Project 2:

Problem1:

1.Read the video and extract individual frames using OpenCV.

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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mo
```

Read the video file from the google drive

```
#Import opencv2
import cv2
from google.colab.patches import cv2_imshow

#Reading the video link path for proj2_v2.mp4
video_link = '/content/drive/MyDrive/ENPM673/project2/proj2_v2.mp4'
```

Reading individual frames from the video and storing it in a list(frames)

```
#Function to read the frames
def store_frames(vid_path):
  #Read video using videoCapture
  obj_video = cv2.VideoCapture(vid_path)

  #defining empty list to store all the frames
  frames  = []

  #Iterate util video is open
  while obj_video.isOpened():

      # Capturing  frames
      ret, frame = obj_video.read()

      # Check if frames are read correctly by reading ret
      if not ret:
          break
      else:
          frames.append(frame)
  return frames

#frames list has all the indiv frame
frames = store_frames(video_link)
```

2.Skip blurry frames (use Variance of the Laplacian and decide a suitable threshold) Note: Any value below 150 for the Variance of the Laplacian, suggests that it's a blurry image.

To find laplacian

Laplacian kernel H is

0 1 0

1 -4 1

0 1 0

F is Input Frames

G is output Image

G = H * F

Take variance of G to detect blurriness

if var < Threshold then it is blurry image

```
import numpy as np

#Function to do variance of laplacian and say image is bluury or not
def blur_detect(frame, threshold):
    # Convert the frame to grayscale
    gray_frame  = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # Define the Laplacian kernel
    laplacian_kernel = np.array([[0, 1, 0],
                                 [1, -4, 1],
                                 [0, 1, 0]], dtype=np.float32)

    # Convolve the Laplacian kernel and the grayscale frame
    laplacian_frame = cv2.filter2D(gray_frame, cv2.CV_32F, laplacian_kernel)

    #TAke variance using np.where
    variance = np.var(laplacian_frame)

    #return true if variance is less than threshold--> blurry image
    #return false for non blurry image

    if(variance < threshold):
      return True
    else:
      return False

#Define threshold to say variance of laplacian is blurry
threshold = 110

#define non blurry frames list
non_blurry_frames = []

#Loop over all the frames and send it to blur_detect function
for index,frame in enumerate(frames):
  val = blur_detect(frame, threshold)
  if(not(val)):
    non_blurry_frames.append(frame)

print("Total frames is:", len(frames))
print("Non blurry frames is:",len(non_blurry_frames))
print("Blurry frames is:", len(frames) - len(non_blurry_frames))
```

```
print('Blurry frames is: ', len(frames) - len(non_blurry_frames))
```

```
Total frames is: 386
Non blurry frames is: 201
Blurry frames is: 185
```

3.Segment out the unwanted background area (example: convert to gray scale and keep white regions)

use cvtcolor to change to grayscale and use np.where to mask only white area of frames

```
#Define sedment frames
segment_frames = []
non_blurry_frames = non_blurry_frames[:-30]

for index,frame in enumerate(non_blurry_frames):

  #cvtcolor to convert bgr to grayscale
  gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

