```python
import numpy as np
import numpy as np
import cv2
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow
import glob

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

Mounted at /content/drive/
```

Problem 1

Step 1: Image Collection

Link to the images : https://drive.google.com/drive/folders/11WEx8GBXNJJ5fdG2qF4GiZqhg97H4oSf?usp=sharing

```python
# Folder path containing the images
dir_path =
'/content/drive/MyDrive/ENPM673/projects_assets/Project3/*.jpg'
```

Step 2: Calibration

Finding the corners of all images in **World Frame** and **Image Frame**

Steps:

- Corners can be calculated through Harris-Corner Detection
- First, the gradient of the image is calculated along x and y direction which will represent the change in intensity.
- Then a Harris Response Function is calculated which represents the likelihood of a pixel being the corner
- After calculating the Response Function, the Corner Detector then identifies corners above a certain threshold
- In order to avoid the corners nearby and filter out the strongest corner in a particular region, Non-Maximal Suppression is used.
- In order to further improve the accuracy, sub-pixel functions can be used.
- In OpenCV, `findChessboardCorners` does the work of finding the corners using a grayscale image

- In order to use the function, we need to specify the number of squared rows and columns in a chessboard and these number of corners are found.

NOTE - As the images are taken with manual focus, it has higher dimensions and thus to find the corners in all 55 images, it will take about 3 Mins.

```
# Number of squares in x and y direction in a checkerboard
sq_dim = (6,8)

# The image size is 4000 x 2252 pixels
frame_size = (4000, 2252)

# Points in 3D World coordinates, w hich can be determined by the
number of corners of a chessboard
world_points = np.zeros(((sq_dim[0]*sq_dim[1]), 3), np.float32)
world_points[:,:2] = np.mgrid[0:6, 0:8].T.reshape(-1,2)

# Initializing the list of corners in image frame and world frame
img_lst = []
world_lst = []

width_small = int(frame_size[0]/4)
height_small = int(frame_size[1]/4)

# Iterating over all 55 images and converting to gray scale and
finding chessboard corners
for imgs in glob.glob(dir_path):

  img = cv2.imread(imgs)
  gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
  ret, corners = cv2.findChessboardCorners(gray, (6,8), None)

  # If the corners are found then the corresponding image and world
points are added to the list
  if ret == True:
    world_lst.append(world_points)
    img_lst.append(corners)
```

Procedure for camera calibration

- First of all, the world points are converted into the image sensor pixel coordinate system using perspective projection equations

- Using the intrinsic parameters like focal length in x and y directions we can create an intrinsic matrix

- Now, estimating the extrinsic matrix which will map the world frame to camera frame using rotation and translation and this will generate an extrinsic matrix

- Thus, we have intrinsic matrix which maps the camera frame to pixel frame and extrinsic matrix which maps the world frame to camera frame

- Combining the two equations, we get the projection matrix or camera matrix ũ = M(int)* X̃c and X̃c = M(ext) * X̃w
- This is dont by the below funcion `cv2.calibrateCamera`

```
# Finding calibration/projection matrix, distortion coefficient,
rotation matrix and translation matrix for the calibration

ret, cmatrix, dist_coeff, rot_matrix, trans_matrix =
cv2.calibrateCamera(world_lst, img_lst,frame_size, None, None)

# print("Distortion Coeff", dist_coeff)
print("Calibration Matrix", cmatrix)
# print("Translation Matrix",trans_matrix )
# print("Rotation Matrix",rot_matrix)

Calibration Matrix [[2.76852099e+03 0.00000000e+00 1.45962753e+03]
 [0.00000000e+00 2.76607947e+03 1.98942219e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]

# Finding the intrinsic and extrinsic matrix of the camera using QR
decomposition
T,A = np.linalg.qr(cmatrix)

print("Intrinsic Matrix \n",T )
print("Extrinsic Matrix \n",A)

Intrinsic Matrix
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
Extrinsic Matrix
 [[2.76852099e+03 0.00000000e+00 1.45962753e+03]
 [0.00000000e+00 2.76607947e+03 1.98942219e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]

# In order to get the optimal camera matrix we can use this function
optimal_cmatrix , roi = cv2.getOptimalNewCameraMatrix(cmatrix,
dist_coeff, frame_size, 1, frame_size)

print(optimal_cmatrix)

[[2.81837585e+03 0.00000000e+00 1.47506870e+03]
 [0.00000000e+00 2.79116754e+03 1.98723680e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
```

Step 3 - Reprojection Error Analysis

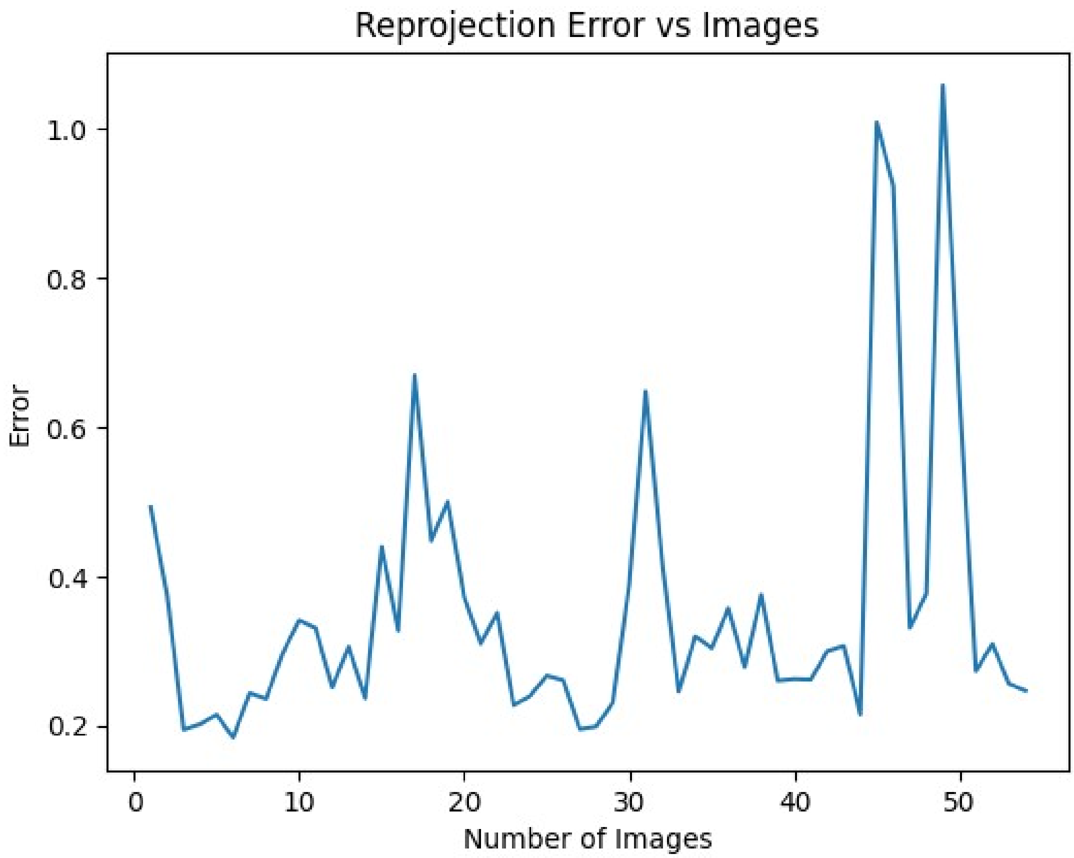
Significance & Implications

It is basically used to measure the calibration accuracy of the process.

If the error is close to 0, indicates the calibration is succesfull and indicates that an accurate mapping of 3D world coordinate to 2D image coordinates is done

It can be used in SLAM and sFM in order to accurateley predict the localization errors and improve the 3D reconstruction

```
err_list = []

for i in range(len(world_lst)):
  # using rotation matrix and translation matrix, every point in world
frame is reprojected into  image plane using this function
  imagepoints_new, ret = cv2.projectPoints(world_lst[i],
rot_matrix[i], trans_matrix[i], cmatrix, dist_coeff)

  # Normalizing the error and storing it into the list to plot this
  err = cv2.norm(img_lst[i], imagepoints_new,
cv2.NORM_L2)/len(imagepoints_new)
  err_list.append(err)

# As the number of images are from 1 to 55, creating the x-axis
variable
x = np.linspace(1,54,54, dtype=np.int8)
x = list(x)

# Plotting the error
plt.xlabel('Number of Images')
plt.ylabel('Error')
plt.title('Reprojection Error vs Images')
plt.plot(x,err_list)
plt.show()
```



