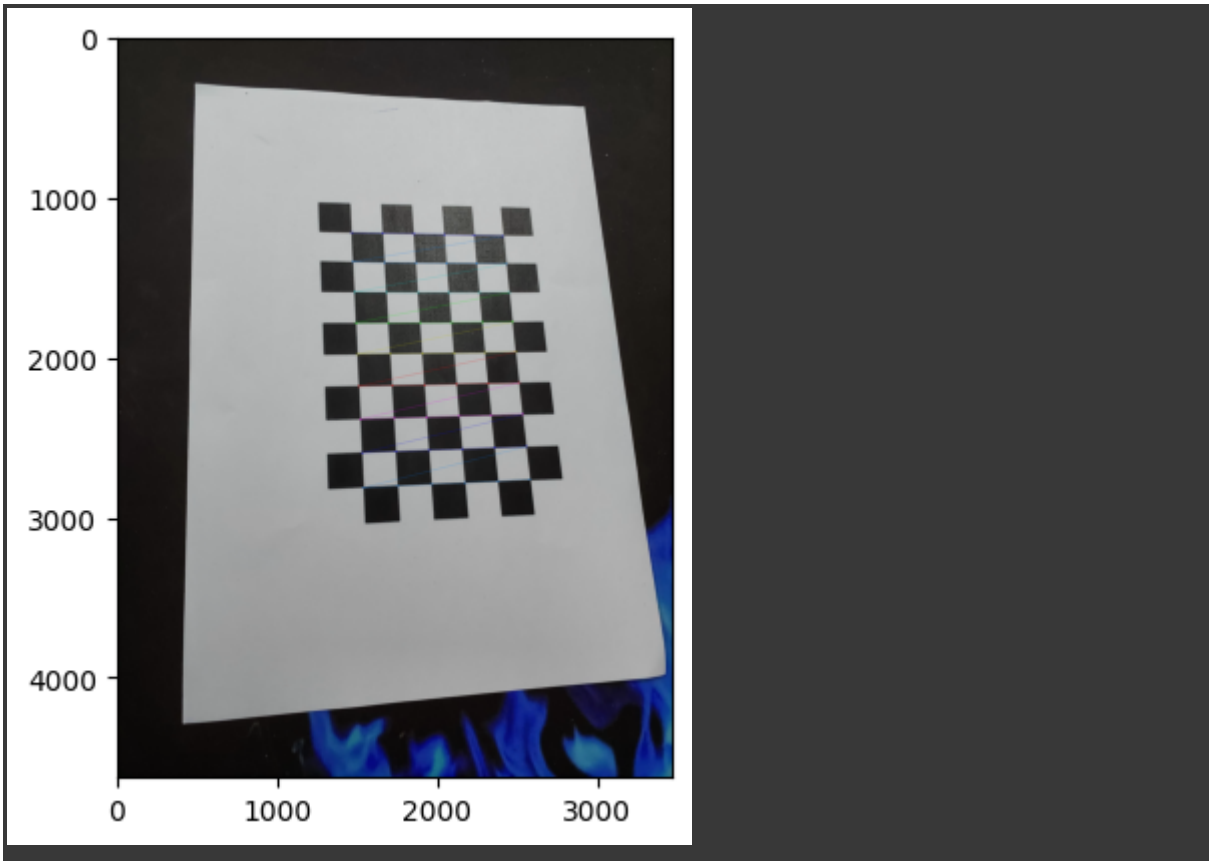


Image 9 Processed



data_cv/20230411_184502.jpg

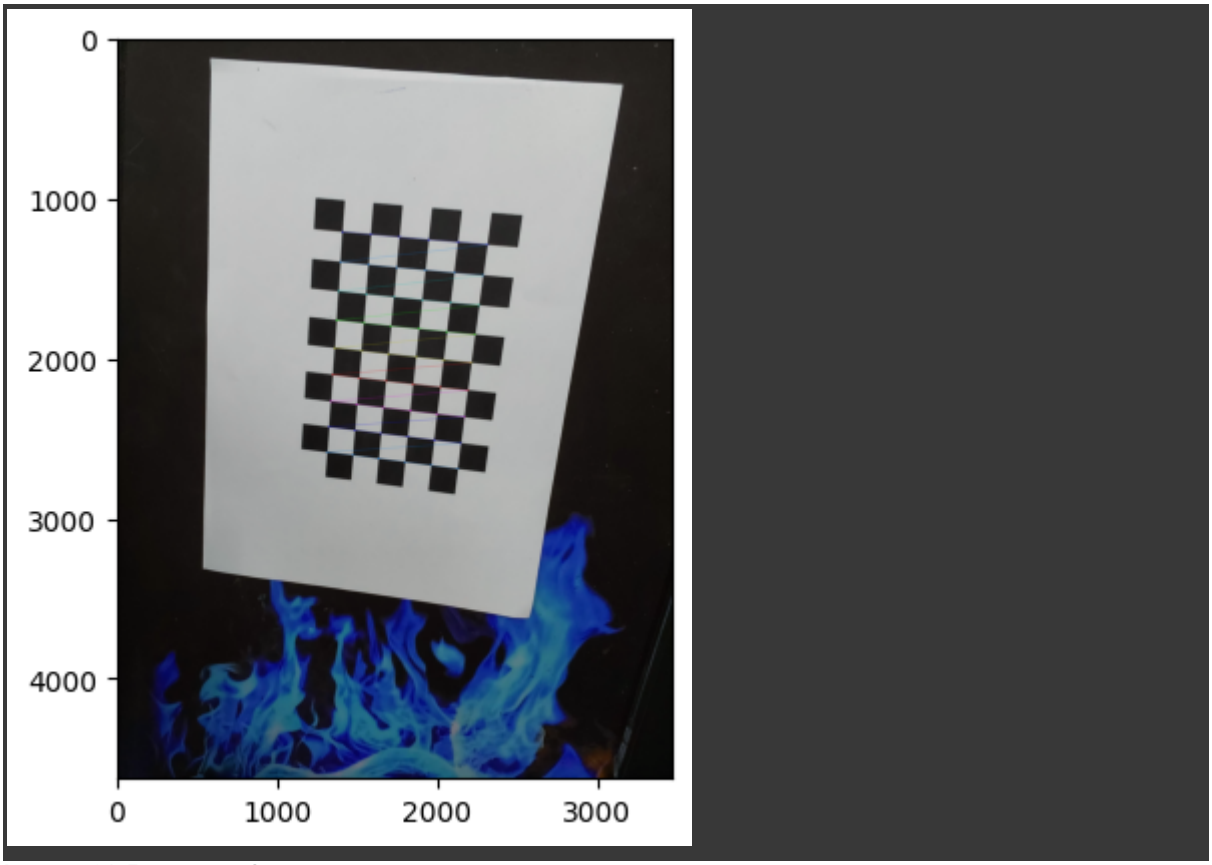
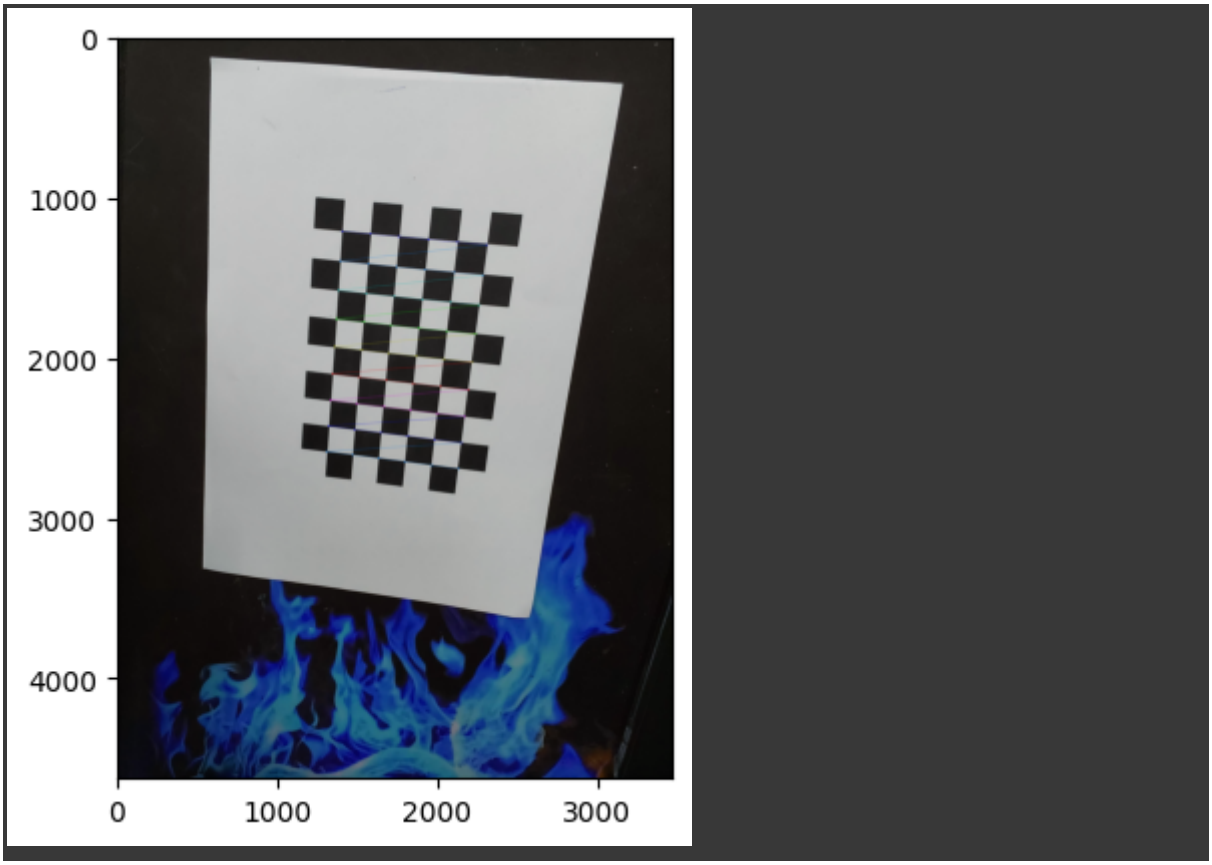


Image 10 Processed



data_cv/20230411_184832.jpg

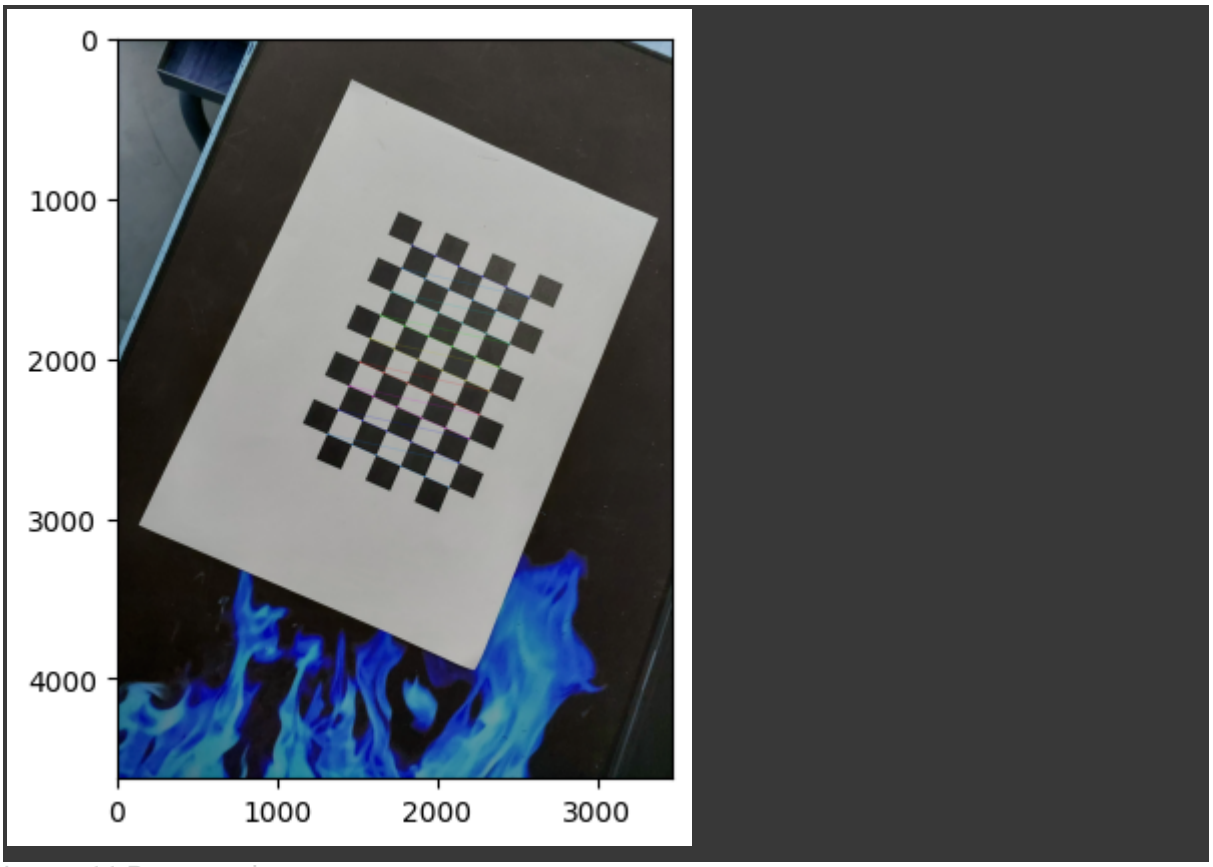
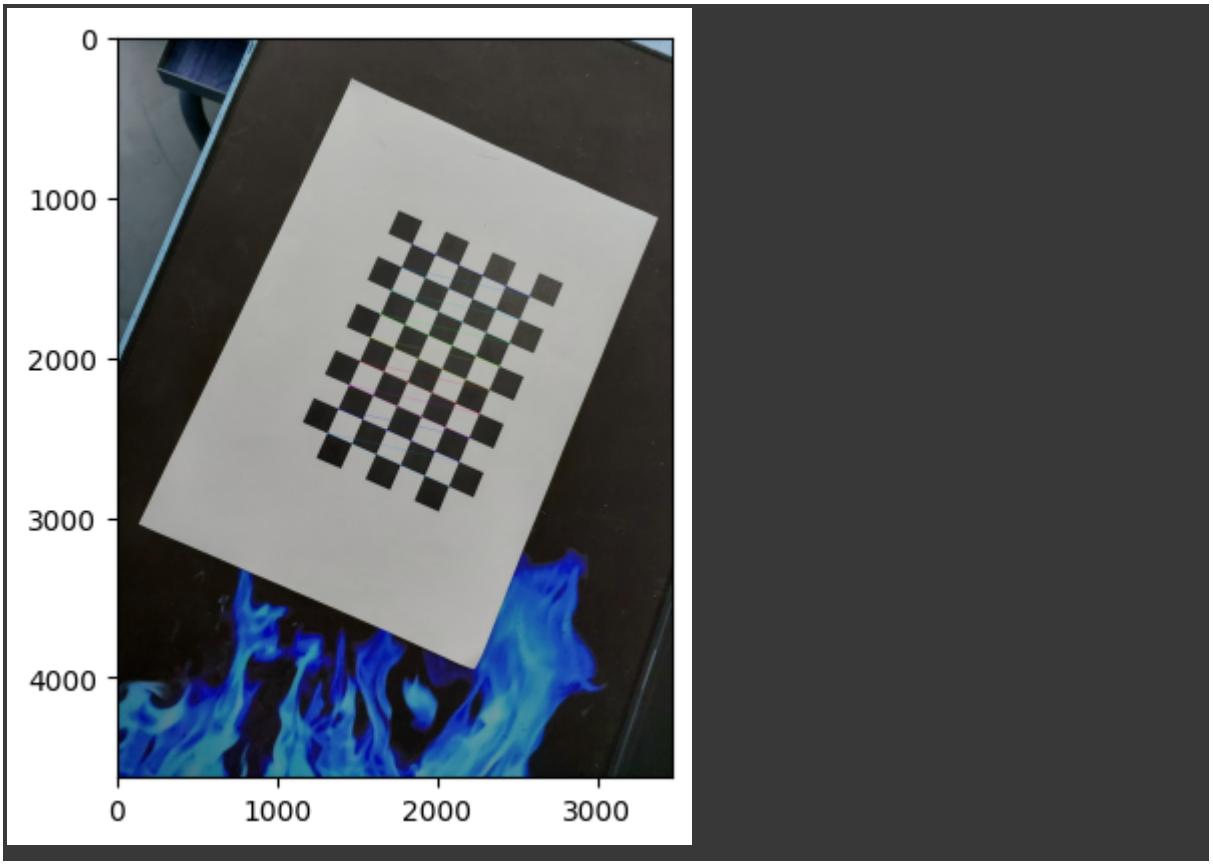