  # Perform thresholding using np.where with threshold 215
  segmented_image = np.where(gray_frame > 215, 255, 0).astype(np.uint8)
  segment_frames.append(segmented_image)
#display one image
cv2_imshow(segmented_image)
```

4.Detect edges pixels in each frame (you can use any edge detector)

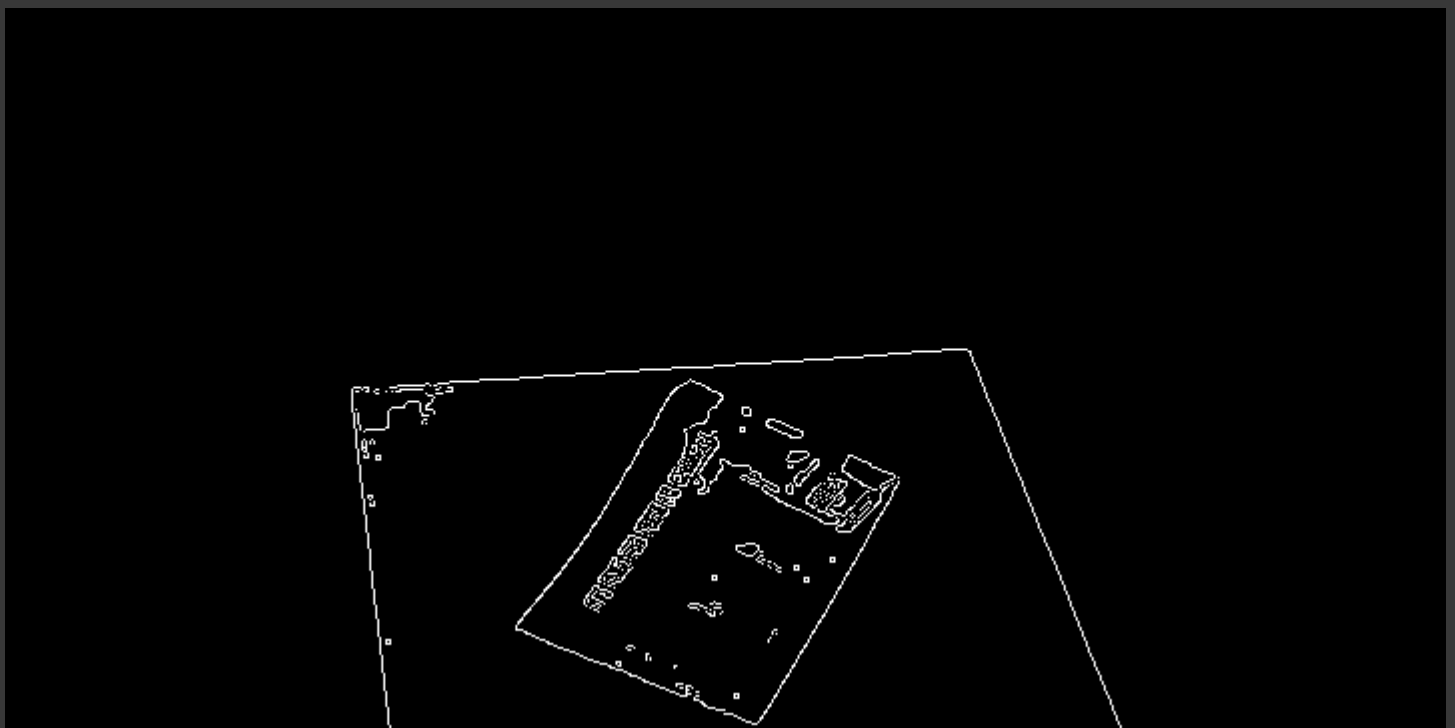
Using canny edge detector

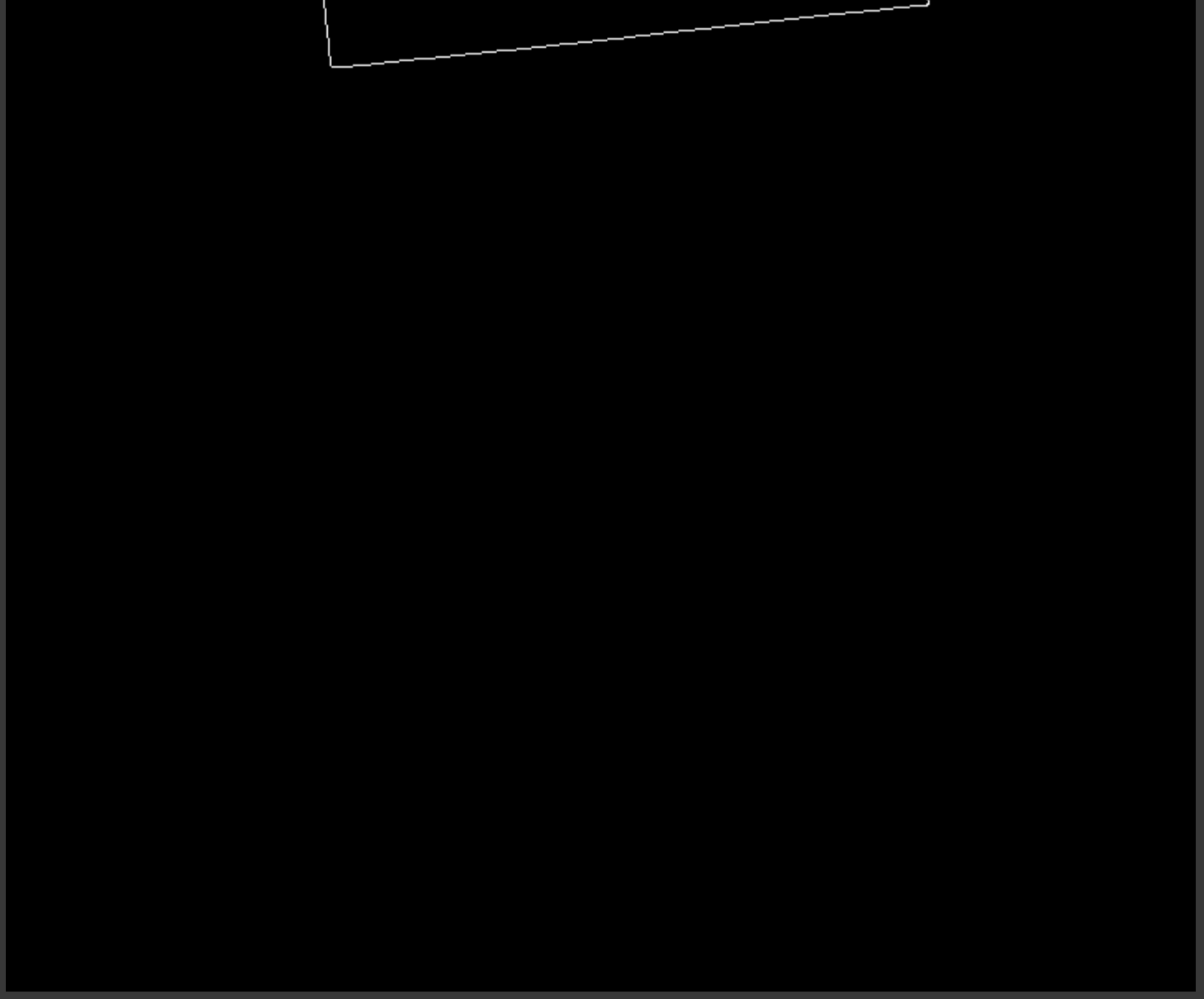
cv2.canny()

using minVAl as 250 so that val less that it is nonedge

using maxVal as 254 so that val greter that that is edge