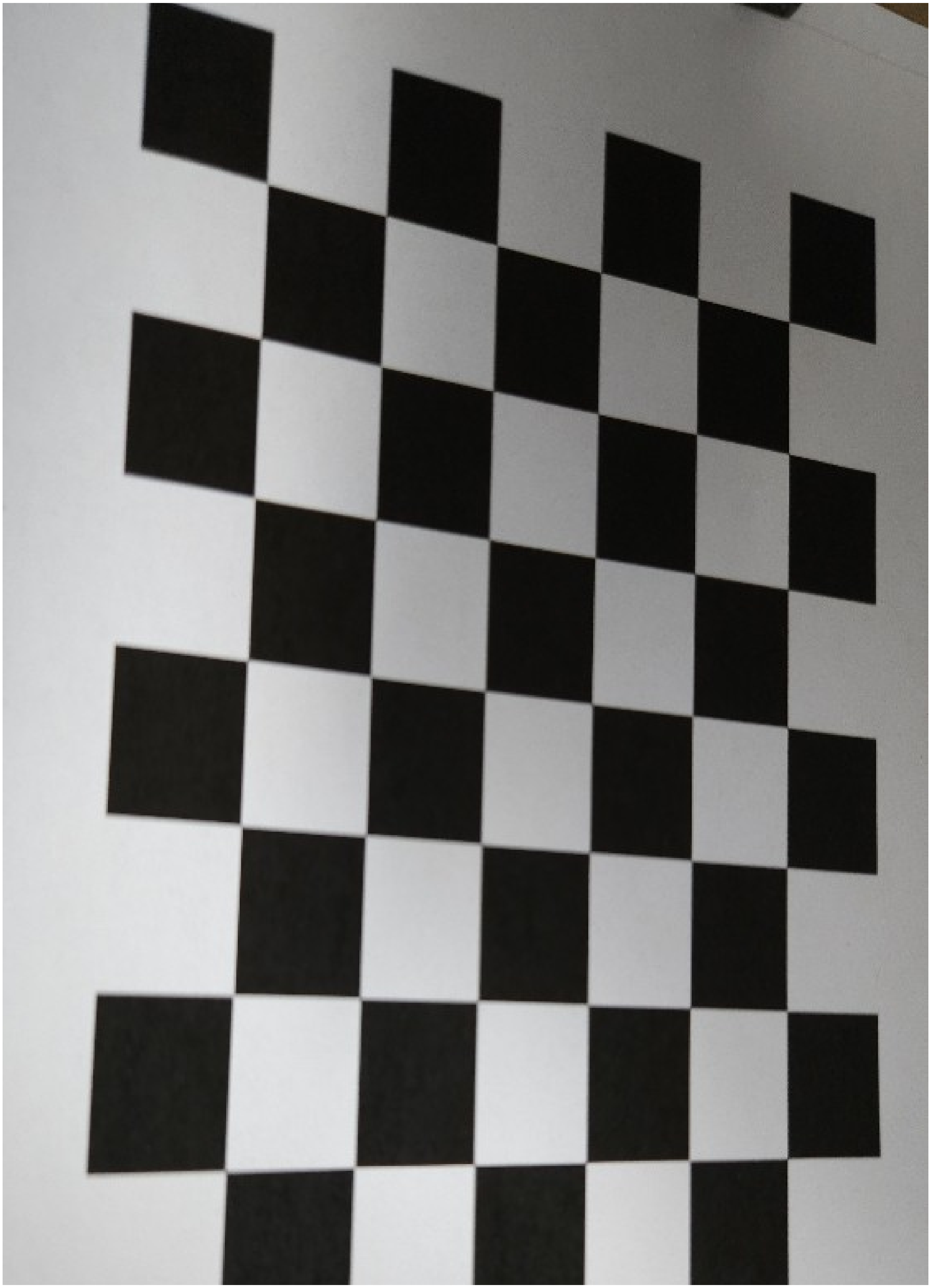
Step 4 - Visulization of Images

Although the distortion is not clearly visible, but you can check if the algorithm works by seeing the top left corner of the second image which indicates that the undistortion of the image

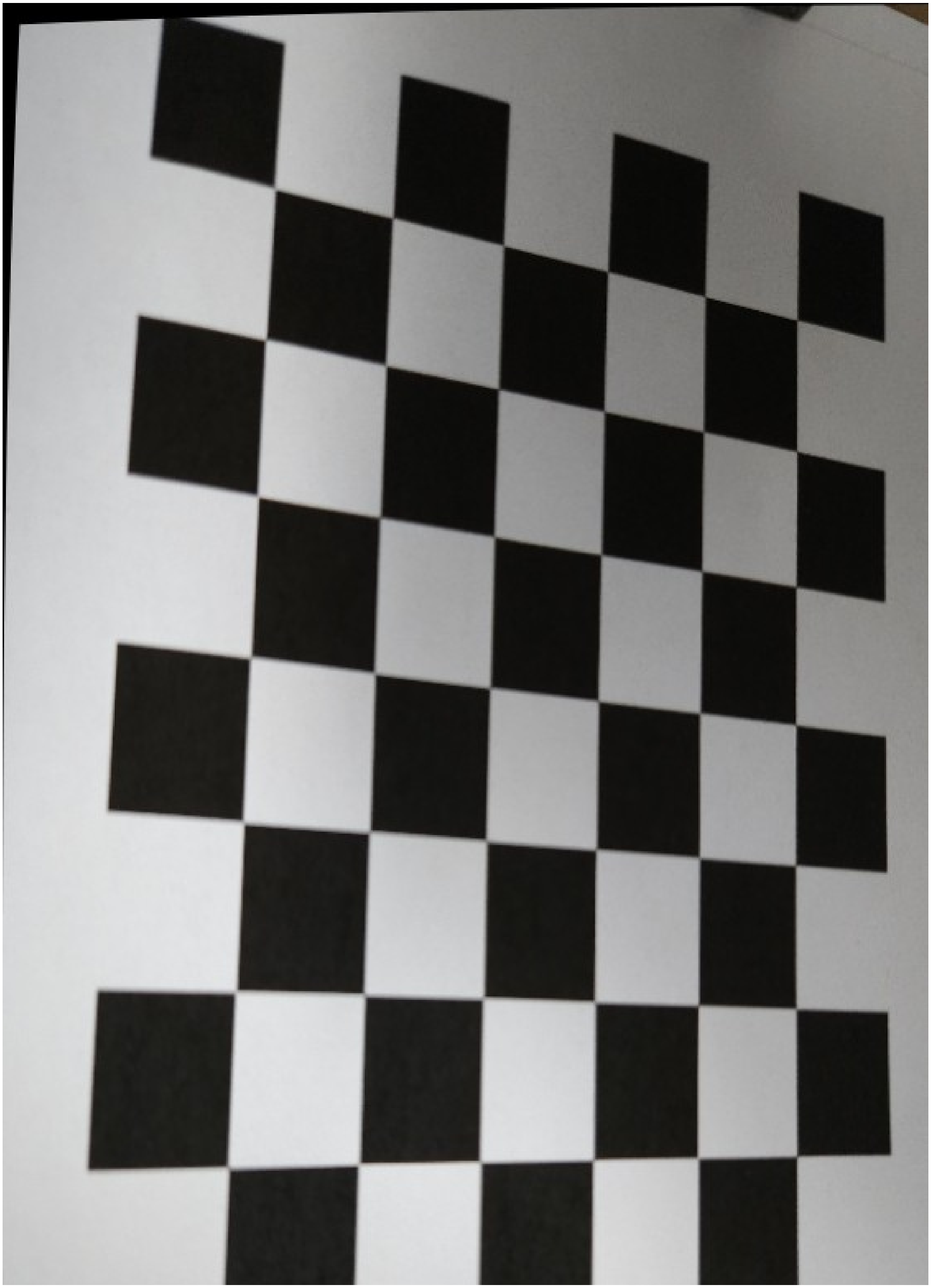
```
new_img =
cv2.imread('/content/drive/MyDrive/ENPM673/projects_assets/Project3/20
240412_143739.jpg')

# Reducing the size of the image
new_img_sm = cv2.resize(new_img, (int(frame_size[1]/4),
int(frame_size[0]/4)))

# Saving the uncalibrated original image
cv2.imwrite('original.jpg',new_img_sm)

# Applying the undistortion using distortion coefficient, camera
matrix and optimal camera matrix
undistort_img = cv2.undistort(new_img_sm, cmatrix, dist_coeff, None,
optimal_cmatrix)
# x,y,w,h = roi
# undistort_img[y:y+h, x:x+w]
cv2.imwrite('Undistorted Image.jpg', undistort_img)
```

```
print("Original Image")
cv2_imshow(new_img_sm)

print("Undistorted Image")
cv2_imshow(undistort_img)
```

```
Original Image
```