Image 11 Processed



data_cv/20230411_184930.jpg

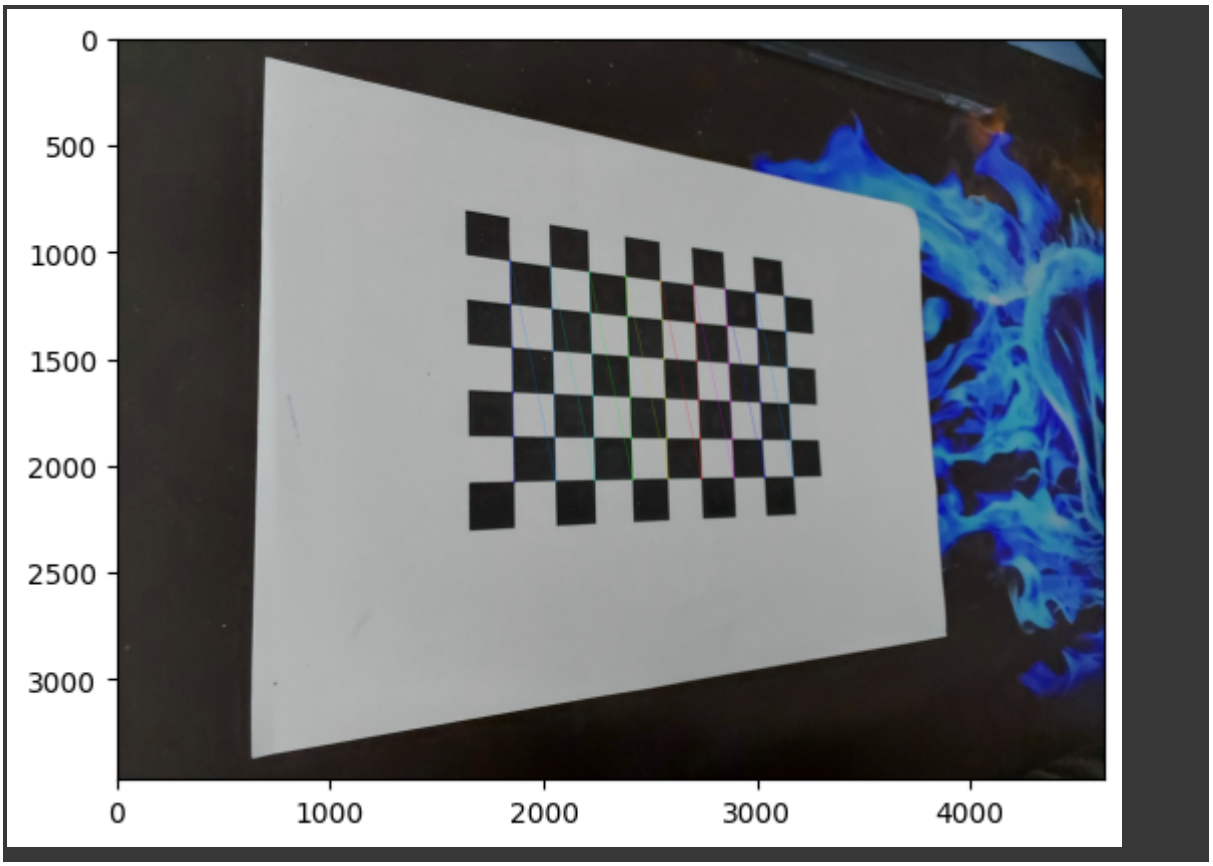
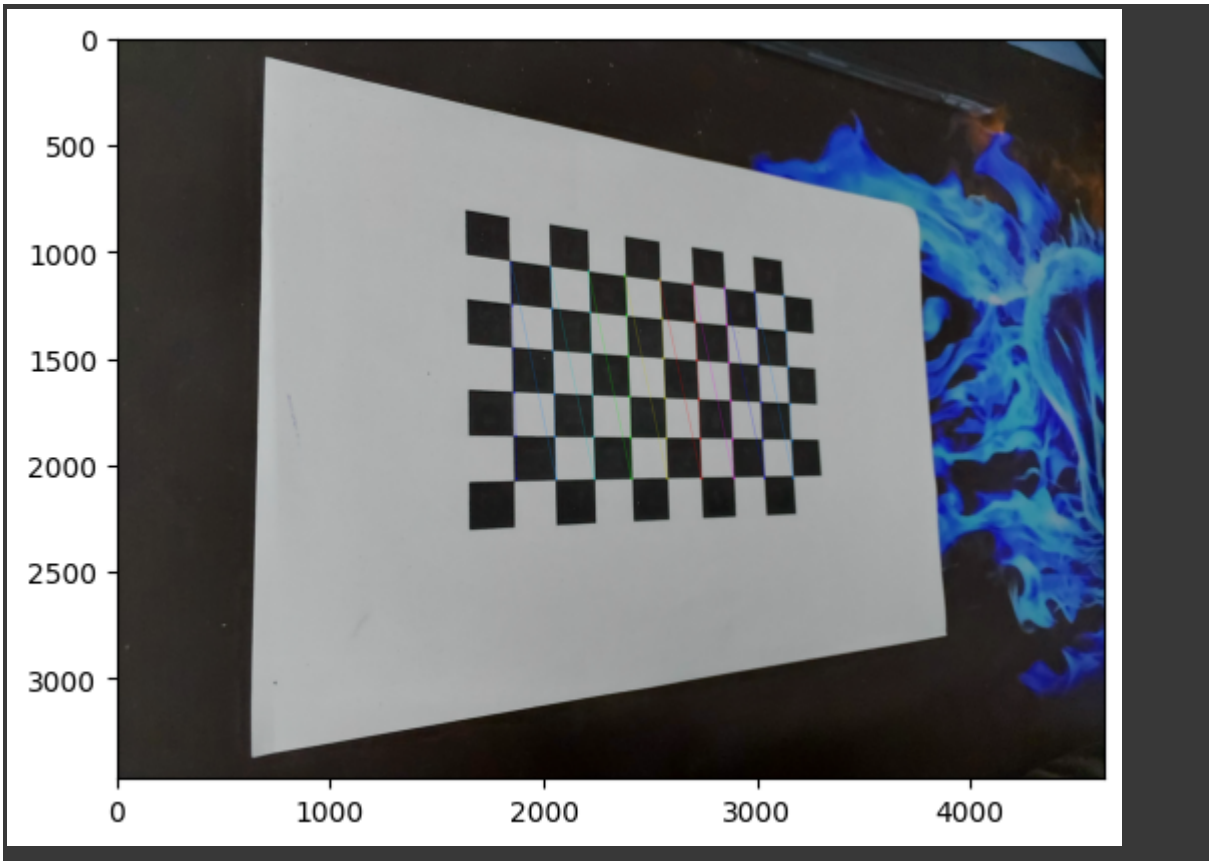


Image 12 Processed



---

data_cv/20230411_184839.jpg

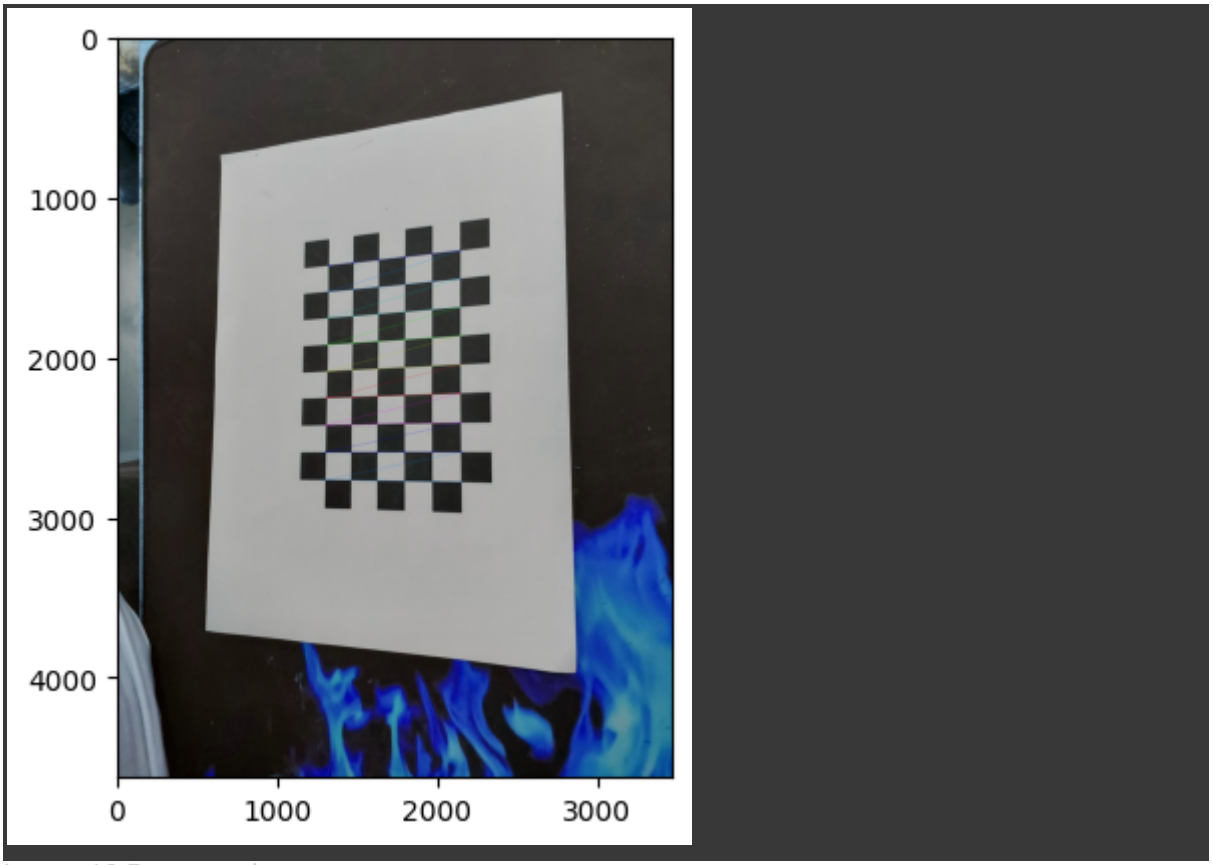
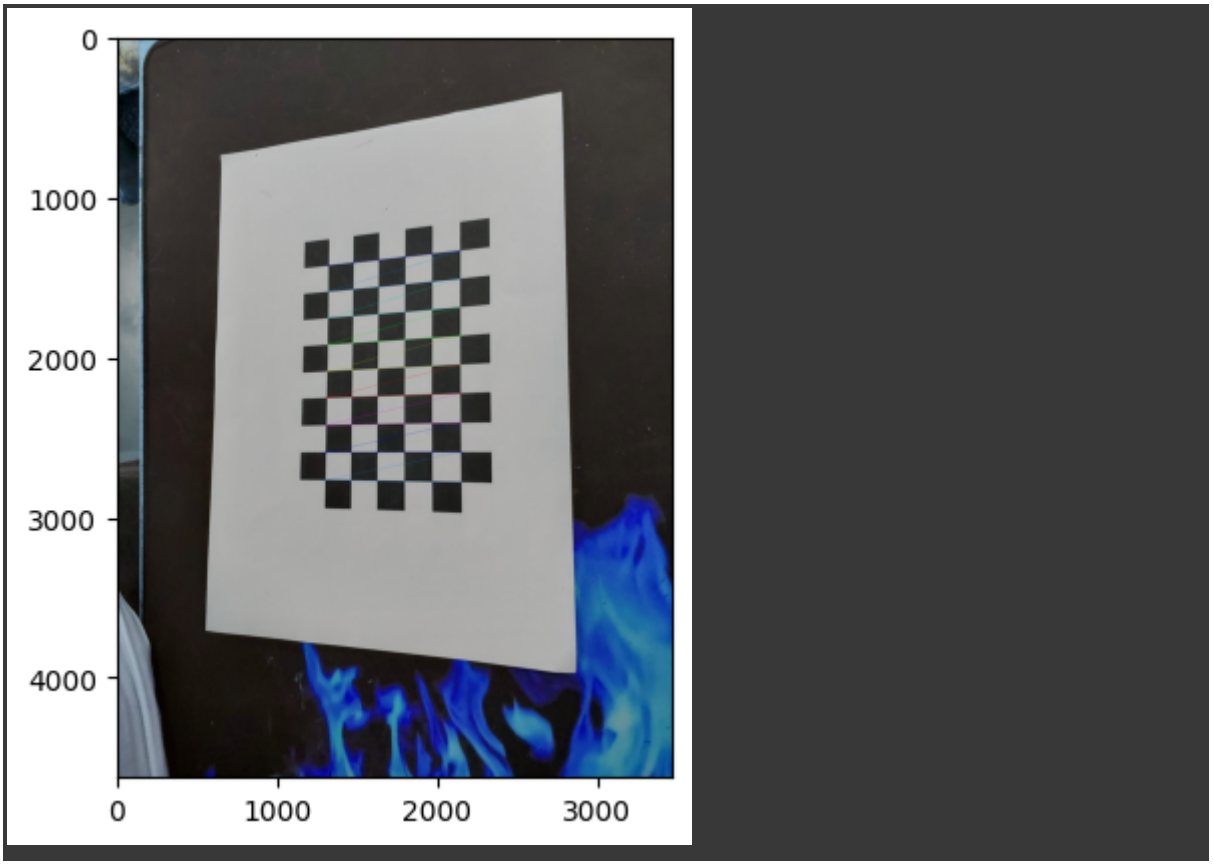