```
#Defining edges frames in list
edge_frames = []
for index,frame in enumerate(segment_frames):
  #using cv2.canny to find the edges minVal=250, maxVal=254
  edges = cv2.Canny(frame, 250, 254)
  edge_frames.append(edges)
#display
cv2_imshow(edges)
```





5.Use the detected edge pixels to extract straight lines in each frame (hint: use Hough Transform)

6. Filter out “short” lines and only keep a few dominant lines.

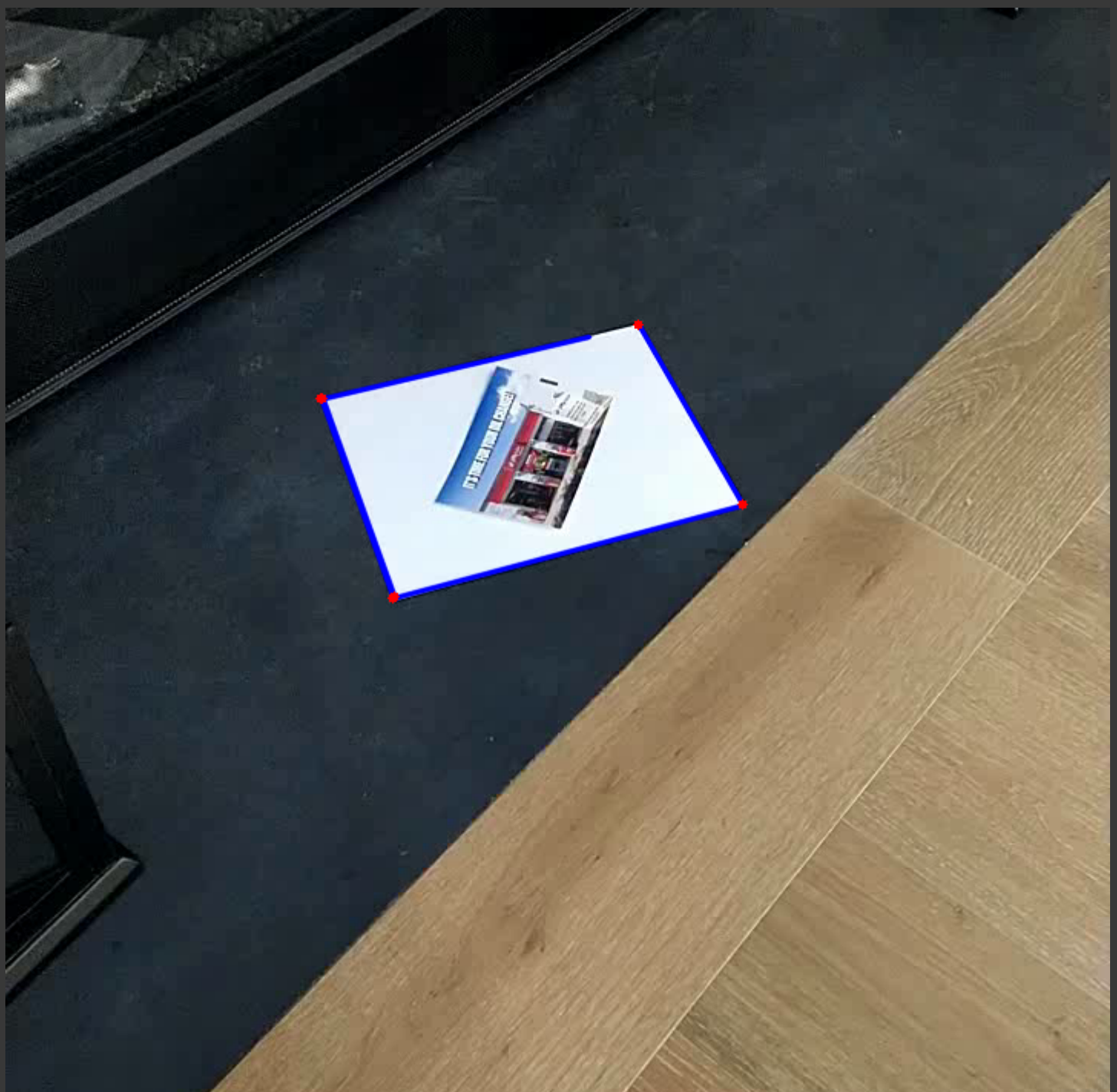
7. Compute the intersections of the Hough Lines – these are the putative corners of the paper.

Using Hough transform Probablistic and removing all short lines