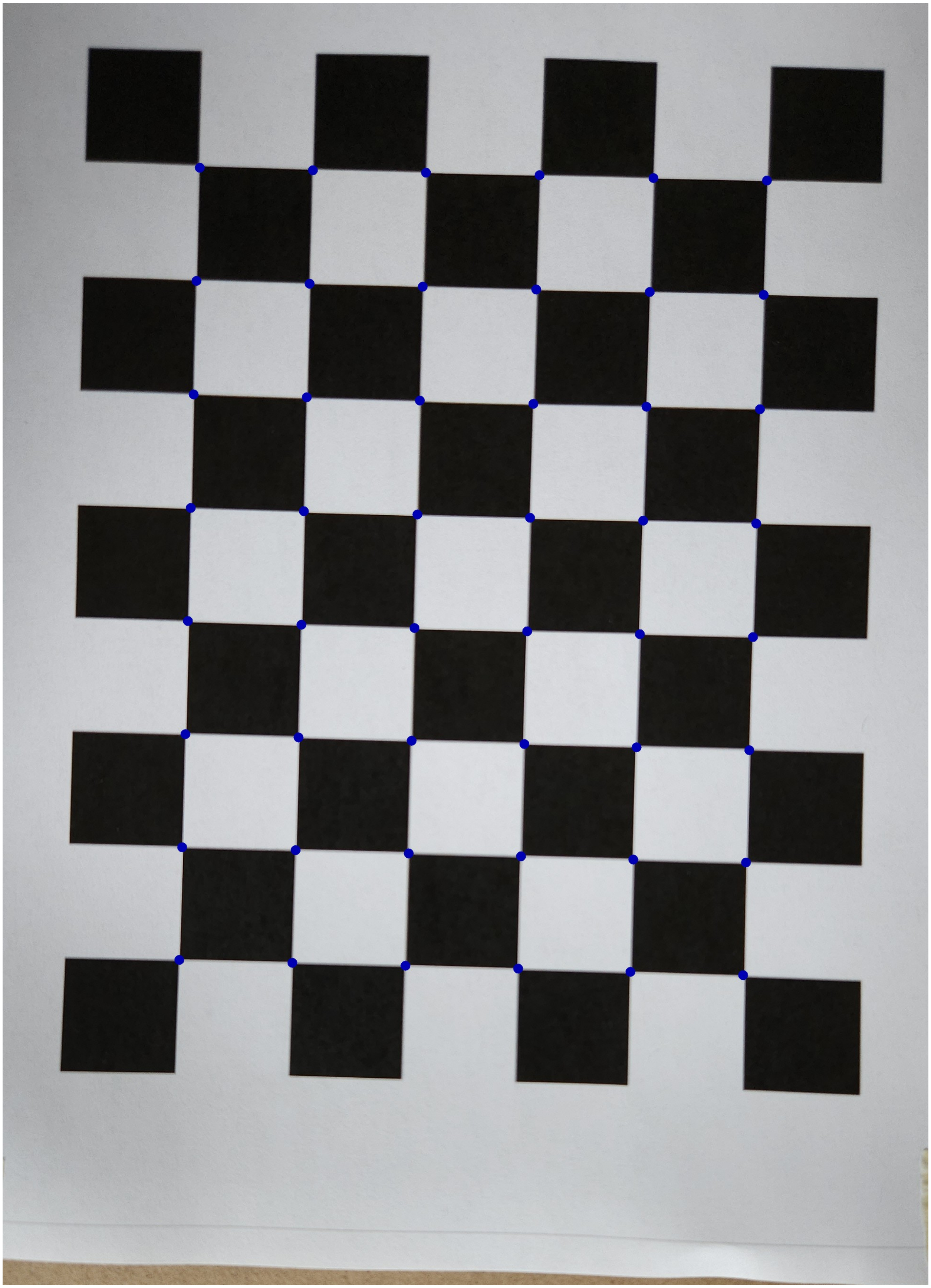


```
Undistorted Image
```

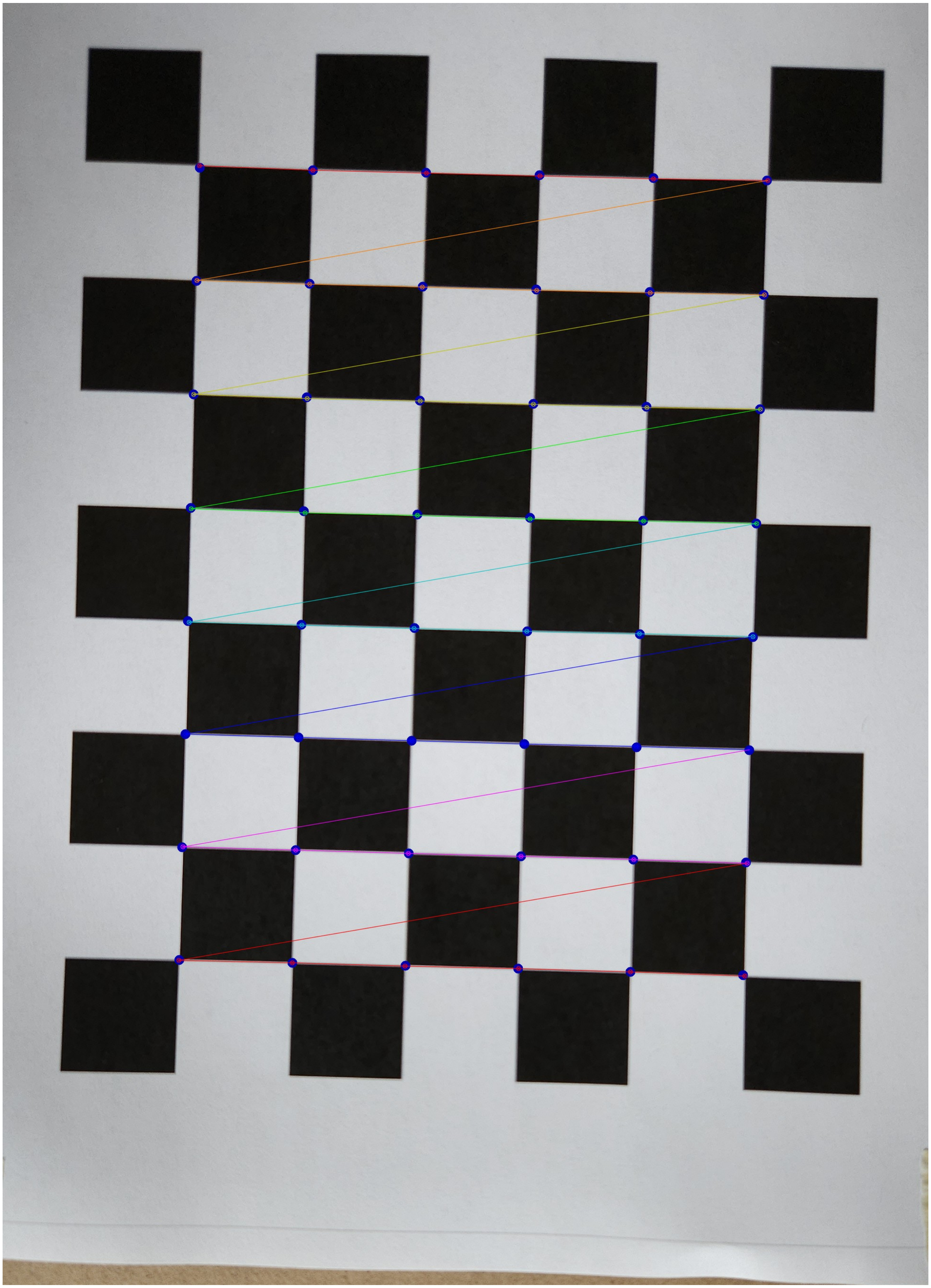


```
# Taking the last image and drawing the corners and reprojected
corners
last =
cv2.imread('/content/drive/MyDrive/ENPM673/projects_assets/Project3/20
240412_143605.jpg')

# Drawing the reprojected corners in blue color
for i in range(len(imagepoints_new)):
  cv2.circle(last, (int(imagepoints_new[i][0][0]),
int(imagepoints_new[i][0][1])), 12, (180,0,0), -1)

print("Reprojected corners on the image")
cv2_imshow(last)

# Drawing the detected corners in different colros and as you can see
that there is minor error between
# the two images which is evident from the reprojection error
cv2.drawChessboardCorners(last, (6,8), corners, True)
print("Detected Corners on Original Image")
cv2_imshow(last)
```

```
Reprojected corners on the image
```



```
Detected Corners on Original Image
```



Problem 2 (Classroom Images)

Stereo Calbration

```
# Reading the images
class_img1 =
cv2.imread('/content/drive/MyDrive/ENPM673/projects_assets/Project3/
classroom/im0.png')
class_img2 =
cv2.imread('/content/drive/MyDrive/ENPM673/projects_assets/Project3/
classroom/im1.png')

'''
cam0=[1746.24 0 14.88; 0 1746.24 534.11; 0 0 1]
cam1=[1746.24 0 14.88; 0 1746.24 534.11; 0 0 1]
doffs=0
baseline=678.37
width=1920
height=1080
ndisp=310
vmin=60
vmax=280

'''
```

{"type":"string"}

```
# Using SIFT Feature Detector between two images and initializing the
detector
sift = cv2.SIFT_create()

# Finding the keypoints and its neighbourhood descriptor in both
images to help the feature detection.
keypoints1, descriptor1 = sift.detectAndCompute(class_img1, None)
keypoints2, descriptor2 = sift.detectAndCompute(class_img2, None)

# Number of matches used in FLANN
num_matches = 60

# For this I have used LINEAR FLANN INDEX which scans through the
features and finds the nearest neighbours
idx_params = dict(algorithm=0, trees=5)
# Searching is aided with the help of number of checks the algorithm
does and more the searches more the accuracy but computationally
expensive
srch_params = dict(checks=80)

flann_matcher = cv2.FlannBasedMatcher(idx_params, srch_params)

# Performs matching the descriptors of the two images
```

```python
matches = flann_matcher.match(descriptor1,descriptor2)
# Sorting based on the distance between the matches
matches = sorted(matches, key=lambda x: x.distance)

# Contains the coordinates of points from the source image which is to
be warped and for each match in matches_flann, it retrieves the
keypoint coordinates
points2 = np.float32([keypoints1[m.queryIdx].pt for m in
matches[:200]]).reshape(-1, 1, 2)
# Contains the coordinates of points from the Initial image and for
each match in matches_flann, it retrieves the keypoint coordinates
points1 = np.float32([keypoints2[m.trainIdx].pt for m in
matches[:200]]).reshape(-1, 1, 2)

# Finding fundamental matrix from the best matches using RANSAC
points1 = np.int32(np.array(points1))
points2 = np.int32(np.array(points2))

# The function returns the binary mask and fundamental matrix
indicating if the point is outlier or inlier
fmatrix, mask = cv2.findFundamentalMat(points1, points2,
cv2.FM_RANSAC)

print("Fundamental Matrix\n ", fmatrix)
```

```
Fundamental Matrix
  [[ 3.43998700e-08  2.38943206e-05 -1.28548255e-02]
 [-2.32378023e-05 -4.58790446e-06 -7.04058047e-02]
 [ 1.26294486e-02  7.11837284e-02  1.00000000e+00]]
```

```python
# Selecting only inlier points by falttening the mask array and
selecting "ONES"
points1 = points1[mask.ravel() == 1]
points2 = points2[mask.ravel() == 1]

# Intrinsic Parameter for Left Camera
k1 = np.array([[1746.24, 0, 14.88],
                [0, 1746.24, 534.11],
                [0, 0, 1]])

# Intrinsic Parameter for Right Camera
k2 = np.array([[1746.24, 0, 14.88],
                [0, 1746.24, 534.11],
                [0, 0, 1]])

baseline=678.37
f = 1746.24

# finding essential matrix from fundamental and intrinsic parameters
E = np.dot(k2.T, np.dot(fmatrix,k1))
```

```
print("Essential Matrix\n\n", E, "\n")

# Decomposing Essential Matrix into two possible rotation matrix and
translation matrix
R1,R2,T = cv2.decomposeEssentialMat(E)

print("Rotation Matrix \n\n", R1, "\n")
print("Translation Matrix \n\n", T)
```

```
Essential Matrix

 [[ 1.04897386e-01  7.28622455e+01 -1.60860150e-01]
 [-7.08602887e+01 -1.39901454e+01 -1.27828311e+02]
 [ 3.81410095e-01  1.20645679e+02  1.08561068e-01]]

Rotation Matrix

 [[ 0.99902771  0.02412456  0.03690051]
 [-0.02255479  0.9988469  -0.04238104]
 [-0.03788039  0.04150755  0.99841985]]

Translation Matrix

 [[ 0.85592419]
 [-0.00151626]
 [-0.5170991 ]]
```