Image 13 Processed



data_cv/20230411_185007.jpg

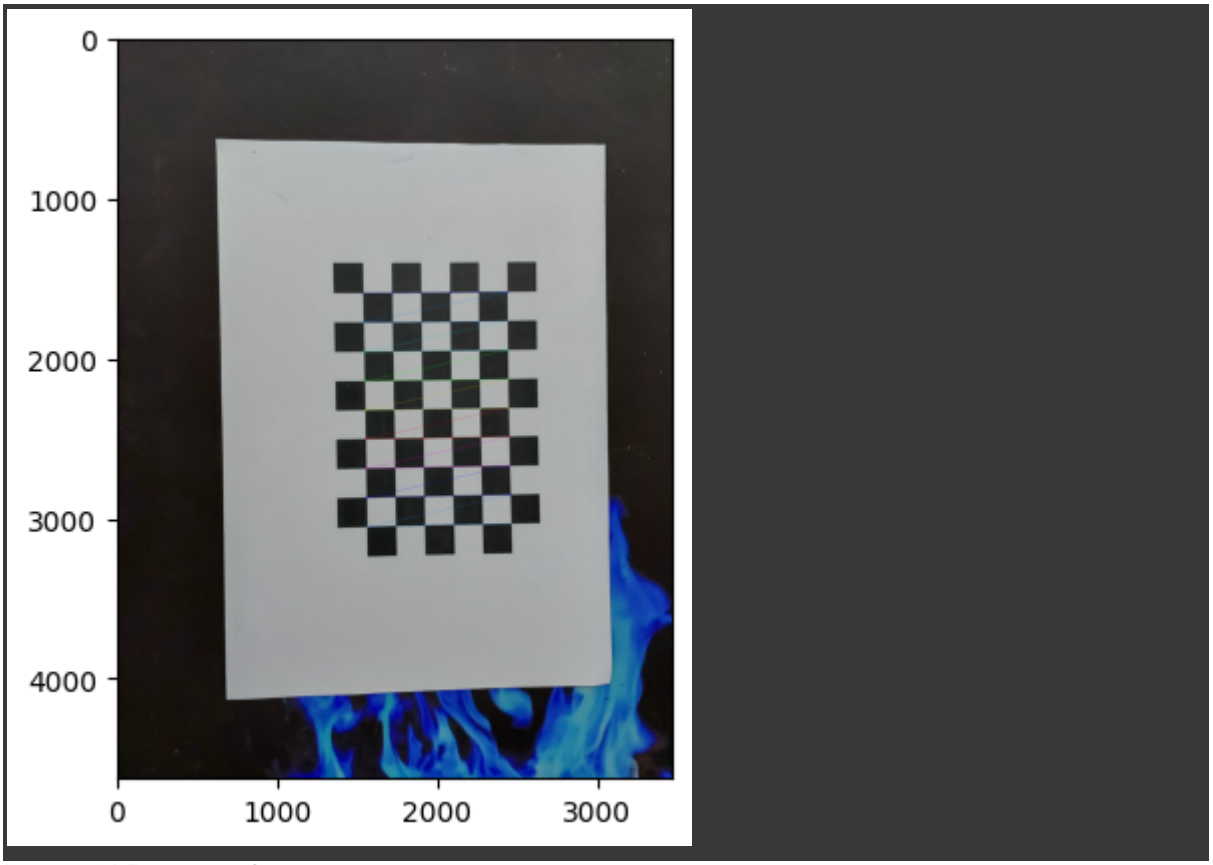
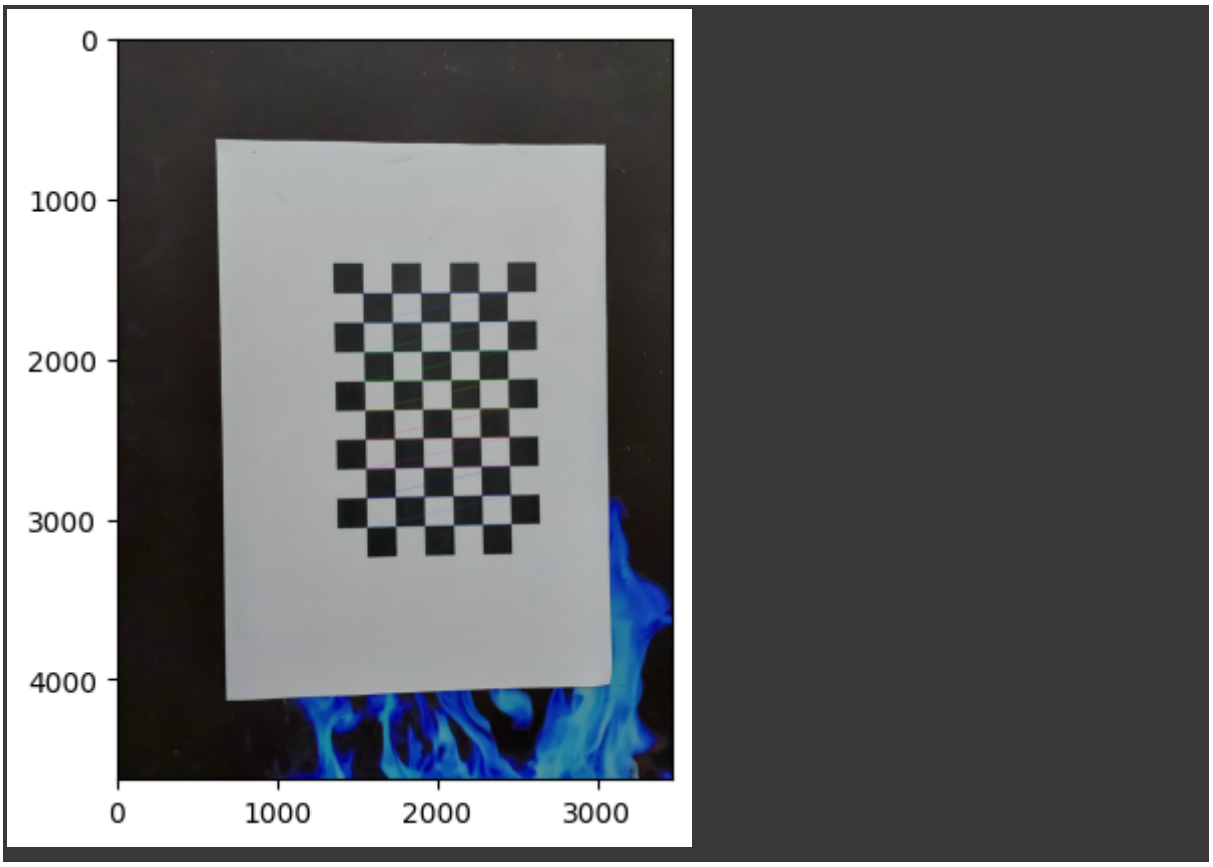


Image 14 Processed



---

data_cv/20230411_184341.jpg

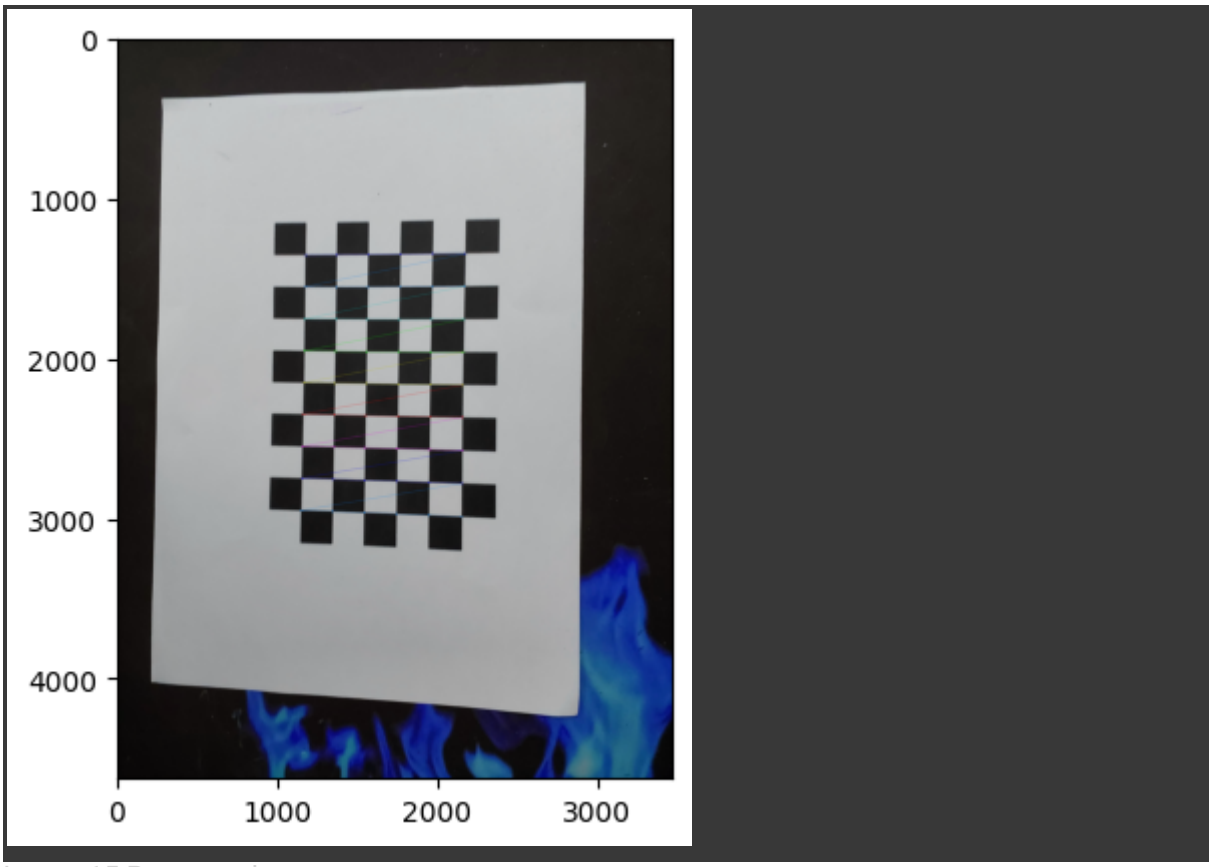
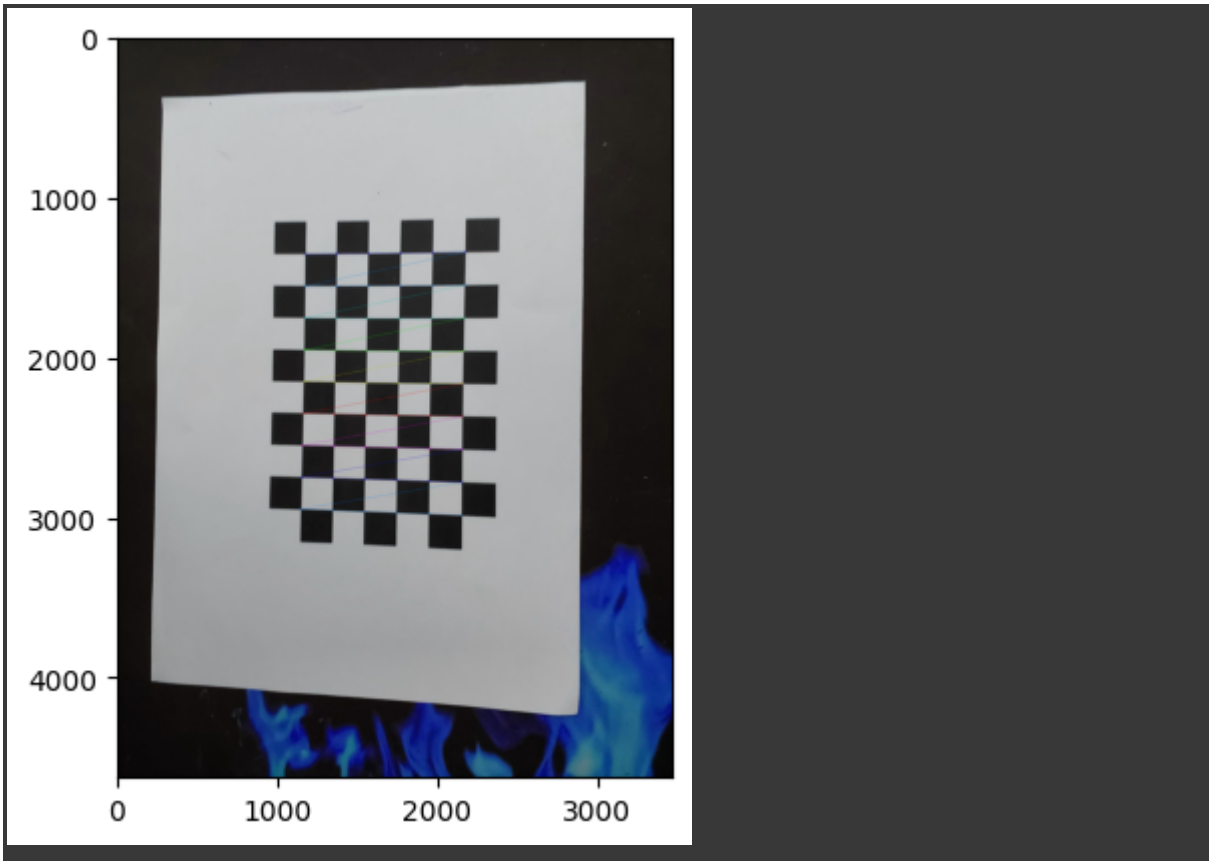