```
import math
import random

#Define a list for hough intersect points
hough_intersect = []

#Function to calculate slope and intercept
def calculate_eqn_line(pt1, pt2):

    x1, y1 = pt1
    x2, y2 = pt2

    # Slope
    m = (y2 - y1) / (x2 - x1)

    # INtercept
    b = (y1 - m * x1)
```

```
    b = (y1 - m * x1)

    return (m, b)

#Function to find intersecting points
def detect_intersect_lines(coeffs1, coeffs2):

    #Split m and b
    m1, b1 = coeffs1
    m2, b2 = coeffs2

    # Condition to avoid parallel lines and lines lying on top
    if m1 == m2:
        if b1 == b2:
            return None, None
        else:
            return None, None

    # Calculate x
    x_intersect = (b2 - b1) / (m1 - m2)

    # Calculate t
    y_intersect = (m1 * x_intersect + b1)

    if((x_intersect<0) or (y_intersect<0)):
      return None, None
    else:
      return x_intersect, y_intersect

#Function to do Hough Transform
def Hough_Transform(frame, non_blur_frame):

  intersect_pts = []

  #define line coefficients
  line_coeff = []

  #Hough transform with itersection points as 40
  lines = cv2.HoughLinesP(frame, 1, np.pi / 180, 40, None, 100, 10)

  # Draw detected lines on the frame
  if lines is not None:
    for i in range(0, len(lines)):
      pts = lines[i][0]
      m, b = calculate_eqn_line((pts[0], pts[1]), (pts[2], pts[3]))
      line_coeff.append((m,b))

      # Generate lines
      cv2.line(non_blur_frame, (pts[0], pts[1]), (pts[2], pts[3]), (255,0,0), 2, cv2.LINE_

    #Find intersection of points
    for i in range(0, len(line_coeff)):
      for j in range(0, len(line_coeff)):
        if(i!=j):
          x,y = detect_intersect_lines(line_coeff[i], line_coeff[j])
          if(x!=None):
            intersect_pts.append((x,y))