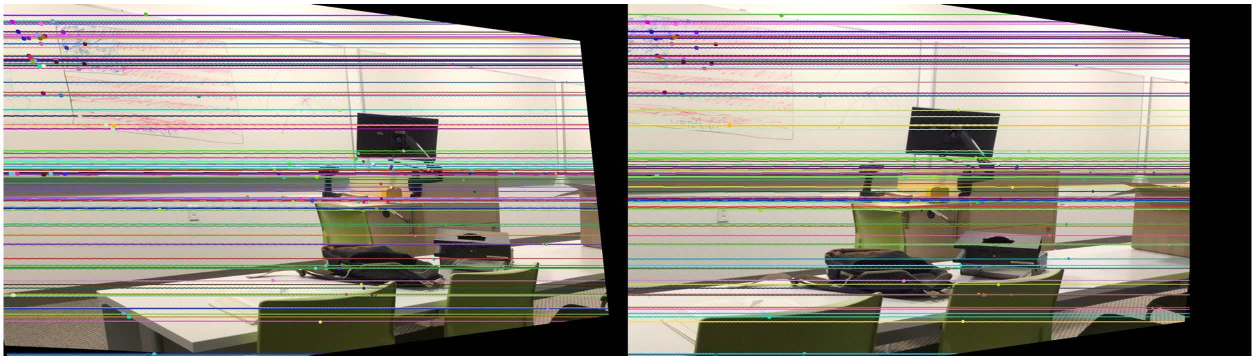
Stereo Rectification

```
# Making the draw lines function to draw the epipolar
linescorresponding to new points in first image on second image
# Img1 - Image on which the epilines for the points in img2 are drawn

def draw_epilines(input_img1, input_img2, lines, pts1, pts2):
  h,w,_ = input_img1.shape

  for l,point1,point2 in zip(lines,pts1,pts2):

    #Generating random color lines
    color = tuple(np.random.randint(0,255,3).tolist())

    # Cordinates for generating a line
    x0,y0 = map(int, [0, -l[2]/l[1]])
    x1,y1 = map(int, [w, -(l[2]+l[0]*w)/l[1]])

    # Drawing the line on image 1 from the corresponding points of
image 2
    img_result1 = cv2.line(input_img1, (x0,y0), (x1,y1), color,2)

    # Representing coresponding points in image 1
```

```
    img_result1 = cv2.circle(input_img1, tuple(point1[0]), 5, color, -
1)
    img_result2 = cv2.circle(input_img2, tuple(point2[0]), 5, color, -
1)

  return img_result1, img_result2

# Creating epipolar lines of points on image 2 as lines in image 1
lines1 = cv2.computeCorrespondEpilines(points2.reshape(-1,1,2), 2,
fmatrix)
lines1 = lines1.reshape(-1,3)

# Generating new image which draws epipolar lines
img1_lines, img1_points = draw_epilines(class_img1, class_img2,
lines1, points1, points2)

# Creating epipolar lines of points on image 1 as lines in image 2
lines2 = cv2.computeCorrespondEpilines(points1.reshape(-1, 1, 2), 1,
fmatrix)
lines2 = lines2.reshape(-1, 3)

# Generating new image which draws epipolar lines
img2_lines, img2_points = draw_epilines(class_img2, class_img1,
lines2, points2, points1)

h1, w1, z1 = class_img1.shape
h2, w2, z2 = class_img2.shape

# Using this function to generate the homography of both images to
make the horizontal epipolar lines
_, homography1, homography2 = cv2.stereoRectifyUncalibrated(points1,
points2, fmatrix, (w1,h1))

# Applying perspective projection to generate the rectified images
cimg1_rect = cv2.warpPerspective(class_img1, homography1, (w1,h1))
cimg2_rect = cv2.warpPerspective(class_img2, homography2, (w2,h2))

# Homopgrahy matrices
print("Homography of First Image \n", homography1)
print("Homography of Second Image \n", homography2)
```

```
Homography of First Image
 [[ 8.76920012e-02  1.37430532e-02 -1.28443099e+01]
 [ 9.65866950e-03  7.17964610e-02 -8.65145813e+00]
 [ 1.77253102e-05  3.50321368e-06  5.37020936e-02]]
Homography of Second Image
 [[ 1.25028872e+00  2.50868592e-04 -2.40412637e+02]
 [ 1.40586766e-01  1.00002823e+00 -1.34978539e+02]
 [ 2.60717434e-04  5.23125697e-08  7.49683015e-01]]
```

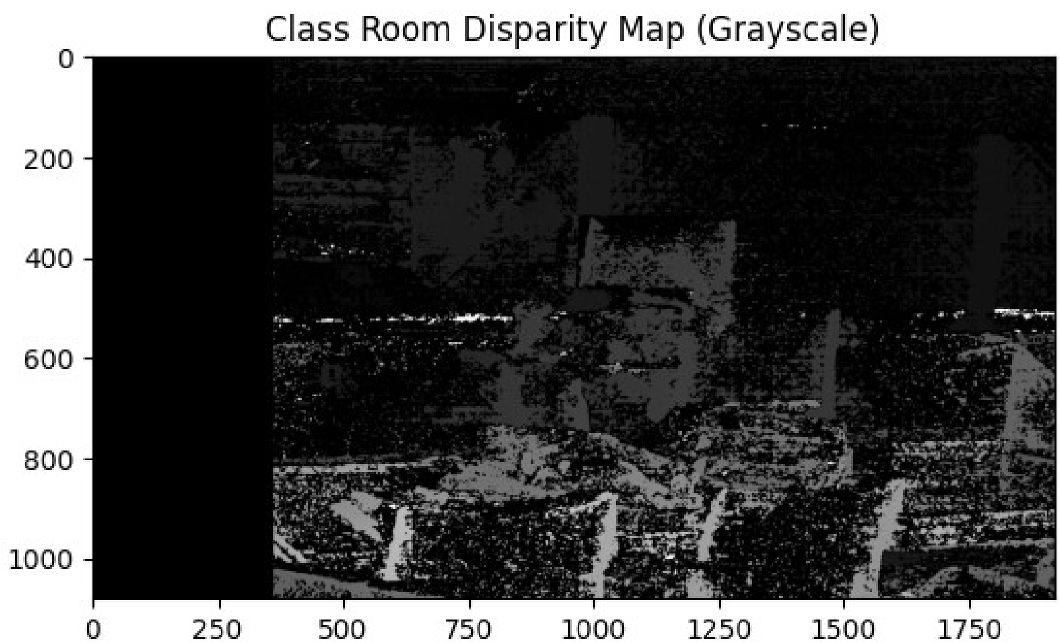
```
combined_img = cv2.hconcat([cimg1_rect, cimg2_rect])
cv2_imshow(combined_img)
```

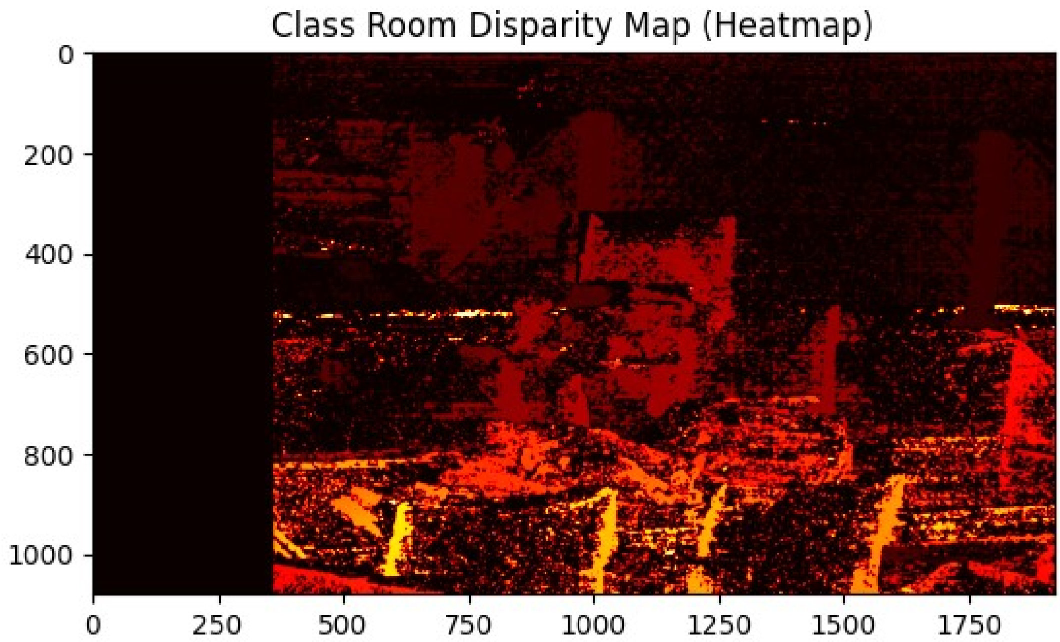


Computing Depth Map

```
def dispMap(input_img1, input_img2, no_disp, min_disp, max_disp,
block, uniq_ratio):

  stereo_new = cv2.StereoSGBM_create(

      # setting minimum possible disparity value
      minDisparity=min_disp,

      # Setting the number of disparity to be displayed
      numDisparities=no_disp,

      # Setting the window size and keeping it as odd number
      blockSize=block,

      # Checking the similarity index and unique disparities are
selected
      uniquenessRatio=uniq_ratio,

      #P1 & P2 are the disparity smoothness parameter and makes the
disparities more smoother
      P1=8 * 1 * block * block,
      P2=32 * 1 * block * block,
  )

  # Creating a disparity map
  disparity_img = stereo_new.compute(input_img1, input_img2)

  # Normalizing the values in pixels
  disparity_img = cv2.normalize(disparity_img, disparity_img,
alpha=255,
                                beta=0, norm_type=cv2.NORM_MINMAX)

  # COnverting into the supported uint8 format
  disparity_img = np.uint8(disparity_img)
```

```
    return disparity_img

# Reading the images
class_img1 =
cv2.imread('/content/drive/MyDrive/ENPM673/projects_assets/Project3/
classroom/im0.png', cv2.IMREAD_GRAYSCALE)
class_img2 =
cv2.imread('/content/drive/MyDrive/ENPM673/projects_assets/Project3/
classroom/im1.png', cv2.IMREAD_GRAYSCALE)

# class_disparity = dispMap(class_img1,class_img2,150, 50,280,11, 15)
class_disparity = dispMap(class_img1,class_img2,310, 50,320,5, 15)

plt.title('Class Room Disparity Map (Grayscale)')
plt.imshow(class_disparity, cmap='gray')
plt.show()

plt.title('Class Room Disparity Map (Heatmap)')
plt.imshow(class_disparity, cmap='hot')
plt.show()
```