Image 15 Processed



data_cv/20230411_184439.jpg

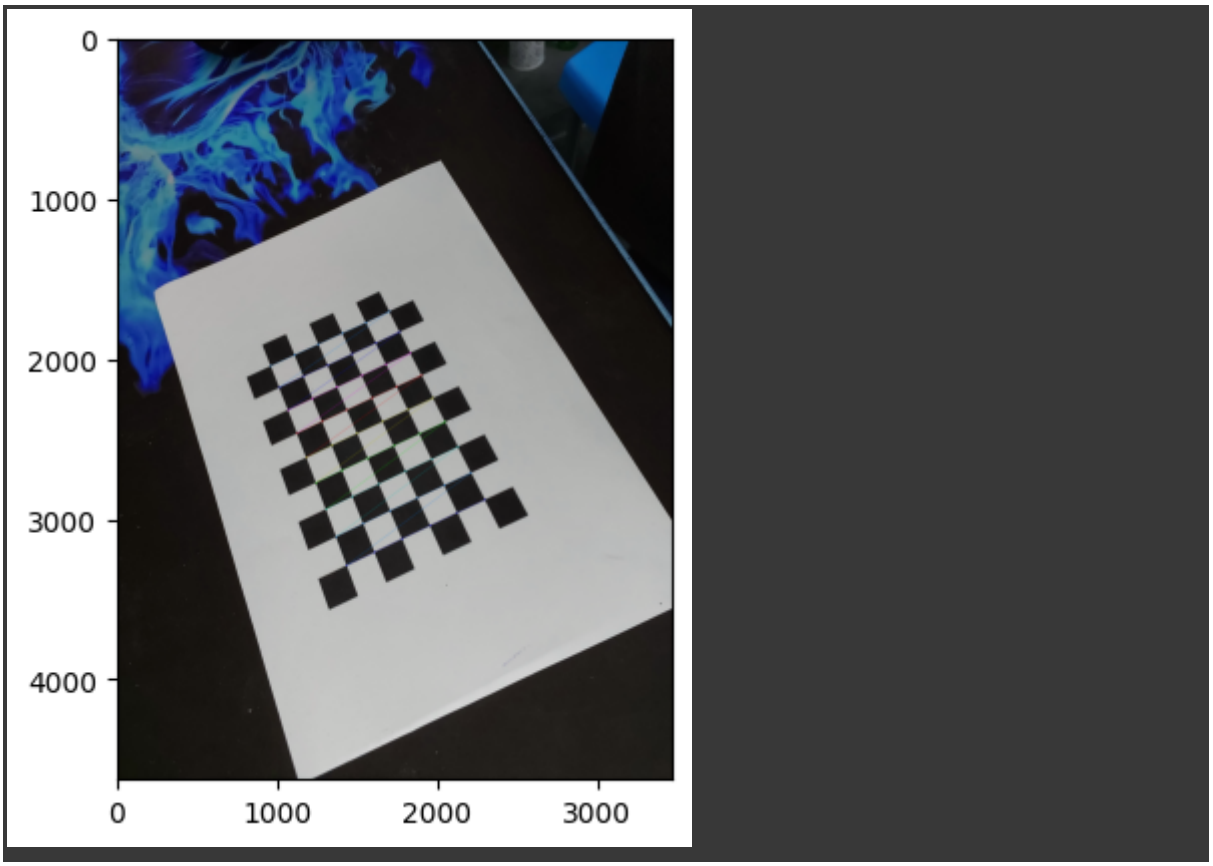
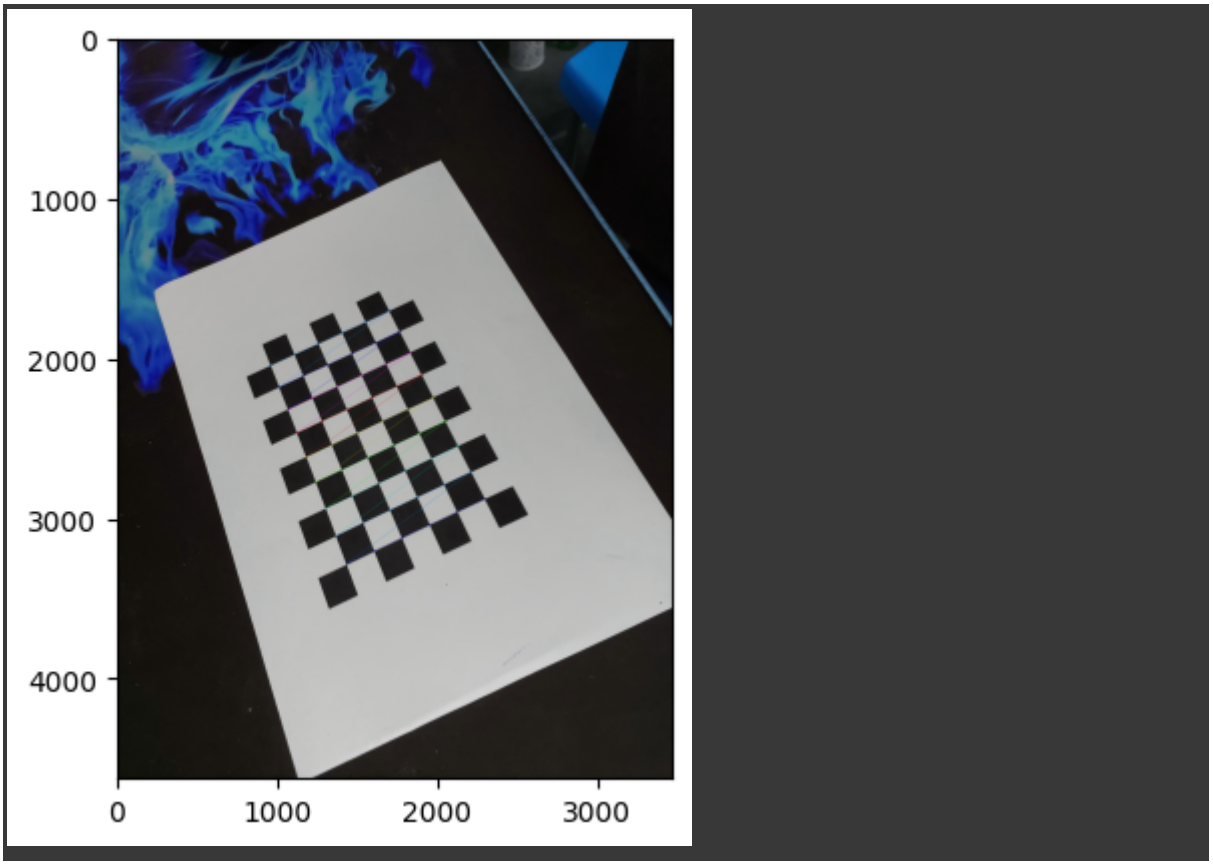


Image 16 Processed



---

data_cv/20230411_184454.jpg

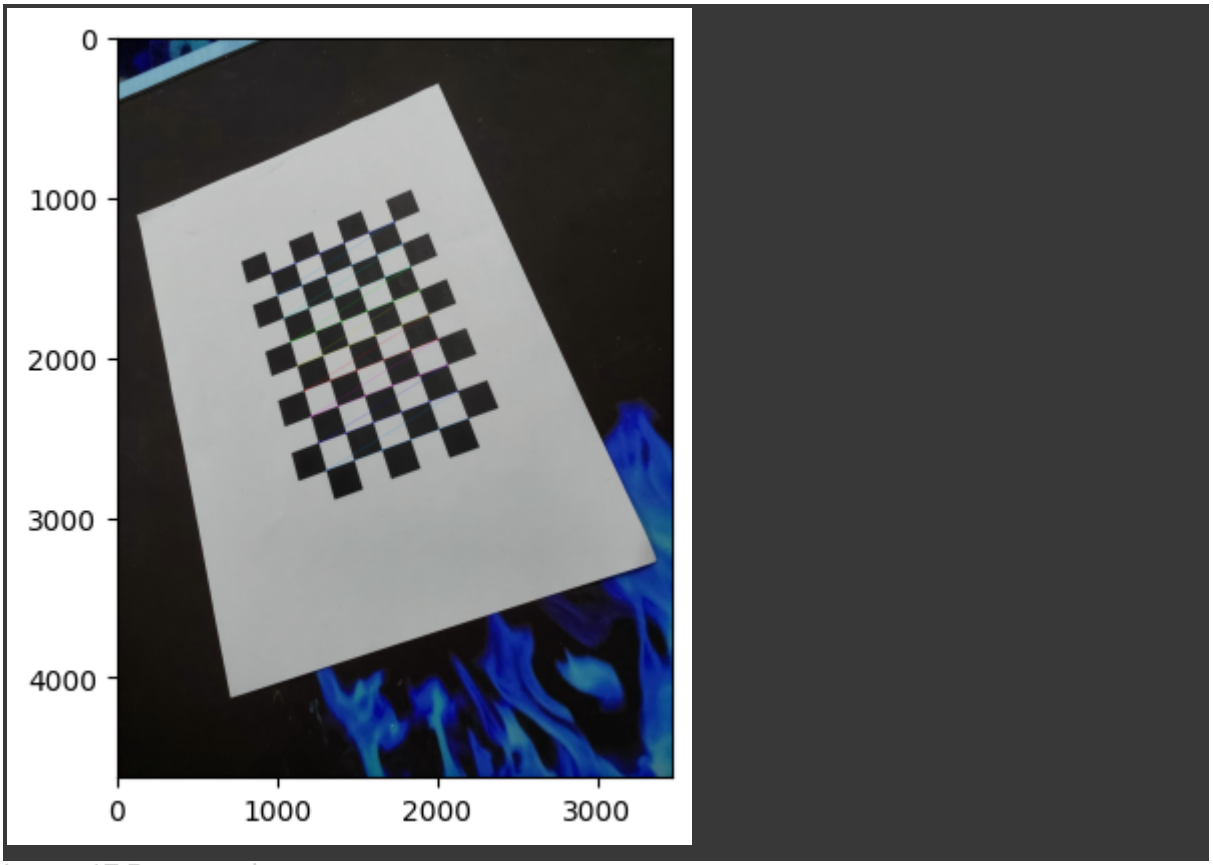
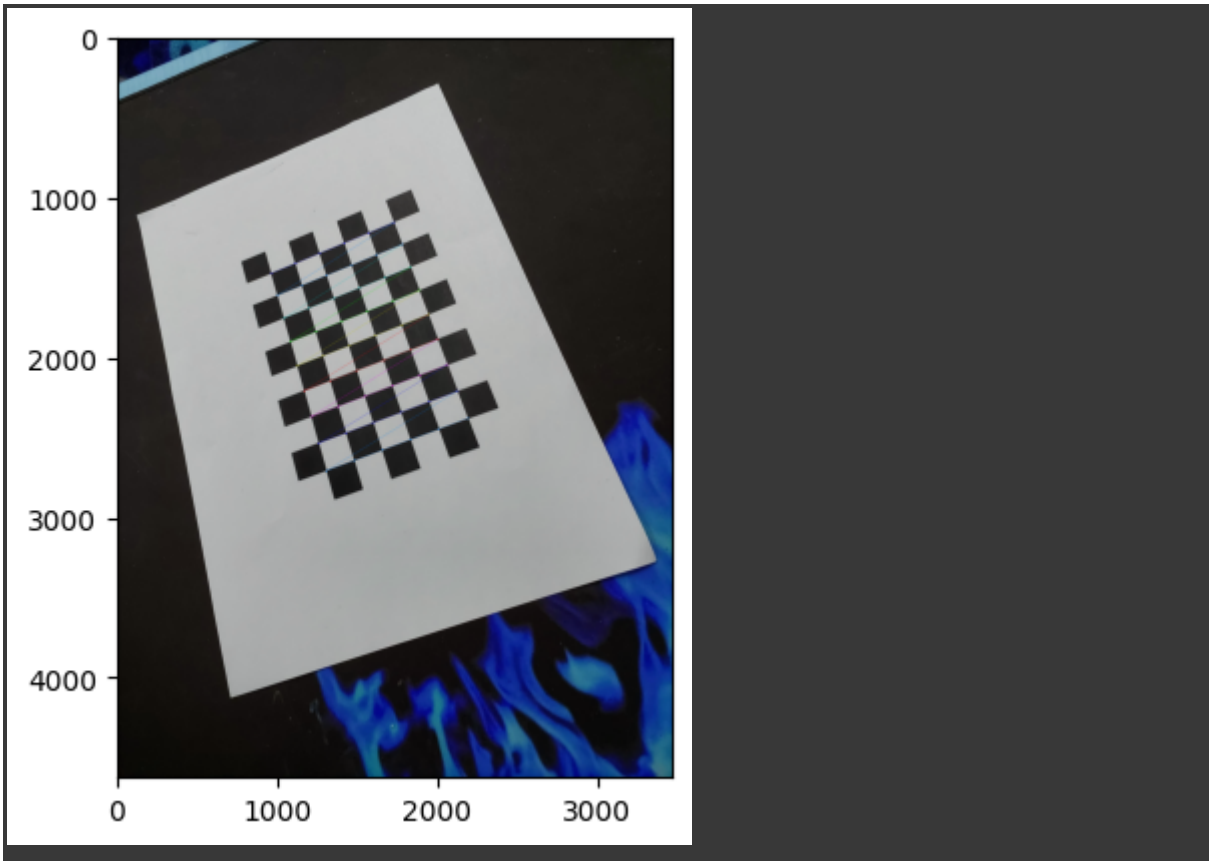


Image 17 Processed



data_cv/20230411_184707.jpg

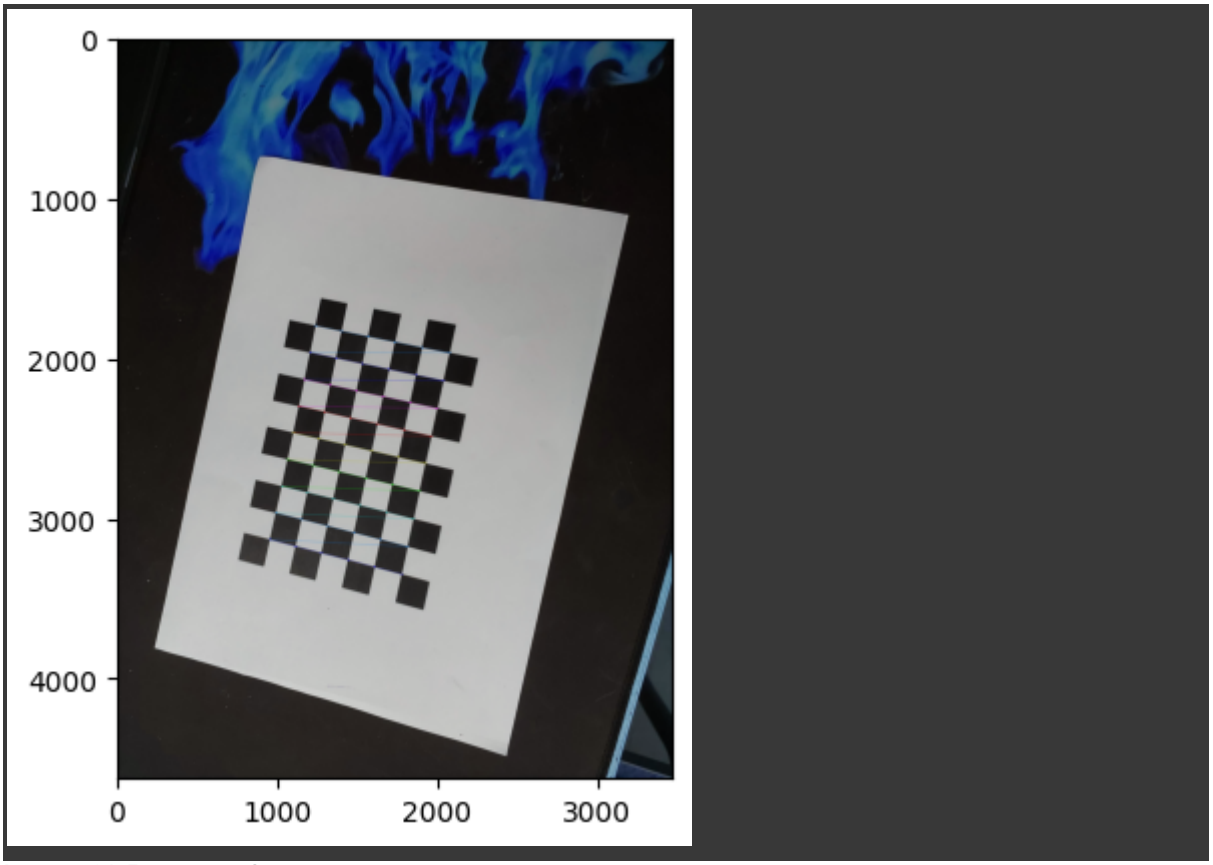
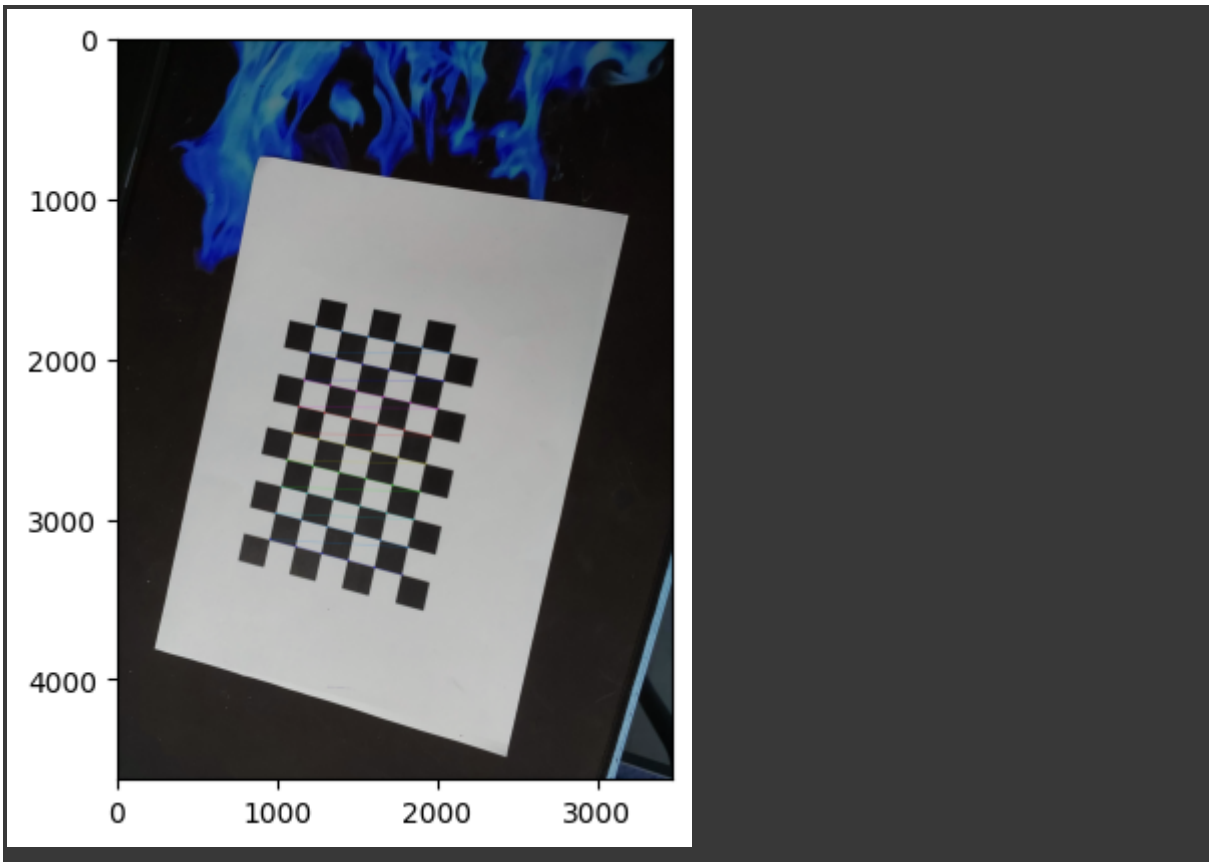