    pts = []
```

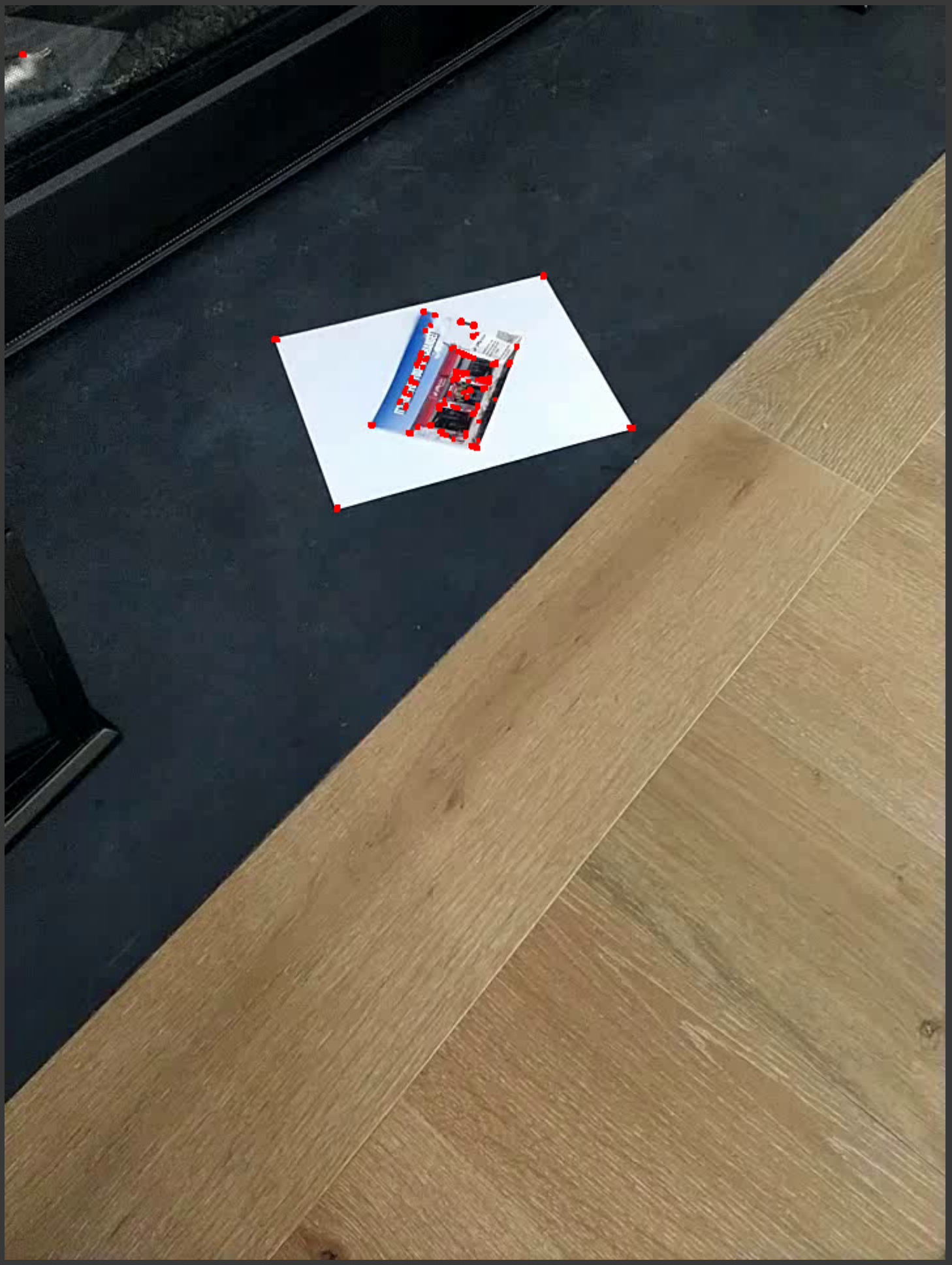
```
    #Draw Intersection points
    for i in range(0, len(intersect_pts)):
      pts.append((int(intersect_pts[i][0]), int(intersect_pts[i][1]) ))
      cv2.circle(non_blur_frame, (int(intersect_pts[i][0]), int(intersect_pts[i][1])), 3,


    #Appending the hough intersection points
    hough_intersect.append(pts)

  return non_blur_frame

#define output from hough out list
hough_out = []

#iterte over all o/ps from canny
for index,frame in enumerate(edge_frames):
  temp = non_blurry_frames[index].copy()
  out = Hough_Transform(frame, temp)
  if(index==0):
    cv2_imshow(out)
```



8.Verify the existence of those corners with Harris corner detector. (use OpenCV built-in function)

9. Filter out remaining extraneous corners that are not the 4 corners of the paper.

```
i=0
#List to store all harris corner points
harris_x = []
harris_y = []

#function to do the harris corner
def harriscorner(frame):
  #convert to gray
  img = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

  #Changing datatype to float 32
  img = np.float32(img)

  #cv2.cornerHarris method
  out = cv2.cornerHarris(img, 3, 7, 0.04)

  # Dilate corners
  out = cv2.dilate(out, None)

  #Display
  frame[out > 0.02 * out.max()]=[0, 0, 255]

  #storing corner points
  corners = np.argwhere(out > 0.01 * out.max())

  # Extract x and y coordinates and swap them
  harris_x.append(corners[:, 1])
  harris_y.append(corners[:, 0])

  global i
  if(i==0):
    i+=1
    cv2_imshow(frame)

#iterte over all o/ps from canny
for index,frame in enumerate(non_blurry_frames):
  temp = frame.copy()
  harriscorner(temp)
```



Filter out remaining extraneous corners that are not the 4 corners of the paper.