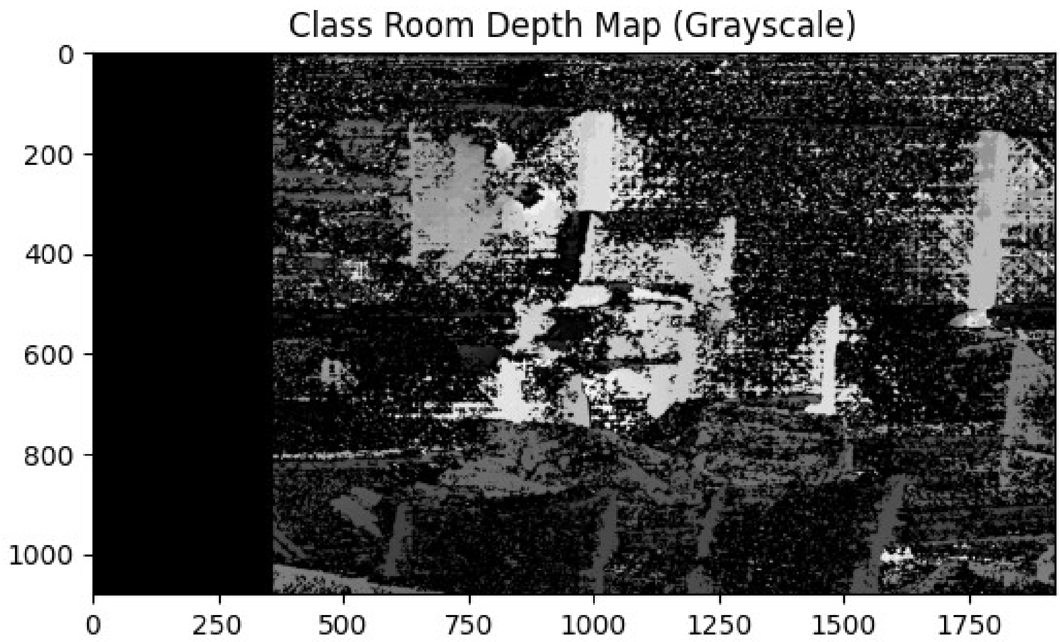


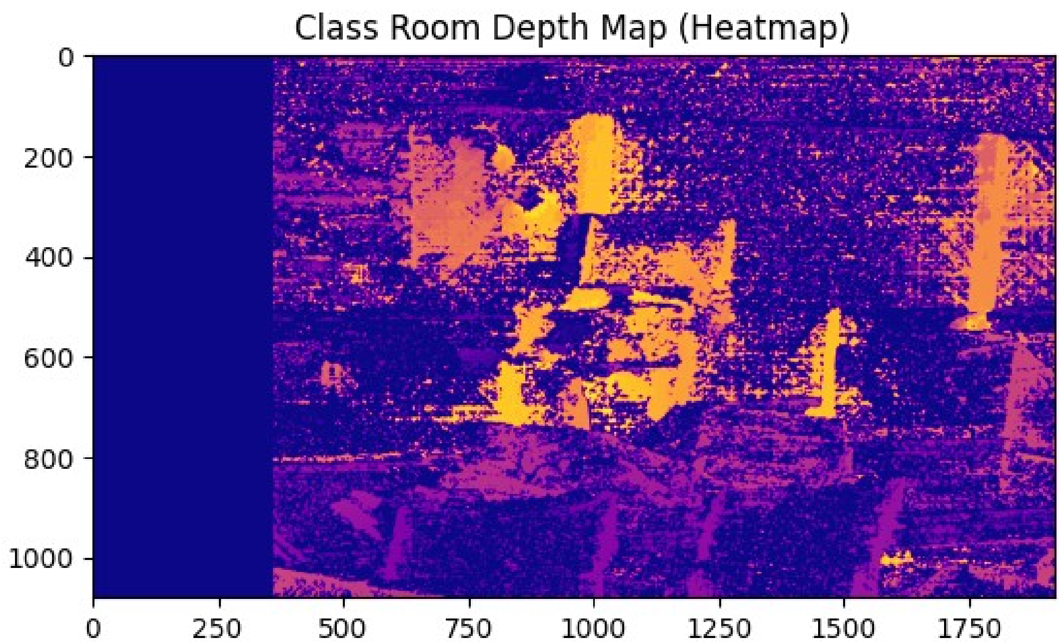



```
# Converting a disparity information to depth map using focal length
and baseline
def disp2depth(disp_img, flength, basel):
  h1,w1 = disp_img.shape
  depth_img = disp_img
  for i in range(w1):
    for j in range(h1):
      depth_img[j,i] = ((flength*basel)/disp_img[j,i])*0.01

  return depth_img


trap_depth = disp2depth(class_disparity, 1746.24, 678.37)
```

```
<ipython-input-100-7c3ceb9ba0f5>:7: RuntimeWarning: divide by zero
encountered in divide
  depth_img[j,i] = ((flength*basel)/disp_img[j,i])*0.01
<ipython-input-100-7c3ceb9ba0f5>:7: RuntimeWarning: invalid value
encountered in cast
  depth_img[j,i] = ((flength*basel)/disp_img[j,i])*0.01
```

```
plt.title('Class Room Depth Map (Grayscale)')
plt.imshow(trap_depth, cmap='gray')
plt.show()

plt.title('Class Room Depth Map (Heatmap)')
plt.imshow(trap_depth, cmap='plasma')
plt.show()
```





Problem 2 (Trap Room)

```
trap_img1 =
cv2.imread('/content/drive/MyDrive/ENPM673/projects_assets/Project3/
traproom/im0.png')
trap_img2 =
```

```
cv2.imread('/content/drive/MyDrive/ENPM673/projects_assets/Project3/
traproom/im1.png')
```

Calibration of Traproom Images

```
'''
cam0=[1769.02 0 1271.89; 0 1769.02 527.17; 0 0 1]
cam1=[1769.02 0 1271.89; 0 1769.02 527.17; 0 0 1]
doffs=0
baseline=295.44
width=1920
height=1080
ndisp=140
vmin=25
vmax=118
'''
```

{"type":"string"}

```
def findFundMat(input_img1, input_img2):
  # Using SIFT Feature Detector between two images and initializing
the detector
  sift = cv2.SIFT_create()

  # Finding the keypoints and its neighbourhood descriptor in both
images to help the feature detection.
  keypoints1, descriptor1 = sift.detectAndCompute(input_img1, None)
  keypoints2, descriptor2 = sift.detectAndCompute(input_img2, None)

  # Number of matches used in FLANN
  num_matches = 60

  # For this I have used LINEAR FLANN INDEX which scans through the
features and finds the nearest neighbours
  idx_params = dict(algorithm=0, trees=5)
  # Searching is aided with the help of number of checks the algorithm
does and more the searches more the accuracy but computationally
expensive
  srch_params = dict(checks=80)

  flann_matcher = cv2.FlannBasedMatcher(idx_params, srch_params)

  # Performs matching the descriptors of the two images
  matches = flann_matcher.match(descriptor1,descriptor2)
  # Sorting based on the distance between the matches
  matches = sorted(matches, key=lambda x: x.distance)

  # Contains the coordinates of points from the source image which is
to be warped and for each match in matches_flann, it retrieves the
keypoint coordinates
```

```
  points2 = np.float32([keypoints1[m.queryIdx].pt for m in
matches[:200]]).reshape(-1, 1, 2)
  # Contains the coordinates of points from the Initial image and for
each match in matches_flann, it retrieves the keypoint coordinates
  points1 = np.float32([keypoints2[m.trainIdx].pt for m in
matches[:200]]).reshape(-1, 1, 2)

  # Finding fundamental matrix from the best matches using RANSAC
  points1 = np.int32(np.array(points1))
  points2 = np.int32(np.array(points2))

  # The function returns the binary mask and fundamental matrix
indicating if the point is outlier or inlier
  fmatrix, mask = cv2.findFundamentalMat(points1, points2,
cv2.FM_RANSAC)

  # Selecting only inlier points by falttening the mask array and
selecting "ONES"
  points1 = points1[mask.ravel() == 1]
  points2 = points2[mask.ravel() == 1]

  return fmatrix, points1, points2

fmatrix_trap, points1_trap, points2_trap = findFundMat(trap_img1,
trap_img2)

print("Fundamental Matrix\n ", fmatrix_trap)

Fundamental Matrix
  [[-2.08907069e-08  2.83998452e-05 -1.08049066e-02]
 [-2.88332434e-05  7.12085051e-07 -1.75718680e-01]
 [ 1.08464731e-02  1.72884099e-01  1.00000000e+00]]

# Intrinsic Parameter for Left Camera
k1_trap = np.array([[1769.02, 0, 1271.89],
                [0, 1769.02, 527.17],
                [0, 0, 1]])

# Intrinsic Parameter for Right Camera
k2_trap = np.array([[1769.02, 0, 1271.89],
                [0, 1769.02, 527.17],
                [0, 0, 1]])

baseline_trap = 295.44
f_trap = 1769.02

# finding essential matrix from fundamental and intrinsic parameters
E_trap = np.dot(k2_trap.T, np.dot(fmatrix_trap,k1_trap))

print("Essential Matrix\n\n", E_trap, "\n")
```

```
# Decomposing Essential Matrix into two possible rotation matrix and translation matrix
R1_trap,R2_trap,T_trap = cv2.decomposeEssentialMat(E_trap)

print("Rotation Matrix \n\n", R1_trap, "\n")
print("Translation Matrix \n\n", T_trap)
```

```
Essential Matrix

 [[-6.53760415e-02  8.88753775e+01  7.32386505e+00]
 [-9.02316675e+01  2.22842157e+00 -3.75060551e+02]
 [-7.74851714e+00  3.70399120e+02 -5.67933219e-01]]

Rotation Matrix

 [[ 9.99999906e-01  1.38336819e-04 -4.11459398e-04]
 [-1.37445861e-04  9.99997648e-01  2.16460026e-03]
 [ 4.11757874e-04 -2.16454351e-03  9.99997573e-01]]

Translation Matrix

 [[ 0.97219123]
 [ 0.01933754]
 [-0.23338866]]
```