Image 18 Processed



---

data_cv/20230411_184620.jpg

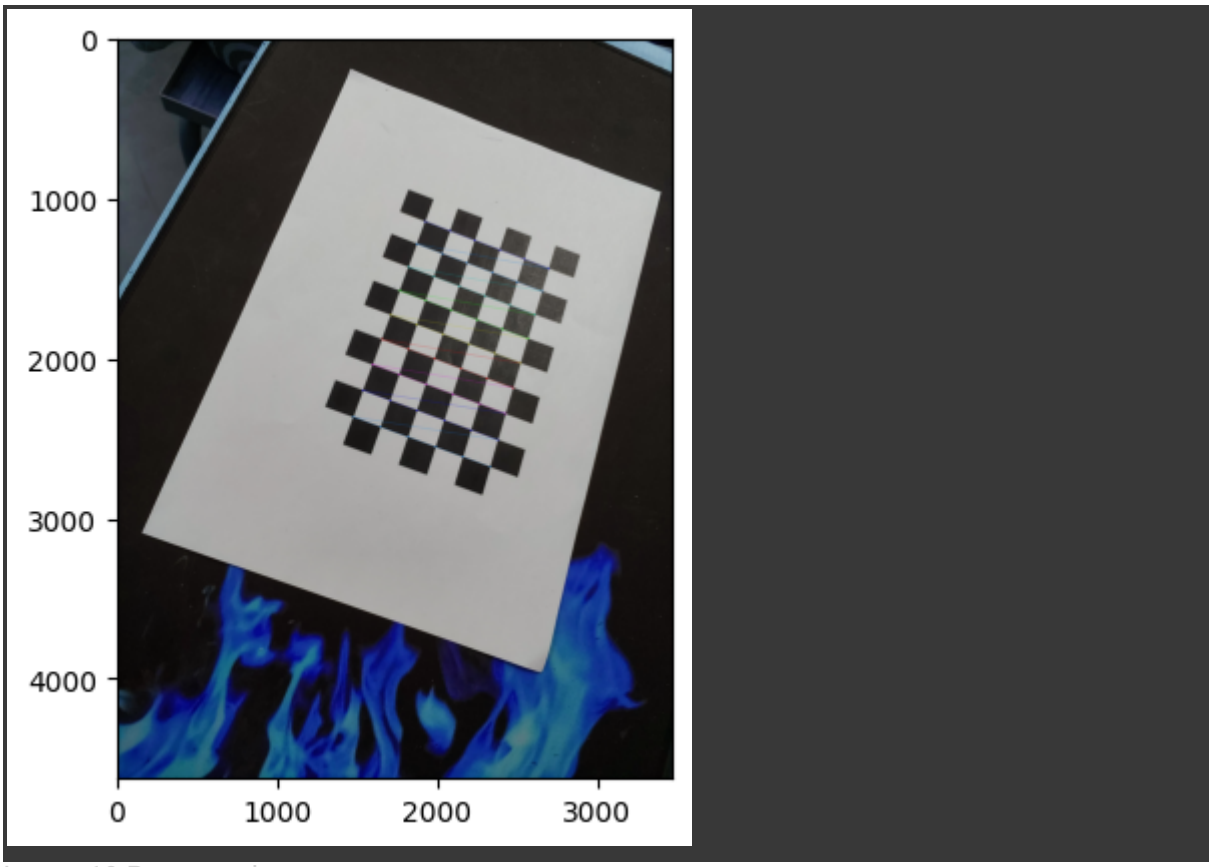
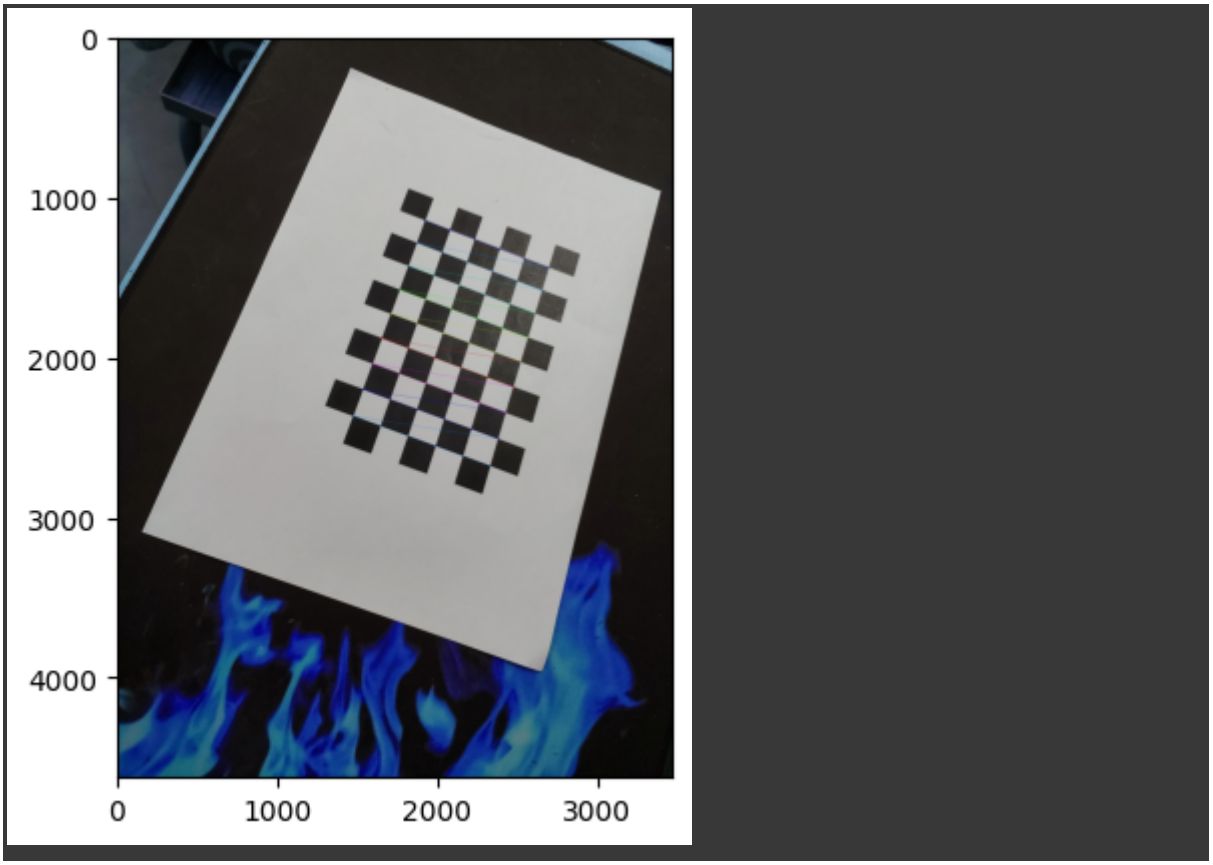


Image 19 Processed



data_cv/20230411_184922.jpg

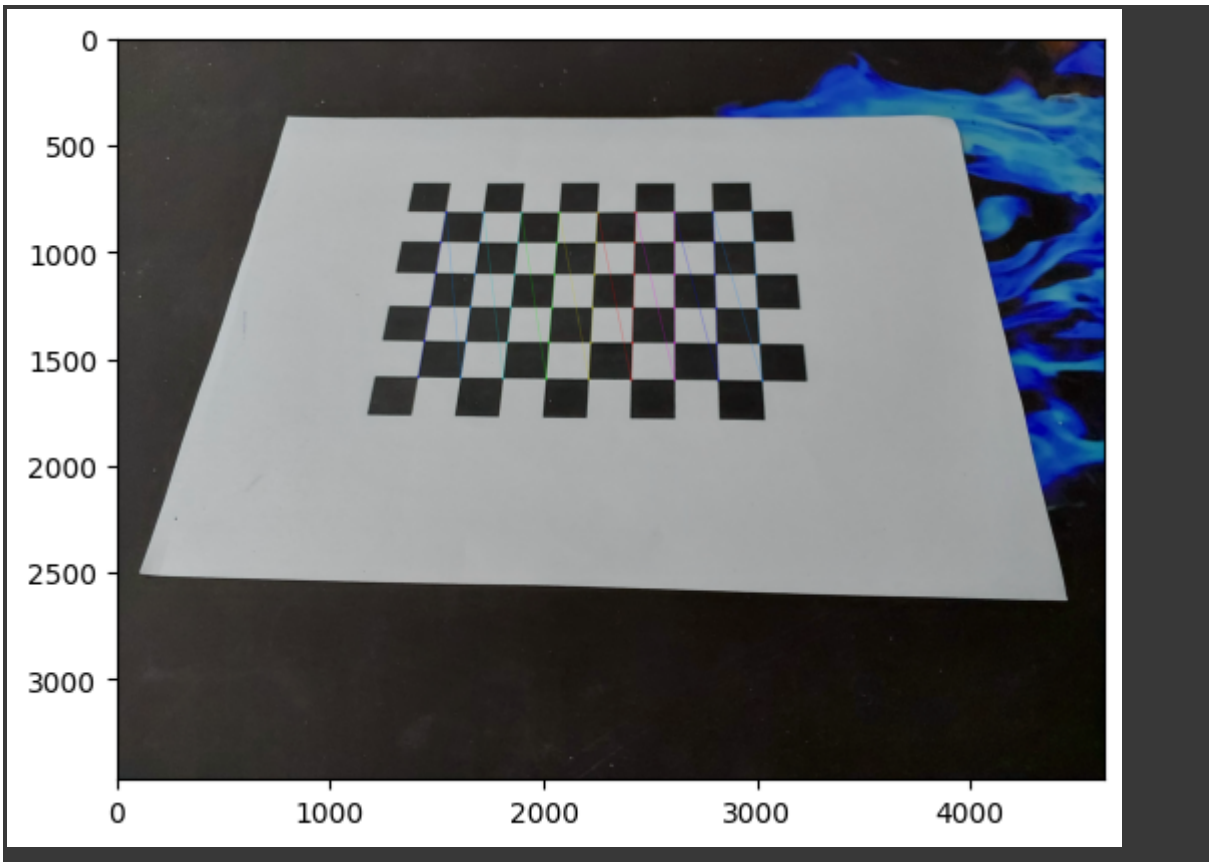
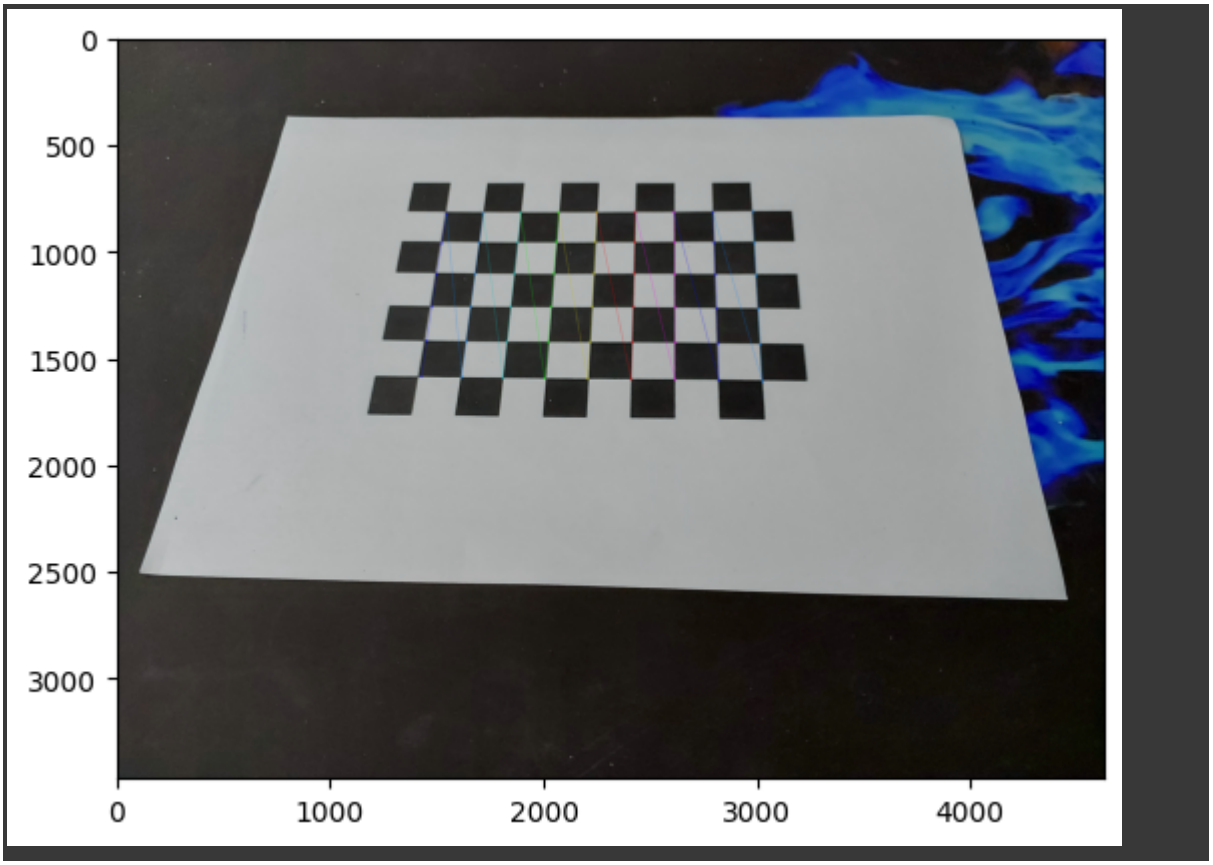