```
import math
```

```
#Function to filter out the points based on distance
def filter_pts(hough_pts, harris_x,harris_y, thresh, frame_index):
    overlap_pts = []

    #Hough Points
    for i in range(0,len(hough_pts[frame_index])):
      x2, y2 = hough_pts[frame_index][i]
      for j in range(0,len(harris_x[frame_index])):
        #Harris points
        x1 = harris_x[frame_index][j]
        y1 = harris_y[frame_index][j]

        #Take eucledian distance to remove overlap points with threshold
        distance = math.sqrt((x2- x1) ** 2 + (y2-y1) ** 2)
        if(distance <=thresh):
          overlap_pts.append((x2, y2))
          break
    return overlap_pts

video_conv = []

#iterte over all list
for index,frame in enumerate(non_blurry_frames):
  temp = frame.copy()
  final_pts = filter_pts(hough_intersect, harris_x, harris_y, 1, index)


  # Define the threshold distance for x and y coordinates
  threshold_x = 2
  threshold_y = 2

  # List to store unique points
  unique_points = []

  # Function to check if a point is a duplicate
  def is_dupl(p1, p2, threshold_x, threshold_y):
    return abs(p1[0] - p2[0]) <= threshold_x and abs(p1[1] - p2[1]) <= threshold_y

  # Iterate through each pt
  for point in final_pts:
    is_unique = True

    # Check if the current point is a duplicate of any previously seen point
    for seen_point in unique_points:
      if is_dupl(point, seen_point, threshold_x, threshold_y):
        is_unique = False
        break

    # Add the point to the list of unique points if it is not a dupl
    if is_unique:
      unique_points.append(point)

  #Sorting for drawing line to plot in line
  sorted_points = sorted(unique_points, key=lambda pt: pt[0])

  # Plot circles--> intersection circles - red  and lines blue
  if(len(sorted_points)==4):
    for i in range(0,len(sorted_points)):
      x, y = sorted_points[i]
```

```
      #plot circles
      cv2.circle(temp, (int(x), int(y)), radius=5, color=(0, 0, 255), thickness=-1)  # Dra

    #plot lines
    cv2.line(temp, (sorted_points[0][0], sorted_points[0][1]), (sorted_points[1][0], sorte
    cv2.line(temp, (sorted_points[1][0], sorted_points[1][1]), (sorted_points[3][0], sorte
    cv2.line(temp, (sorted_points[2][0], sorted_points[2][1]), (sorted_points[3][0], sorte
    cv2.line(temp, (sorted_points[0][0], sorted_points[0][1]), (sorted_points[2][0], sorte
    video_conv.append(temp)

    #Display or save the image with circles
    #cv2_imshow(temp)
```

```
video_output = '/content/drive/MyDrive/ENPM673/project2/output.mp4'
fourcc = cv2.VideoWriter_fourcc(*'mp4v')
video_writer = cv2.VideoWriter(video_output, fourcc, 25, (video_conv[0].shape[1], video_co

# Write each image to the video
for image in video_conv:
    video_writer.write(image)

# Release the VideoWriter object
video_writer.release()
print("DOne saving...")
```

```
DOne saving...
```

Problem2

1.Reading the images from google drive

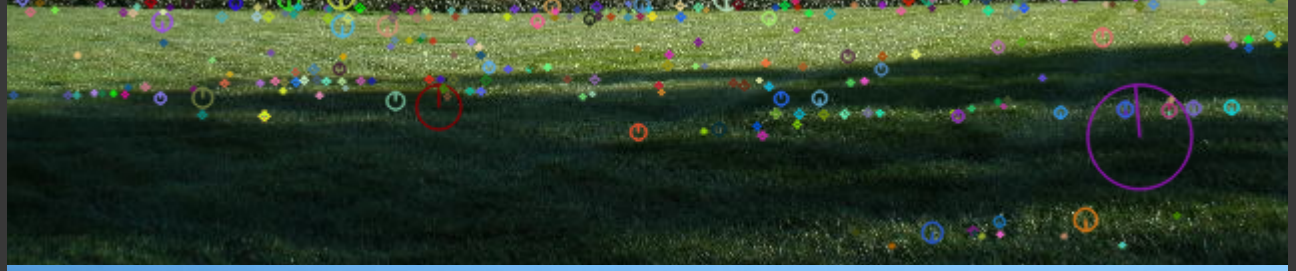
```
#Mount your google drive
from google.colab import drive
import cv2
from google.colab.patches import cv2_imshow