Rectification of Traproom Images

```
def draw_epilines(input_img1, input_img2, lines, pts1, pts2):
  h,w,_ = input_img1.shape

  for l,point1,point2 in zip(lines,pts1,pts2):

    #Generating random color lines
    color = tuple(np.random.randint(0,255,3).tolist())

    # Cordinates for generating a line
    x0,y0 = map(int, [0, -l[2]/l[1]])
    x1,y1 = map(int, [w, -(l[2]+l[0]*w)/l[1]])

    # Drawing the line on image 1 from the corresponding points of image 2
    img_result1 = cv2.line(input_img1, (x0,y0), (x1,y1), color,2)

    # Representing coresponding points in image 1
    img_result1 = cv2.circle(input_img1, tuple(point1[0]), 5, color, -1)
    img_result2 = cv2.circle(input_img2, tuple(point2[0]), 5, color, -1)

  return img_result1, img_result2
```

```
def computeEpilines(input_img1,input_img2, input_points1,
input_points2, fund_mat):
# Creating epipolar lines of points on image 2 as lines in image 1
  lines1 = cv2.computeCorrespondEpilines(input_points2.reshape(-
1,1,2), 2, fund_mat)
  lines1 = lines1.reshape(-1,3)

  # Generating new image which draws epipolar lines
  img1_lines, img1_points = draw_epilines(input_img1, input_img2,
lines1, input_points1, input_points2)

  # Creating epipolar lines of points on image 1 as lines in image 2
  lines2 = cv2.computeCorrespondEpilines(input_points1.reshape(-1, 1,
2), 1, fund_mat)
  lines2 = lines2.reshape(-1, 3)

  # Generating new image which draws epipolar lines
  img2_lines, img2_lines = draw_epilines(input_img2, input_img1,
lines2, input_points2, input_points1)

  return img1_lines, img2_lines


def rectifyImage(input_img1, input_img2, input_points1, input_points2,
fund_mat):
  h1, w1, z1 = input_img1.shape
  h2, w2, z2 = input_img2.shape

  # Using this function to generate the homography of both images to
make the horizontal epipolar lines
  _, homography1, homography2 =
cv2.stereoRectifyUncalibrated(input_points1, input_points2, fund_mat,
(w1,h1))

  # Applying perspective projection to generate the rectified images
  img_rect1 = cv2.warpPerspective(input_img1, homography1, (w1,h1))
  img_rect2 = cv2.warpPerspective(input_img2, homography2, (w2,h2))

  combined_img = cv2.hconcat([img_rect1, img_rect2])


  return combined_img,homography1,homography2


trap1_lines, trap2_lines = computeEpilines(trap_img1, trap_img2,
points1_trap, points2_trap, fmatrix_trap)

rectified_combined_img, trap_h1, trap_h2 = rectifyImage(trap_img1,
trap_img2, points1_trap, points2_trap, fmatrix_trap)
```

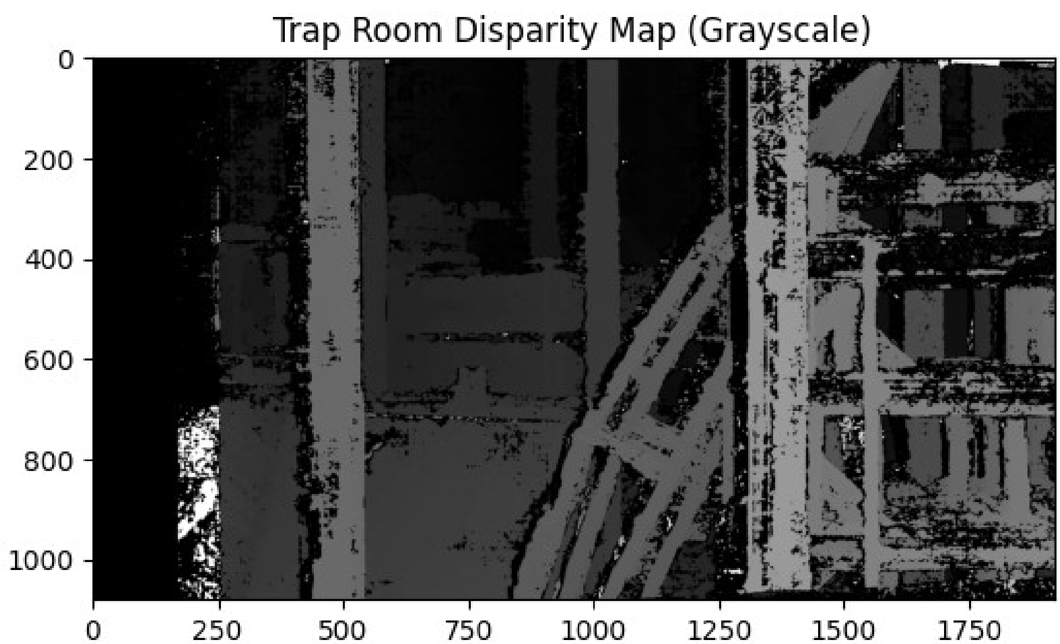
```
print("TrapRoom Image Left Homography \n",trap_h1 )
print("TrapRoom Image Right Homography \n",trap_h2 )

cv2_imshow(rectified_combined_img)
```

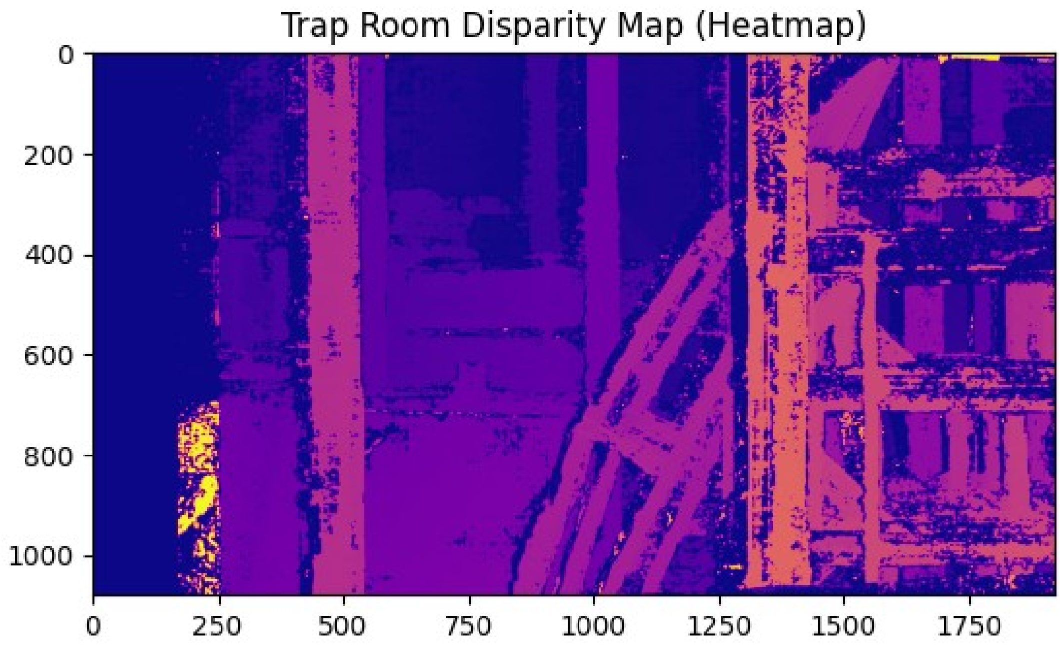
```
TrapRoom Image Left Homography
 [[ 2.04255774e-01 -1.46923067e-03 -1.54544988e+01]
 [ 9.36581390e-03  1.73513162e-01 -8.24636101e+00]
 [ 2.49462852e-05 -6.10794614e-08  1.51821684e-01]]
TrapRoom Image Right Homography
 [[ 1.13570991e+00  2.56531780e-02 -1.44134230e+02]
 [ 5.38982348e-02  1.00147251e+00 -5.25374628e+01]
 [ 1.41630122e-04  3.19911157e-06  8.62307563e-01]]
```

Depth Map of Traproom Images

```
def dispMap(input_img1, input_img2, no_disp, min_disp, max_disp,
block, uniq_ratio):

  stereo_new = cv2.StereoSGBM_create(
      minDisparity=min_disp,
      numDisparities=no_disp,
      blockSize=block,
      uniquenessRatio=uniq_ratio,
      P1=8 * 1 * block * block,
      P2=32 * 1 * block * block,
  )

  disparity_img = stereo_new.compute(input_img1, input_img2)

  disparity_img = cv2.normalize(disparity_img, disparity_img,
alpha=255,
                                beta=0, norm_type=cv2.NORM_MINMAX)

  disparity_img = np.uint8(disparity_img)

  return disparity_img

# Reading the images
```

```
trap_img1 =
cv2.imread('/content/drive/MyDrive/ENPM673/projects_assets/Project3/
traproom/im0.png', cv2.IMREAD_GRAYSCALE)
trap_img2 =
cv2.imread('/content/drive/MyDrive/ENPM673/projects_assets/Project3/
traproom/im1.png', cv2.IMREAD_GRAYSCALE)

trap_disparity = dispMap(trap_img1,trap_img2,140, 30,120,5, 20)

plt.title('Trap Room Disparity Map (Grayscale)')
plt.imshow(trap_disparity, cmap='gray')
plt.show()

plt.title('Trap Room Disparity Map (Heatmap)')
plt.imshow(trap_disparity, cmap='plasma')
plt.show()
```