Image 20 Processed



---

data_cv/20230411_184647.jpg

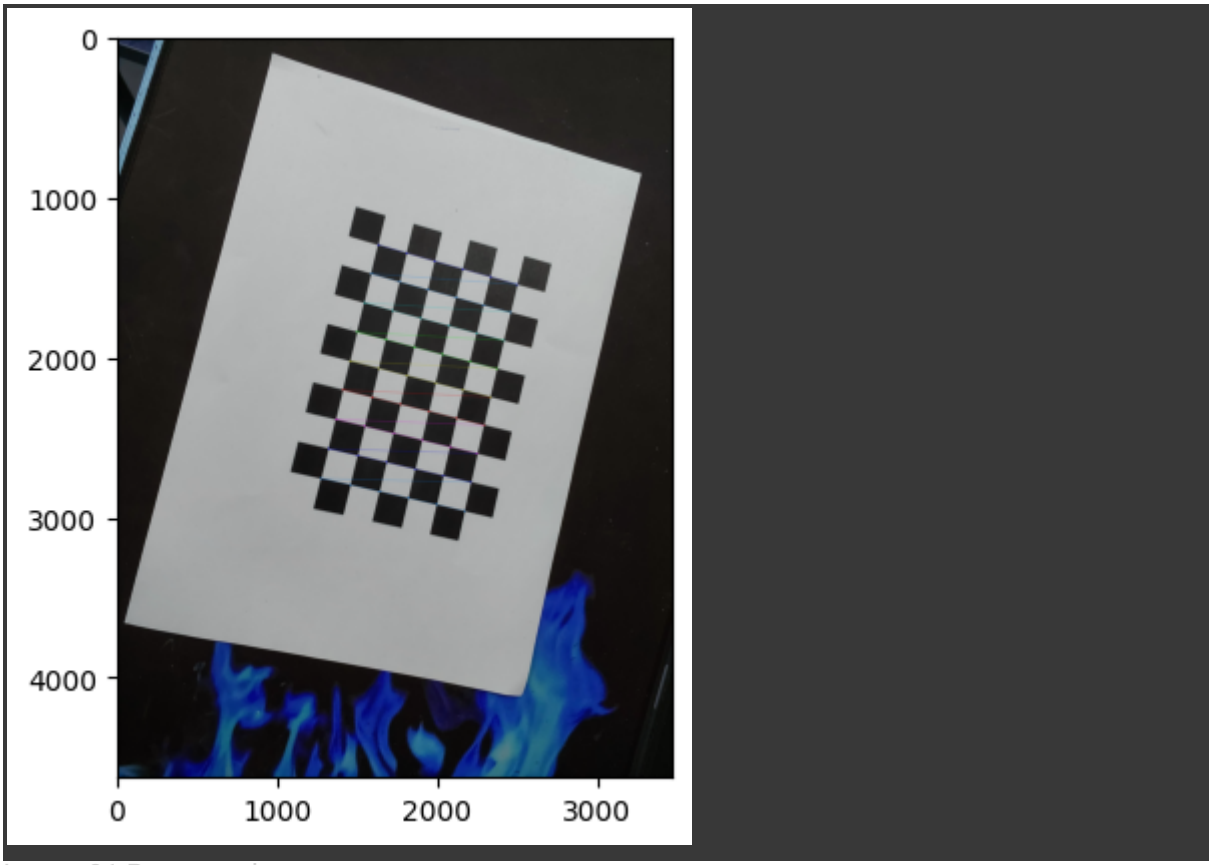
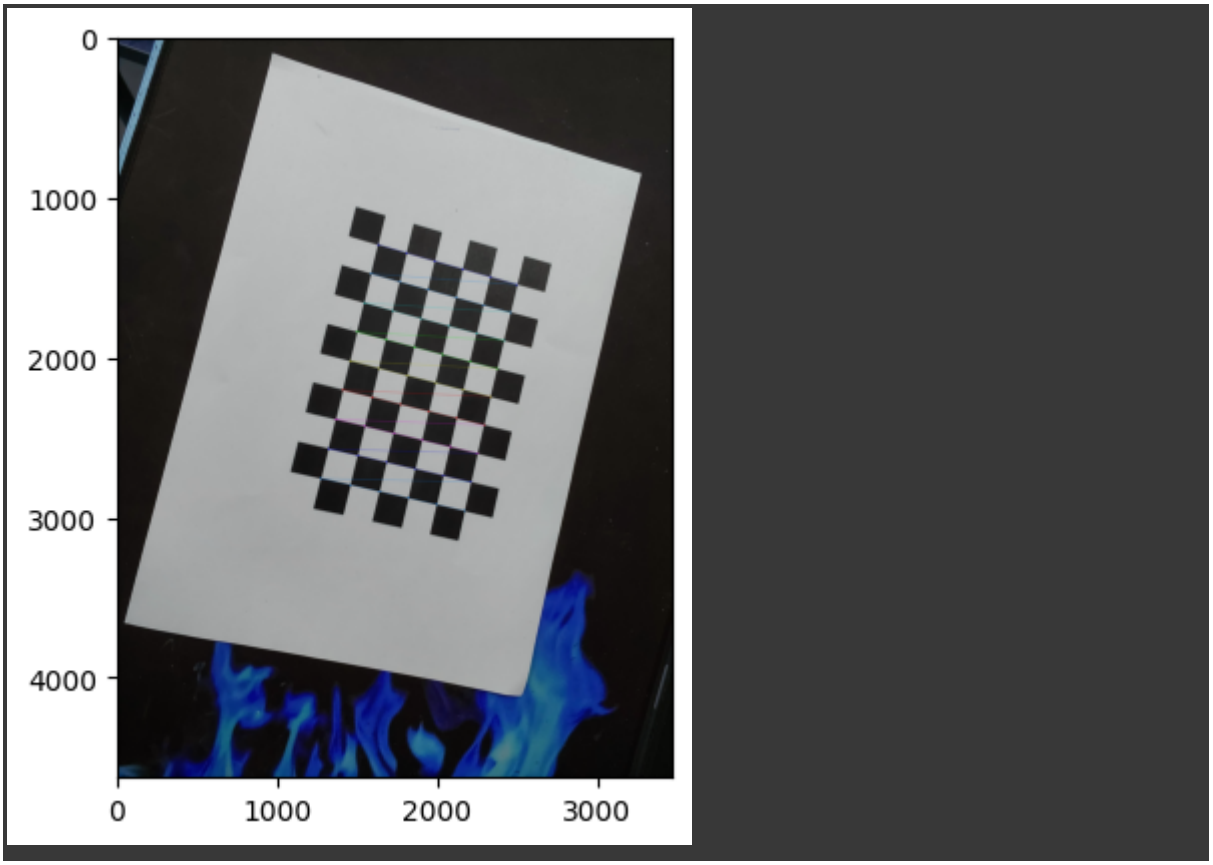


Image 21 Processed



data_cv/20230411_184540.jpg

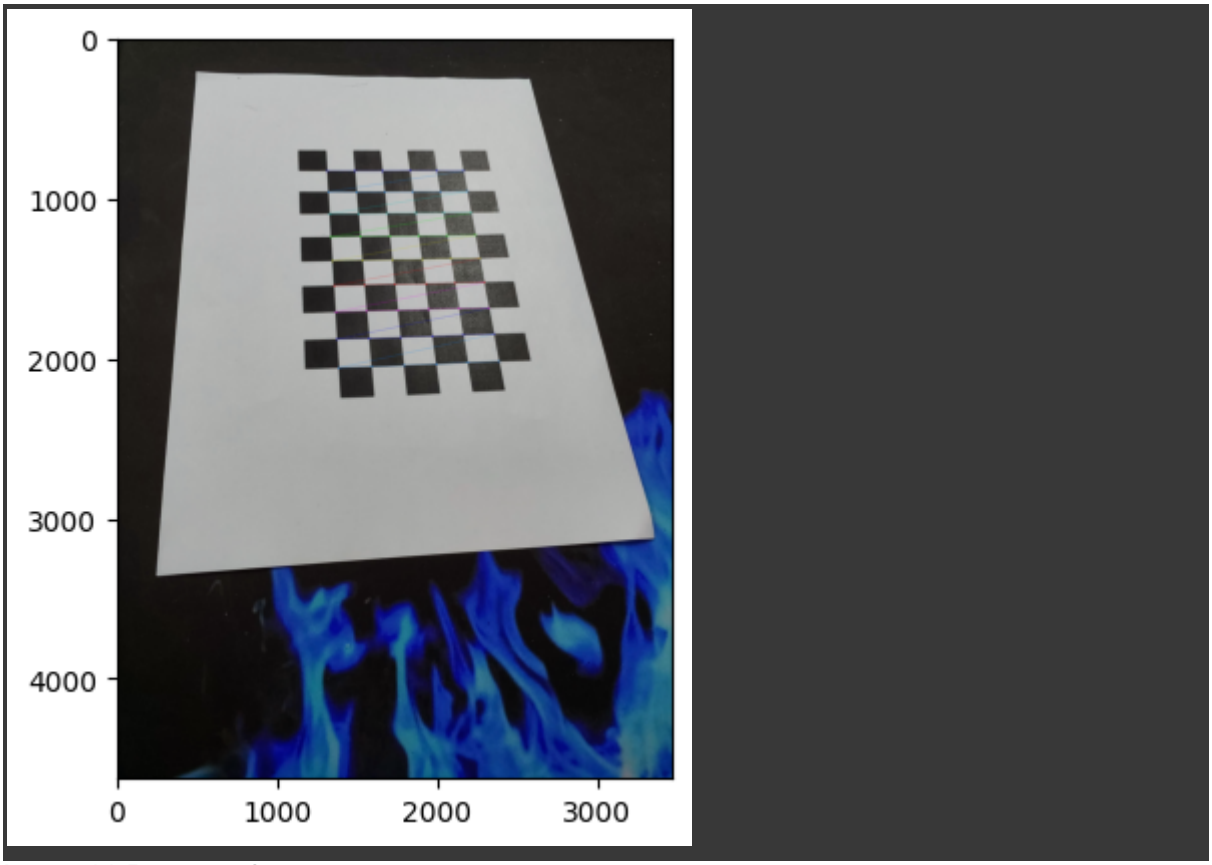
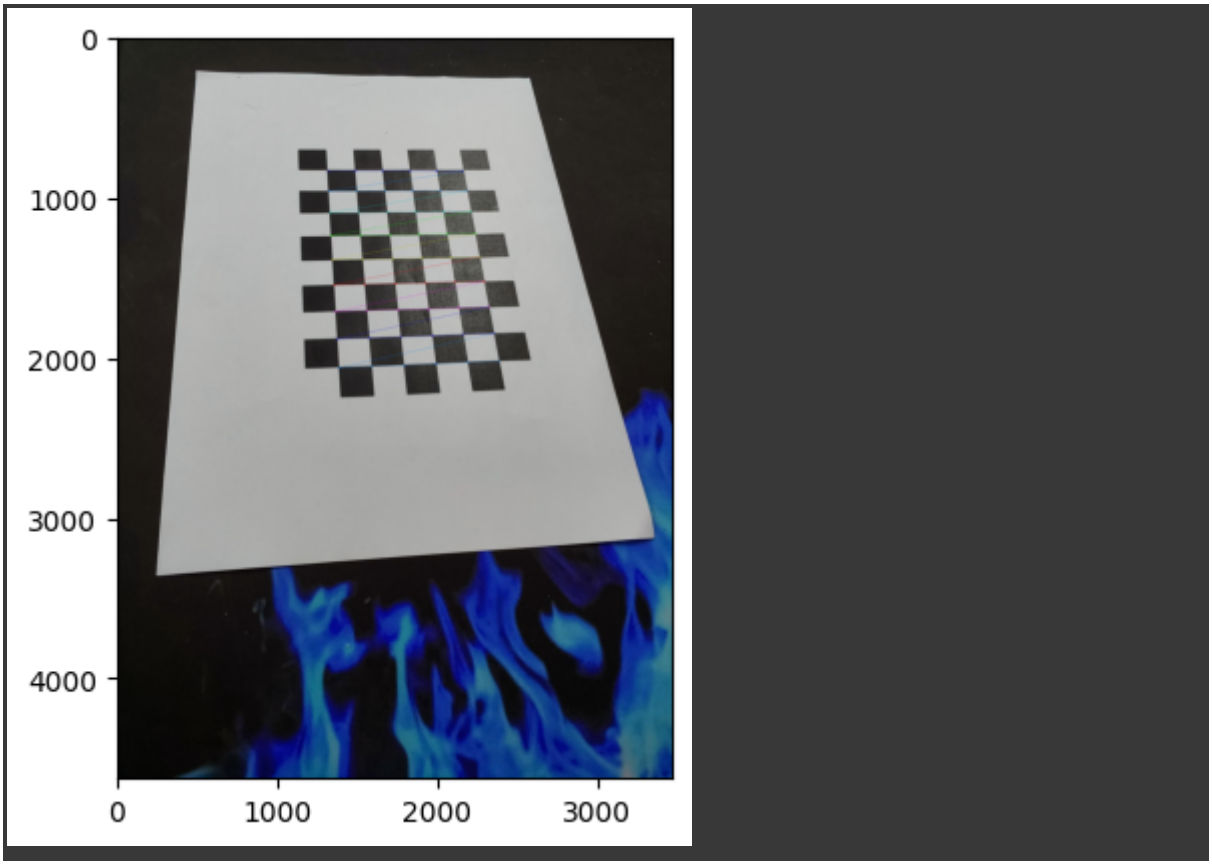