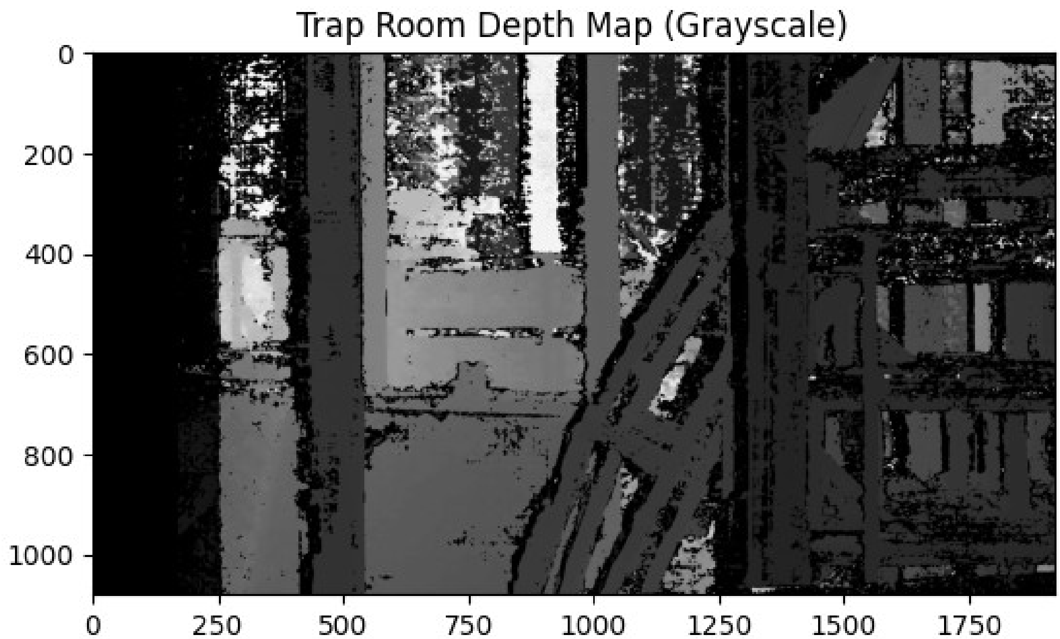


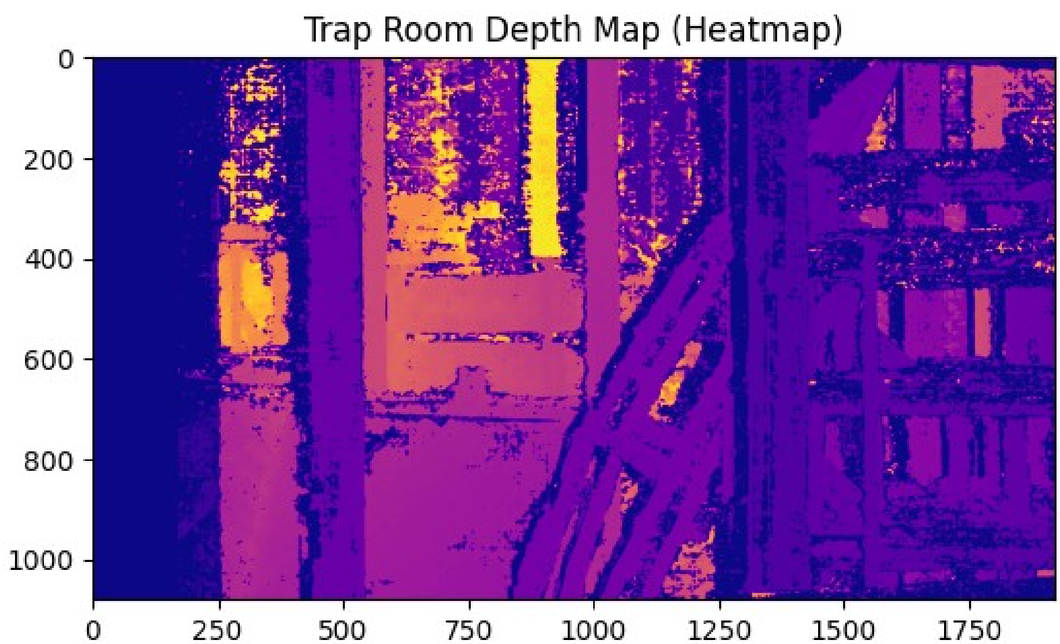



```
def disp2depth(disp_img, flength, basel):
  h1,w1 = disp_img.shape
  depth_img = disp_img
  for i in range(w1):
    for j in range(h1):
      depth_img[j,i] = ((flength*basel)/disp_img[j,i])*0.01

  return depth_img


trap_depth = disp2depth(trap_disparity, 1769.02, 295.44)
```

```
<ipython-input-133-9e22a5966108>:6: RuntimeWarning: divide by zero
encountered in divide
  depth_img[j,i] = ((flength*basel)/disp_img[j,i])*0.01
<ipython-input-133-9e22a5966108>:6: RuntimeWarning: invalid value
encountered in cast
  depth_img[j,i] = ((flength*basel)/disp_img[j,i])*0.01
```

```
plt.title('Trap Room Depth Map (Grayscale)')
plt.imshow(trap_depth, cmap='gray')
plt.show()

plt.title('Trap Room Depth Map (Heatmap)')
plt.imshow(trap_depth, cmap='plasma')
plt.show()
```





Problem 2 (StorageRoom)

```
storage_img1 =
cv2.imread('/content/drive/MyDrive/ENPM673/projects_assets/Project3/
storage room/im0.png')
storage_img2 =
```

```
cv2.imread('/content/drive/MyDrive/ENPM673/projects_assets/Project3/
storage room/im1.png')
```

Calibration of Traproom Images

```
'''
cam0=[1742.11 0 804.90; 0 1742.11 541.22; 0 0 1]
cam1=[1742.11 0 804.90; 0 1742.11 541.22; 0 0 1]
doffs=0
baseline=221.76
width=1920
height=1080
ndisp=100
vmin=29
vmax=61
'''
```

{"type":"string"}

```
def findFundMat(input_img1, input_img2):
  # Using SIFT Feature Detector between two images and initializing
the detector
  sift = cv2.SIFT_create()

  # Finding the keypoints and its neighbourhood descriptor in both
images to help the feature detection.
  keypoints1, descriptor1 = sift.detectAndCompute(input_img1, None)
  keypoints2, descriptor2 = sift.detectAndCompute(input_img2, None)

  # Number of matches used in FLANN
  num_matches = 60

  # For this I have used LINEAR FLANN INDEX which scans through the
features and finds the nearest neighbours
  idx_params = dict(algorithm=0, trees=5)
  # Searching is aided with the help of number of checks the algorithm
does and more the searches more the accuracy but computationally
expensive
  srch_params = dict(checks=80)

  flann_matcher = cv2.FlannBasedMatcher(idx_params, srch_params)

  # Performs matching the descriptors of the two images
  matches = flann_matcher.match(descriptor1,descriptor2)
  # Sorting based on the distance between the matches
  matches = sorted(matches, key=lambda x: x.distance)

  # Contains the coordinates of points from the source image which is
to be warped and for each match in matches_flann, it retrieves the
keypoint coordinates
```

```
  points2 = np.float32([keypoints1[m.queryIdx].pt for m in
matches[:200]]).reshape(-1, 1, 2)
  # Contains the coordinates of points from the Initial image and for
each match in matches_flann, it retrieves the keypoint coordinates
  points1 = np.float32([keypoints2[m.trainIdx].pt for m in
matches[:200]]).reshape(-1, 1, 2)

  # Finding fundamental matrix from the best matches using RANSAC
  points1 = np.int32(np.array(points1))
  points2 = np.int32(np.array(points2))

  # The function returns the binary mask and fundamental matrix
indicating if the point is outlier or inlier
  fmatrix, mask = cv2.findFundamentalMat(points1, points2,
cv2.FM_RANSAC)

  # Selecting only inlier points by falttening the mask array and
selecting "ONES"
  points1 = points1[mask.ravel() == 1]
  points2 = points2[mask.ravel() == 1]

  return fmatrix, points1, points2

fmatrix_storage, points1_storage, points2_storage =
findFundMat(storage_img1, storage_img2)

print("Fundamental Matrix\n ", fmatrix_storage)
```

```
Fundamental Matrix
  [[-1.57549198e-08 -1.64427913e-04  3.38995693e-02]
 [ 1.64625183e-04 -1.80356851e-06 -7.51583708e-01]
 [-3.43496080e-02  7.57782619e-01  1.00000000e+00]]
```

```
# Intrinsic Parameter for Left Camera
k1_storage = np.array([[1742.11 ,0 ,804.90],
                [0 ,1742.11 ,541.22],
                [0, 0, 1]])

# Intrinsic Parameter for Right Camera
k2_storage = np.array([[1742.11 ,0 ,804.90],
                [0 ,1742.11 ,541.22],
                [0, 0, 1]])

baseline_storage = 221.76
f_storage = 1742.11

# finding essential matrix from fundamental and intrinsic parameters
E_storage = np.dot(k2_storage.T, np.dot(fmatrix_storage,k1_storage))

print("Essential Matrix\n\n", E_storage, "\n")
```

```
# Decomposing Essential Matrix into two possible rotation matrix and
translation matrix
R1_storage,R2_storage,T_storage = cv2.decomposeEssentialMat(E_storage)

print("Rotation Matrix \n\n", R1_storage, "\n")
print("Translation Matrix \n\n", T_storage)
```

```
Essential Matrix

 [[-4.78153506e-02 -4.99030042e+02 -9.59985999e+01]
 [ 4.99628747e+02 -5.47373530e+00 -1.08020058e+03]
 [ 9.53563986e+01  1.08787534e+03  3.54016800e+00]]

Rotation Matrix

 [[ 0.63938363 -0.14435481  0.75521538]
 [-0.14504493 -0.98722775 -0.065904  ]
 [ 0.75508314 -0.06740222 -0.65215519]]

Translation Matrix

 [[ 0.90622913]
 [-0.07917647]
 [ 0.41530693]]
```

Rectification of Traproom Images

```
def draw_epilines(input_img1, input_img2, lines, pts1, pts2):
  h,w,_ = input_img1.shape

  for l,point1,point2 in zip(lines,pts1,pts2):

    #Generating random color lines
    color = tuple(np.random.randint(0,255,3).tolist())

    # Cordinates for generating a line
    x0,y0 = map(int, [0, -l[2]/l[1]])
    x1,y1 = map(int, [w, -(l[2]+l[0]*w)/l[1]])

    # Drawing the line on image 1 from the corresponding points of
image 2
    img_result1 = cv2.line(input_img1, (x0,y0), (x1,y1), color,2)

    # Representing coresponding points in image 1
    img_result1 = cv2.circle(input_img1, tuple(point1[0]), 5, color, -
1)
    img_result2 = cv2.circle(input_img2, tuple(point2[0]), 5, color, -
1)

  return img_result1, img_result2
```

```
def computeEpilines(input_img1,input_img2, input_points1,
input_points2, fund_mat):
# Creating epipolar lines of points on image 2 as lines in image 1
  lines1 = cv2.computeCorrespondEpilines(input_points2.reshape(-
1,1,2), 2, fund_mat)
  lines1 = lines1.reshape(-1,3)

  # Generating new image which draws epipolar lines
  img1_lines, img1_points = draw_epilines(input_img1, input_img2,
lines1, input_points1, input_points2)

  # Creating epipolar lines of points on image 1 as lines in image 2
  lines2 = cv2.computeCorrespondEpilines(input_points1.reshape(-1, 1,
2), 1, fund_mat)
  lines2 = lines2.reshape(-1, 3)

  # Generating new image which draws epipolar lines
  img2_lines, img2_lines = draw_epilines(input_img2, input_img1,
lines2, input_points2, input_points1)

  return img1_lines, img2_lines


def rectifyImage(input_img1, input_img2, input_points1, input_points2,
fund_mat):
  h1, w1, z1 = input_img1.shape
  h2, w2, z2 = input_img2.shape

  # Using this function to generate the homography of both images to
make the horizontal epipolar lines
  _, homography1, homography2 =
cv2.stereoRectifyUncalibrated(input_points1, input_points2, fund_mat,
(w1,h1))

  # Applying perspective projection to generate the rectified images
  img_rect1 = cv2.warpPerspective(input_img1, homography1, (w1,h1))
  img_rect2 = cv2.warpPerspective(input_img2, homography2, (w2,h2))

  combined_img = cv2.hconcat([img_rect1, img_rect2])


  return combined_img,homography1,homography2


storage1_lines, storage2_lines = computeEpilines(storage_img1,
storage_img2, points1_storage, points2_storage, fmatrix_storage)

rectified_combined_img_storage, storage_h1, storage_h2 =
rectifyImage(storage_img1, storage_img2, points1_storage,
```