Image 22 Processed



---

data_cv/20230411_184432.jpg

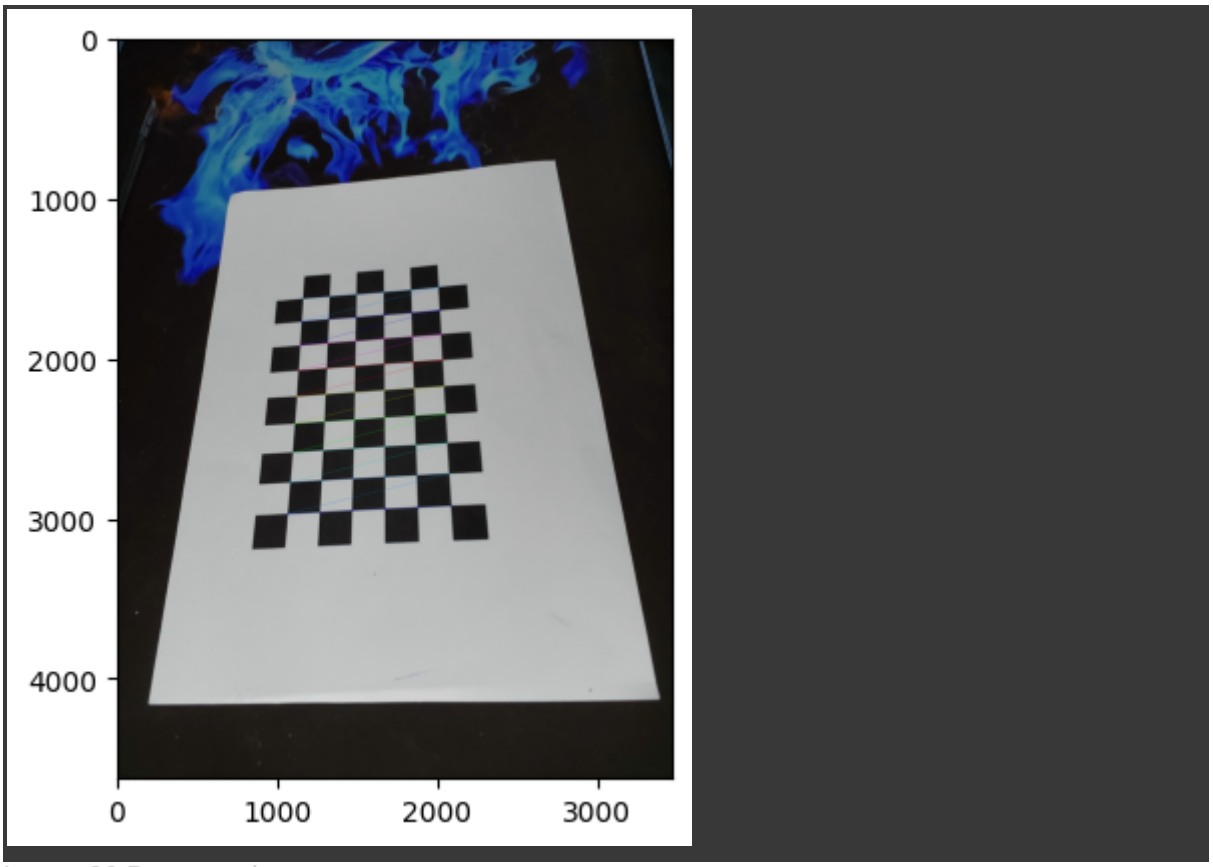
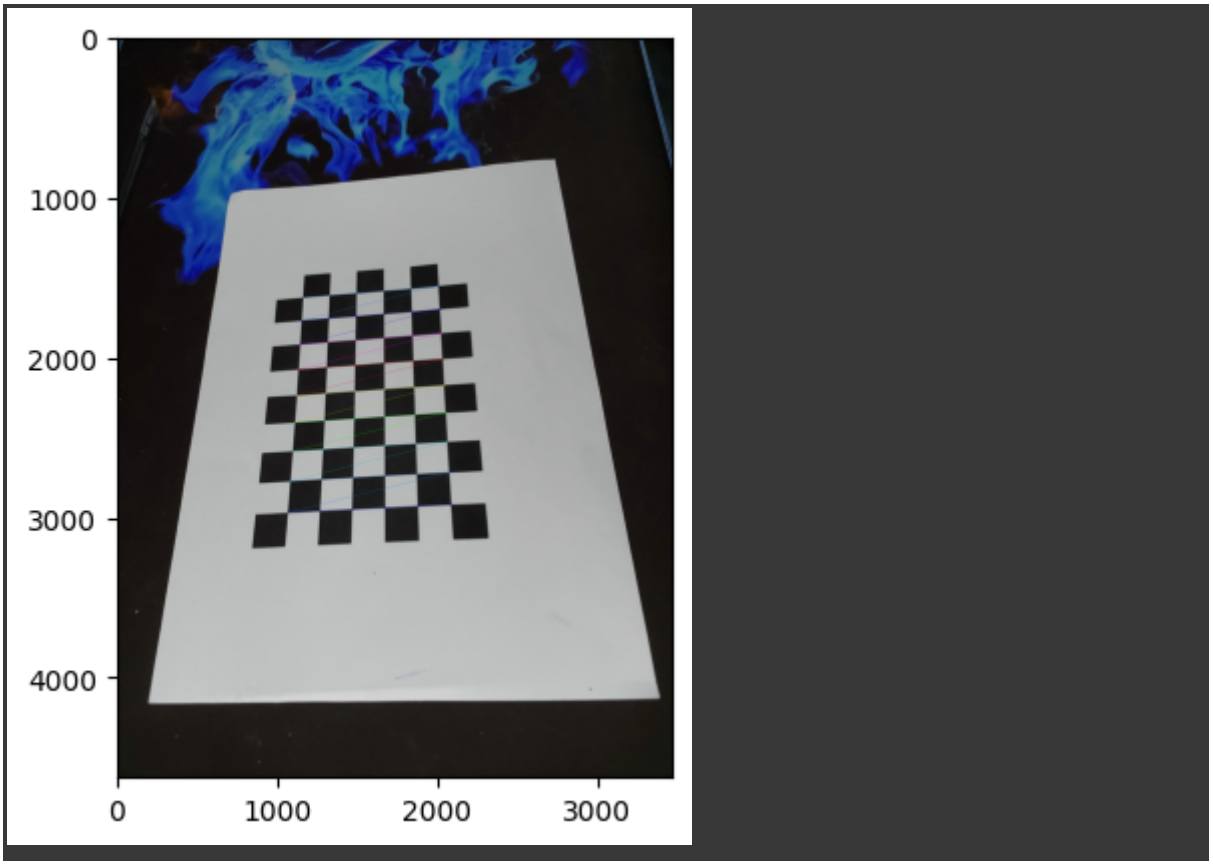


Image 23 Processed



data_cv/20230411_184447.jpg

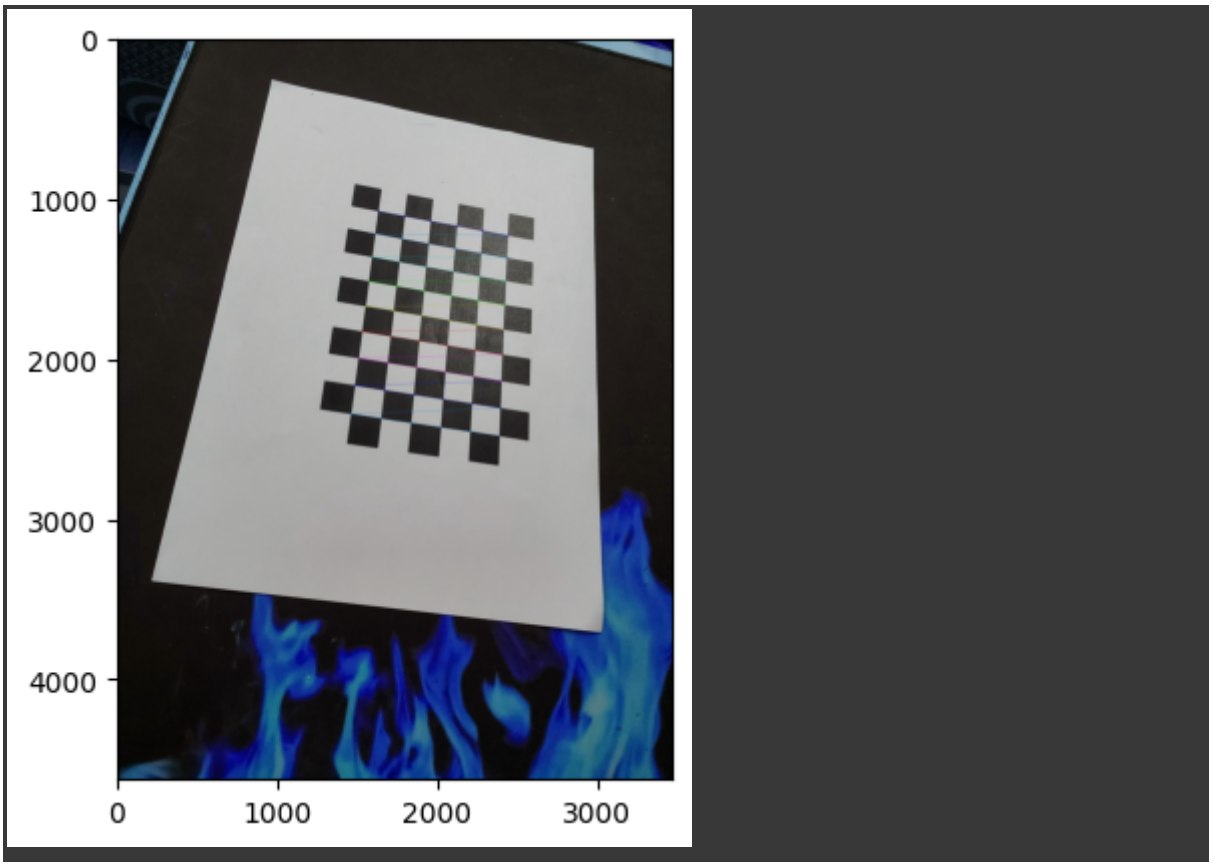
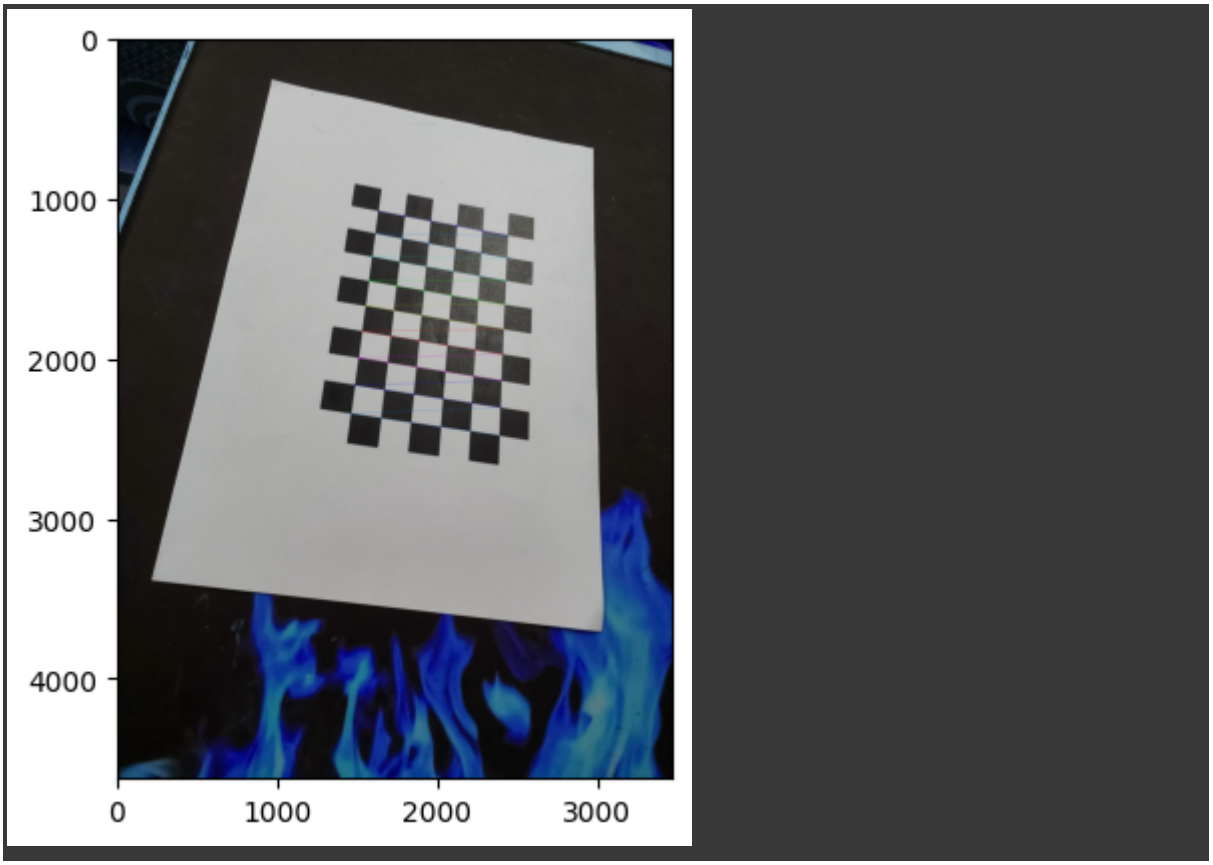


Image 24 Processed



---

data_cv/20230411_184561.jpg

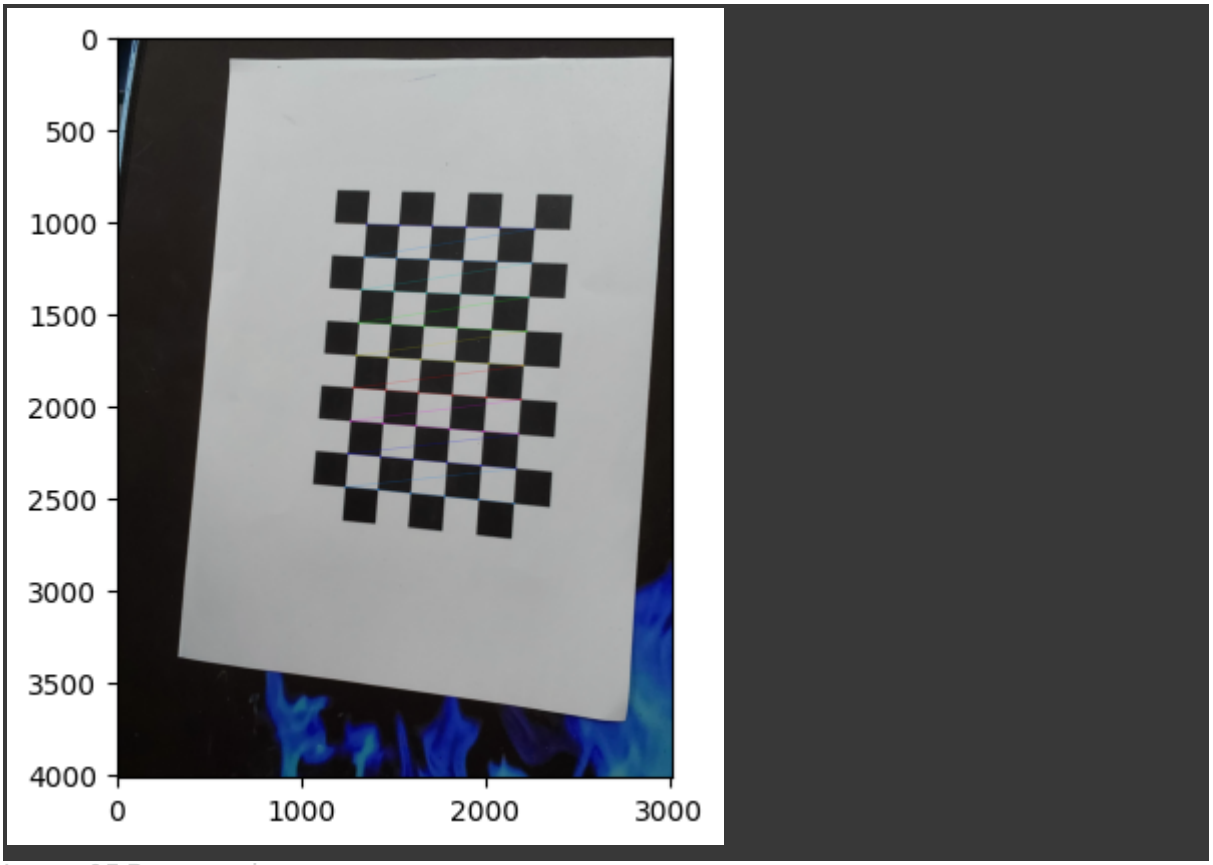
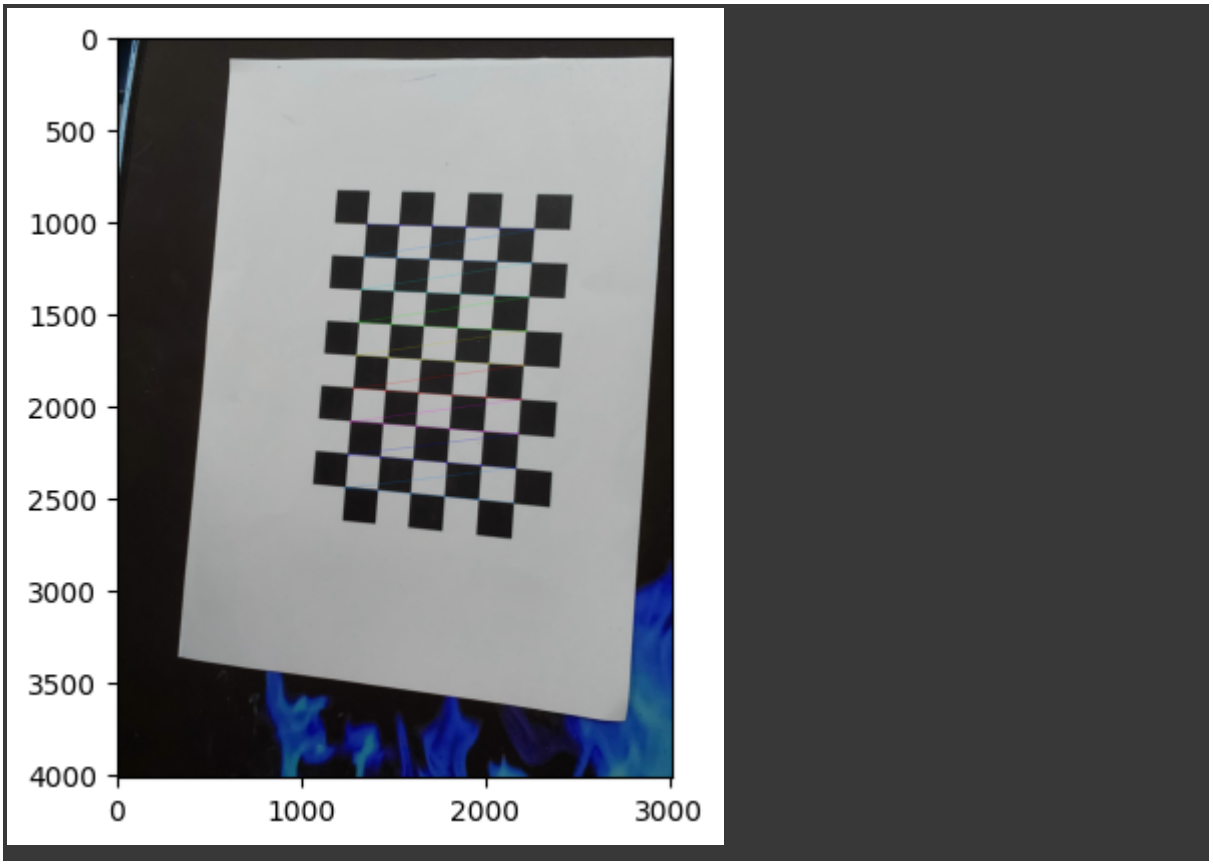