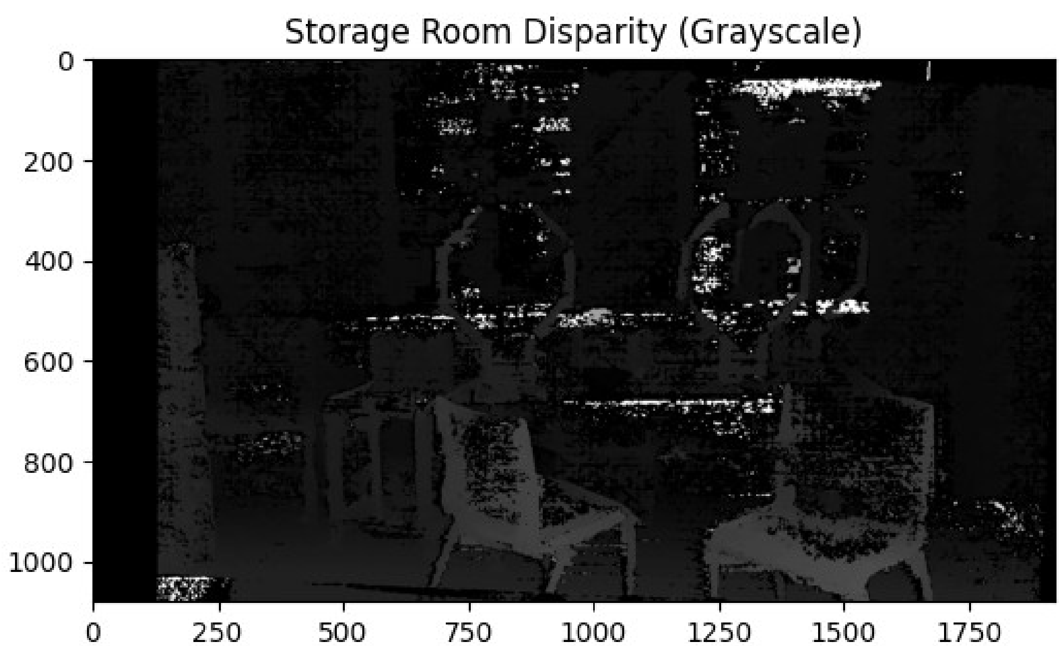
```
points2_storage, fmatrix_storage)


print("TrapRoom Image Left Homography \n",storage_h1 )
print("TrapRoom Image Right Homography \n",storage_h2 )

cv2_imshow(rectified_combined_img_storage)
```

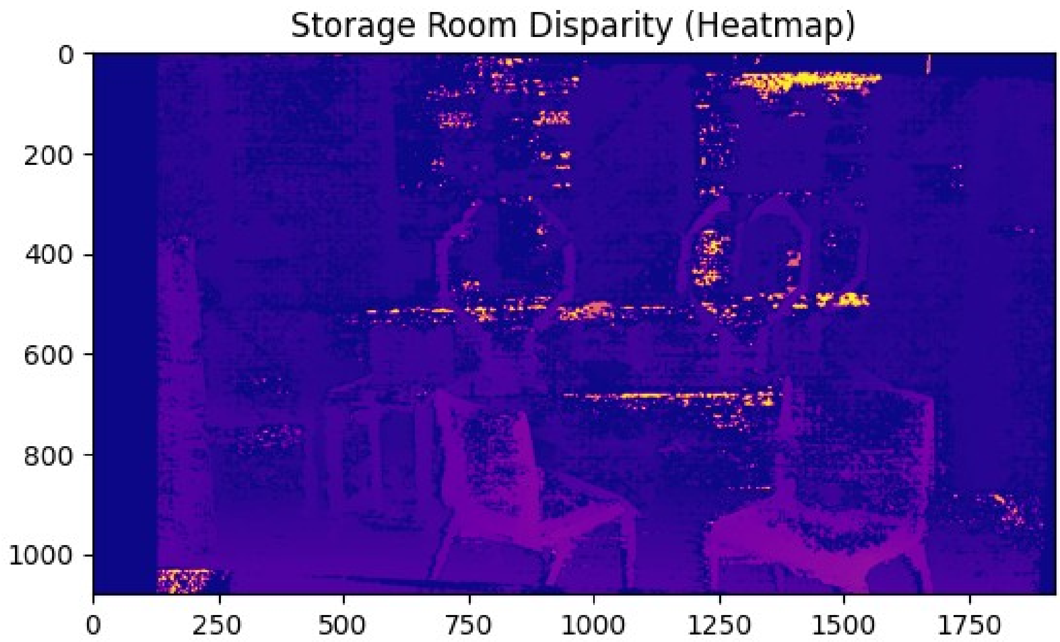
```
TrapRoom Image Left Homography
 [[-5.55818305e-01  4.42102045e-02 -2.39841495e+02]
 [ 4.28199648e-02 -7.65659012e-01 -3.80694556e+01]
 [ 2.06479549e-04 -2.09780455e-05 -9.38816208e-01]]
TrapRoom Image Right Homography
 [[ 7.34777934e-01 -6.66817731e-02  2.90621341e+02]
 [-5.65058111e-02  1.00923738e+00  4.92573959e+01]
 [-2.72009858e-04  2.46851447e-05  1.24779949e+00]]
```

Depth Map of Traproom Images

```
def dispMap(input_img1, input_img2, no_disp, min_disp, max_disp,
block, uniq_ratio):

  stereo_new = cv2.StereoSGBM_create(
      minDisparity=min_disp,
      numDisparities=no_disp,
      blockSize=block,
      uniquenessRatio=uniq_ratio,
      P1=8 * 1 * block * block,
      P2=32 * 1 * block * block,
  )

  disparity_img = stereo_new.compute(input_img1, input_img2)

  disparity_img = cv2.normalize(disparity_img, disparity_img,
alpha=255,
                                beta=0, norm_type=cv2.NORM_MINMAX)

  disparity_img = np.uint8(disparity_img)
```

```
    return disparity_img



# Reading the images
storage_img1 =
cv2.imread('/content/drive/MyDrive/ENPM673/projects_assets/Project3/
storage room/im0.png', cv2.IMREAD_GRAYSCALE)
storage_img2 =
cv2.imread('/content/drive/MyDrive/ENPM673/projects_assets/Project3/
storage room/im1.png', cv2.IMREAD_GRAYSCALE)

storage_disparity = dispMap(storage_img1,storage_img2,100, 30,80,5,
20)

plt.title('Storage Room Disparity (Grayscale)')
plt.imshow(storage_disparity, cmap='gray')
plt.show()

plt.title('Storage Room Disparity (Heatmap)')
plt.imshow(storage_disparity, cmap='plasma')
plt.show()
```



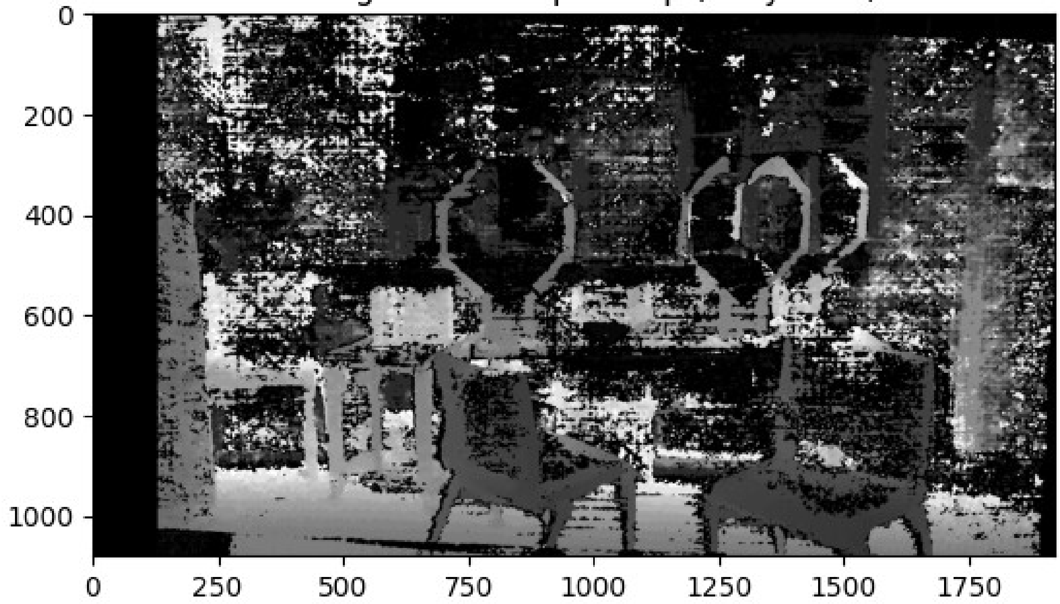



```
def disp2depth(disp_img, flength, basel):
  h1,w1 = disp_img.shape
  depth_img = disp_img
  for i in range(w1):
    for j in range(h1):
      depth_img[j,i] = ((flength*basel)/disp_img[j,i])*0.01

  return depth_img

storage_depth = disp2depth(storage_disparity, f_storage,
baseline_storage)
```

```
<ipython-input-137-922184edb65c>:6: RuntimeWarning: divide by zero
encountered in divide
  depth_img[j,i] = ((flength*basel)/disp_img[j,i])*0.01
<ipython-input-137-922184edb65c>:6: RuntimeWarning: invalid value
encountered in cast
  depth_img[j,i] = ((flength*basel)/disp_img[j,i])*0.01
```

```
plt.title('Storage Room Depth Map (Grayscale)')
plt.imshow(storage_depth, cmap='gray')
plt.show()

plt.title('Storage Room Depth Map (Heatmap)')
plt.imshow(storage_depth, cmap='plasma')
plt.show()
```

Storage Room Depth Map (Grayscale)



Storage Room Depth Map (Heatmap)