Image 25 Processed



Q6) Compute the checkerboard plane normals for each of the 25 selected images in the camera coordinate frame of reference

```
for i in range(len(images)):
    rmat = cv.Rodrigues(rvecs[i])[0]
    rmat_inv = np.linalg.inv(rmat)
    tvec_inv = rmat_inv.dot(tvecs[i])
    plane_normal = rmat_inv.dot(np.array([0,0,1]))
    print("Plane normal for image ", images[i], "\n")
    print(plane_normal)
```

We loop on each image. The Rodrigues() function converts the rotation vector to the rotation matrix. Then we invert it using numpy, then we multiply the inverse of the rotation matrix by the corresponding translation vector. Then we take the dot product of this vector with a unit vector in the z-direction and finally print it.

```
[-0.05050892 -0.45749562  0.88777621]
Plane normal for image  data_cv/20230411_184633.jpg

[0.12824744 0.03982821 0.99094213]
Plane normal for image  data_cv/20230411_184325.jpg

[0.42639231 0.11504391 0.89719256]
Plane normal for image  data_cv/20230411_184318.jpg

[ 0.14656731 -0.23810783  0.96011597]
Plane normal for image  data_cv/20230411_184502.jpg

[-0.10337527  0.20446046  0.97340098]
Plane normal for image  data_cv/20230411_184832.jpg

[-0.00391103 -0.11219743  0.99367824]
Plane normal for image  data_cv/20230411_184930.jpg

[0.03420381 0.31247281 0.94931072]
Plane normal for image  data_cv/20230411_184839.jpg

[-0.31164191 -0.12479961  0.94196835]
Plane normal for image  data_cv/20230411_185007.jpg

[ 0.04729184 -0.02647019  0.99853033]
Plane normal for image  data_cv/20230411_184341.jpg

[-0.12336568 -0.03000081  0.99190769]
Plane normal for image  data_cv/20230411_184439.jpg
```

```
[-0.0137344   0.39404009  0.91899063]
Plane normal for image  data_cv/20230411_184454.jpg

[-0.10221657 -0.35100491  0.93077781]
Plane normal for image  data_cv/20230411_184707.jpg

[0.17091584 0.00852581 0.98524874]
Plane normal for image  data_cv/20230411_184620.jpg

[ 0.02829985 -0.27851575  0.96001463]
Plane normal for image  data_cv/20230411_184922.jpg

[ 0.35824621 -0.1750718   0.91706571]
Plane normal for image  data_cv/20230411_184647.jpg

[ 0.12702361 -0.06580093  0.98971473]
Plane normal for image  data_cv/20230411_184540.jpg

[ 0.11102773 -0.4186386   0.90134043]
Plane normal for image  data_cv/20230411_184432.jpg

[0.11190818 0.4435167  0.88925221]
Plane normal for image  data_cv/20230411_184447.jpg

[ 0.10213833 -0.36210579  0.92652423]
Plane normal for image  data_cv/20230411_184561.jpg

[-0.13610618 -0.05136668  0.9893617 ]
```

Q2:

Q1) Compute the chessboard plane normals and the corresponding offsets using the planar LIDAR points in each of the pcd files. You can estimate these by making use of singular value decomposition.

```
images = glob.glob('lidar_data/*.pcd')

for fname in images:
    print(fname)
    ply_point_cloud = o3d.data.PLYPointCloud()
    pcd = o3d.io.read_point_cloud(fname)
    print(pcd, "\n")
    U, s, Vd = np.linalg.svd(np.asarray(pcd.points) -
np.mean(np.asarray(pcd.points), axis=0))
    print("\nNormal Vector:", Vd[-1])
    offset = np.dot(Vd[-1], np.mean(np.asarray(pcd.points), axis=0))
    print("Offset:", offset)
```

```
    print("\n-------------------------------\n")
```

We collect the lidar images using glob library. Next, we loop on each of the LIDAR images. The o3d.data.PLYPointCloud() function creates an empty PLY Point Cloud object. The o3d.io.read_point_cloud reads the PCD file and stores it. Next, we use the svd() function of the numpy library, which computes the Singular Value Decomposition of the PCD image.

The offset is calculated by taking the normal vector (Vd) dot product and the mean/centroid of the PCD points of the image.

```
lidar_data/frame_1075.pcd
PointCloud with 153 points.


Normal Vector: [ 0.70169742 -0.70118932  0.1263102 ]
Offset: 4.720779659728763

-------------------------------

lidar_data/frame_2030.pcd
PointCloud with 42 points.


Normal Vector: [0.94902101 0.13112718 0.28664401]
Offset: 9.49902545111904

-------------------------------

lidar_data/frame_1195.pcd
PointCloud with 93 points.


Normal Vector: [0.94151039 0.18949708 0.27865578]
Offset: 6.323572215248566

-------------------------------

lidar_data/frame_1163.pcd
PointCloud with 91 points.


Normal Vector: [ 0.91935323 -0.3607026   0.15710911]
Offset: 5.70252457528633

-------------------------------

lidar_data/frame_2717.pcd
PointCloud with 44 points.
```

```
Normal Vector: [ 0.95936444 -0.00899235 -0.2820266 ]
Offset: 8.724272669222737

-------------------------------

lidar_data/frame_795.pcd
PointCloud with 74 points.


Normal Vector: [ 0.662242   -0.74563698  0.07389876]
Offset: 7.749464153135403

-------------------------------

lidar_data/frame_1816.pcd
PointCloud with 61 points.


Normal Vector: [ 0.87718352 -0.41582556  0.24007951]
Offset: 7.4355467843521295

-------------------------------

lidar_data/frame_1430.pcd
PointCloud with 104 points.


Normal Vector: [ 0.93288206 -0.2440387   0.26490788]
Offset: 5.674258463241262

-------------------------------

lidar_data/frame_2771.pcd
PointCloud with 63 points.


Normal Vector: [-0.43460444  0.89601842 -0.09093933]
Offset: -6.595058223525193

-------------------------------

lidar_data/frame_1580.pcd
PointCloud with 150 points.


Normal Vector: [ 0.75932657 -0.56254883  0.32705042]
Offset: 4.954015451590527

-------------------------------
```

```
lidar_data/frame_1093.pcd
PointCloud with 116 points.


Normal Vector: [ 0.93809241 -0.22958973  0.25936688]
Offset: 5.204305546806319

------------------------------

lidar_data/frame_1850.pcd
PointCloud with 86 points.


Normal Vector: [-0.9065703   0.30866042  0.28785247]
Offset: -8.020993504372285

------------------------------

lidar_data/frame_2107.pcd
PointCloud with 42 points.


Normal Vector: [ 0.98745566  0.06712949 -0.14291586]
Offset: 10.309979614425066

------------------------------

lidar_data/frame_1061.pcd
PointCloud with 230 points.


Normal Vector: [ 0.63693437 -0.76499839  0.09535237]
Offset: 4.993811298096005

------------------------------

lidar_data/frame_2822.pcd
PointCloud with 101 points.


Normal Vector: [-0.61872738  0.78293781 -0.06469016]
Offset: -6.5895645932239955

------------------------------

lidar_data/frame_339.pcd
PointCloud with 39 points.


Normal Vector: [-0.89093965 -0.24536379  0.38212975]
```

```
Offset: -7.618700179547205

------------------------------

lidar_data/frame_3329.pcd
PointCloud with 358 points.


Normal Vector: [-0.73737765  0.5739658   0.35614247]
Offset: -3.971542150381967

------------------------------

lidar_data/frame_1153.pcd
PointCloud with 107 points.


Normal Vector: [ 0.83333533 -0.55208241  0.02751785]
Offset: 5.220803337324185

------------------------------

lidar_data/frame_595.pcd
PointCloud with 108 points.


Normal Vector: [ 0.93223066 -0.19842704 -0.30260984]
Offset: 6.26535811591381

------------------------------

lidar_data/frame_1366.pcd
PointCloud with 92 points.


Normal Vector: [ 0.81878735  0.50763718 -0.26812639]
Offset: 6.445439719171264

------------------------------

lidar_data/frame_812.pcd
PointCloud with 120 points.


Normal Vector: [ 0.73370584 -0.67940958  0.00885226]
Offset: 6.96918888440808

------------------------------

lidar_data/frame_2725.pcd
PointCloud with 41 points.
```

```
Normal Vector: [-0.9493562  -0.22833398  0.21583881]
Offset: -7.271752317637934

------------------------------

lidar_data/frame_256.pcd
PointCloud with 55 points.


Normal Vector: [-0.90149712 -0.39908052  0.16744454]
Offset: -8.140582366167767

------------------------------

lidar_data/frame_548.pcd
PointCloud with 50 points.


Normal Vector: [-0.825289    0.10756967  0.55437067]
Offset: -5.978131037863134

------------------------------

lidar_data/frame_607.pcd
PointCloud with 82 points.


Normal Vector: [ 0.98936009 -0.13850217 -0.04453952]
Offset: 7.325197139472717

------------------------------

lidar_data/frame_1558.pcd
PointCloud with 167 points.


Normal Vector: [ 0.60407281 -0.77412903  0.18926246]
Offset: 5.0623522119865365

------------------------------

lidar_data/frame_1798.pcd
PointCloud with 60 points.


Normal Vector: [-0.96334688  0.25648046  0.07861659]
Offset: -8.213132600834639

------------------------------
```

```
lidar_data/frame_2991.pcd
PointCloud with 166 points.


Normal Vector: [ 0.60371533 -0.79239849  0.08736381]
Offset: 3.7038756210944097

------------------------------

lidar_data/frame_1638.pcd
PointCloud with 173 points.


Normal Vector: [-0.47346333  0.7599202   0.44536924]
Offset: -5.254984226428699

------------------------------

lidar_data/frame_1725.pcd
PointCloud with 113 points.


Normal Vector: [-0.40409023  0.88184713 -0.24301591]
Offset: -5.43672051219062

------------------------------

lidar_data/frame_2962.pcd
PointCloud with 215 points.


Normal Vector: [-0.97607976  0.21597961 -0.02492227]
Offset: -4.772067004354305

------------------------------

lidar_data/frame_690.pcd
PointCloud with 50 points.


Normal Vector: [ 0.99427448 -0.1042191   0.02359328]
Offset: 8.846168921169088

------------------------------

lidar_data/frame_1698.pcd
PointCloud with 109 points.


Normal Vector: [-0.52528321  0.84450165  0.1043768 ]
```

```
Offset: -6.103645571922049

------------------------------

lidar_data/frame_2800.pcd
PointCloud with 86 points.


Normal Vector: [-0.76477664  0.64422625  0.00944619]
Offset: -6.98783594738935

------------------------------

lidar_data/frame_457.pcd
PointCloud with 117 points.


Normal Vector: [ 0.99121488 -0.12226682 -0.05043704]
Offset: 6.093678679246951

------------------------------

lidar_data/frame_639.pcd
PointCloud with 34 points.


Normal Vector: [ 0.96691202 -0.15794554 -0.2003356 ]
Offset: 8.493221054907726

------------------------------

lidar_data/frame_1139.pcd
PointCloud with 143 points.


Normal Vector: [ 0.72352642 -0.68812782 -0.05467752]
Offset: 4.79350162621247

------------------------------

lidar_data/frame_1473.pcd
PointCloud with 110 points.


Normal Vector: [-0.95207118  0.1539658  -0.26430097]
Offset: -5.743694294895789

------------------------------
```

Q2) Now that you have the plane normals in camera and LIDAR frame of reference respectively for all the selected images, derive the set of

equations that you would use to solve for estimating the transformation from the LIDAR frame to the camera frame. Explain how the rotation and translation matrices are calculated. [Hint: You may refer to this thesis (Sec. 5) for deriving the necessary equations.]

We have the given equations in the provided document,
Camera coordinate system
$\theta c = [\theta c,1\ \theta c,2\ .\ .\ .\ \theta c,n]T$, $\theta l = [\theta l,1\ \theta l,2\ .\ .\ .\ \theta l,n]T$,
Laser coordinate system
$\alpha c = [\alpha c,1\ \alpha c,2\ .\ .\ .\ \alpha c,n]T$, $\alpha l = [\alpha l,1\ \alpha l,2\ .\ .\ .\ \alpha l,n]T$

The rotation between the reference frames that minimize the difference between the normal from the origin to the corresponding planes in the two frames is given by R = U.(V.Transpose)

$\theta l.(\theta c(Transpose)) = U.(V.Transpose).S$

Next we have the Euler-Rodrigues Equations:
N = [[0, -S3, S2], [S3, 0, -S1], [-S2, S1, 0]] (N is a matrix)
$R = I + \sin(\theta) * N + (1 - \cos(\theta)) * N * N$

With the help ofn these equations we can derive the set of equations that we would use to solve for estimating the transformation from the LIDAR frame to the camera frame.