#Reading from drive
drive.mount('/content/drive')
img1 = cv2.imread('/content/drive/MyDrive/ENPM673/project2/Images/PA120275.JPG')
img2 = cv2.imread('/content/drive/MyDrive/ENPM673/project2/Images/PA120274.JPG')
img3 = cv2.imread('/content/drive/MyDrive/ENPM673/project2/Images/PA120273.JPG')
img4 = cv2.imread('/content/drive/MyDrive/ENPM673/project2/Images/PA120272.JPG')

#Display the images
cv2_imshow(img1)
cv2_imshow(img2)
cv2_imshow(img3)
cv2_imshow(img4)
```

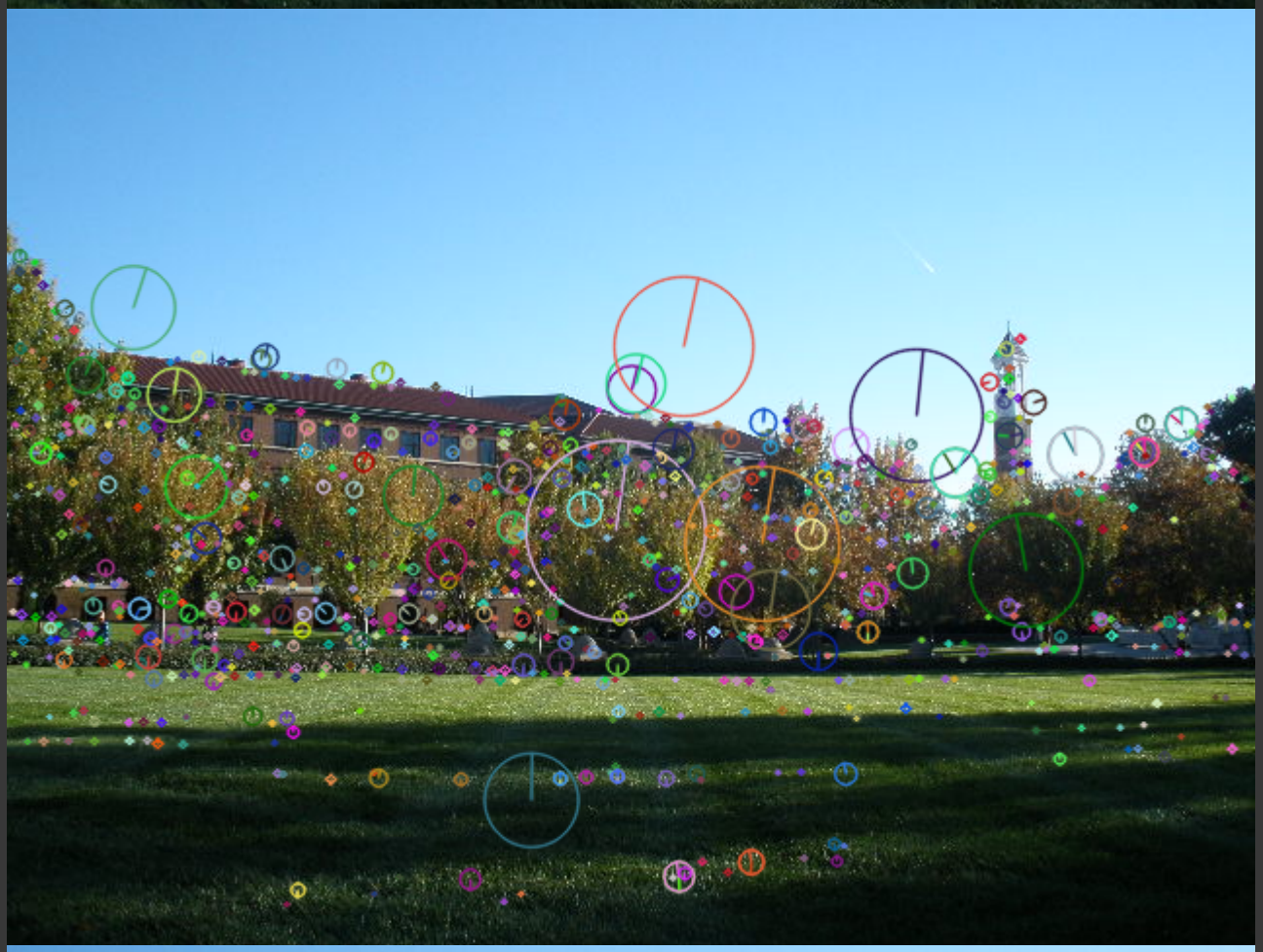
```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mo
```

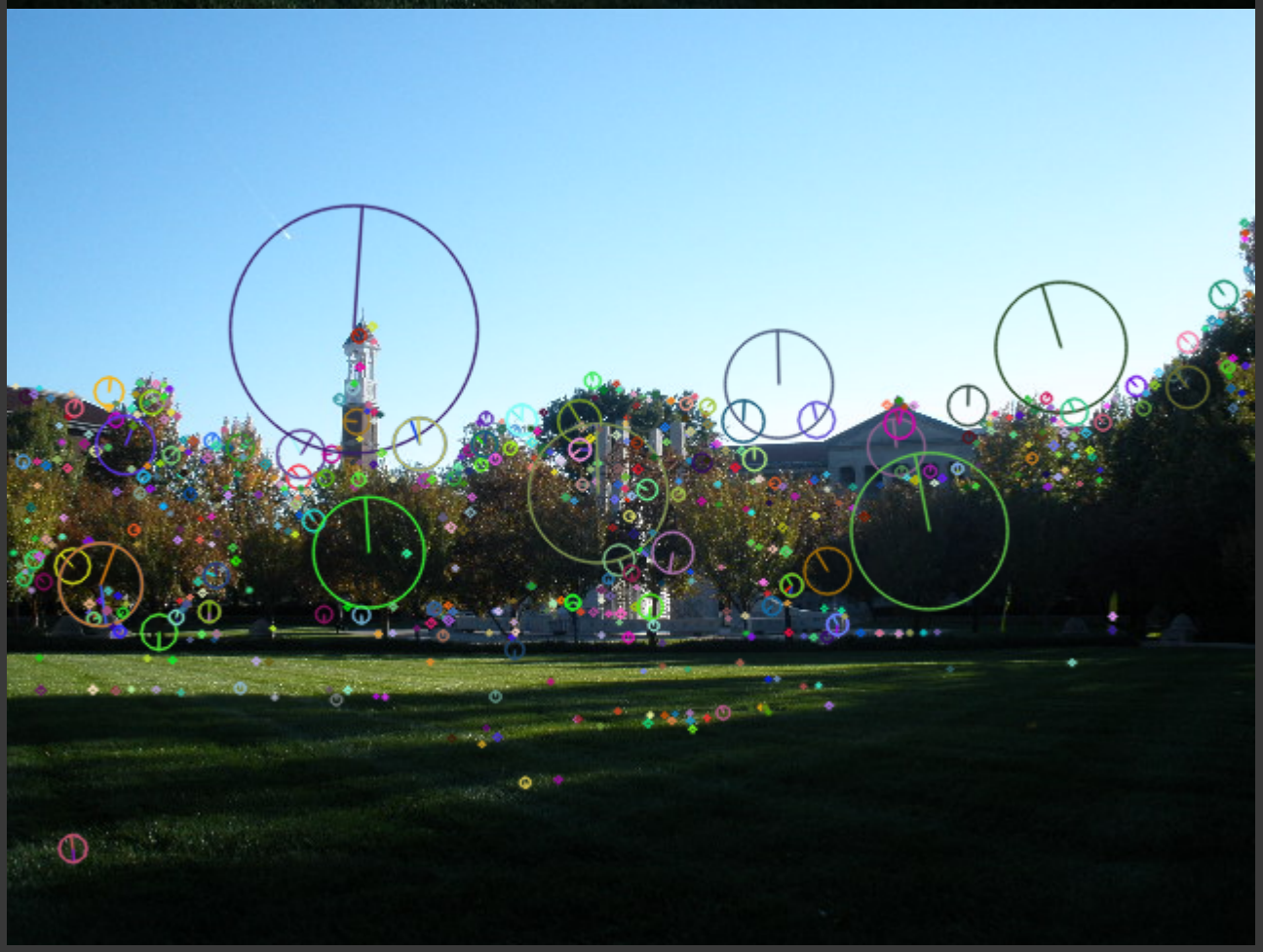
Extracting the features from each frame using SIFT

```
#converting the image to grayscale
img1_gray = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
img2_gray = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
img3_gray = cv2.cvtColor(img3, cv2.COLOR_BGR2GRAY)
img4_gray = cv2.cvtColor(img4, cv2.COLOR_BGR2GRAY)

#SIFT function
def SIFT(Image):

    #create sift
    SIFTDetector= cv2.xfeatures2d.SIFT_create()

    #get keypoints and descriptors
    kp, des = SIFTDetector.detectAndCompute(Image, None)

    return kp, des

#Calling SIFT for all 4 images to get all features in 4 images
kp1, des1 = SIFT(img1_gray)
```

```
kp2, des2 = SIFT(img2_gray)
kp3, des3 = SIFT(img3_gray)
kp4, des4 = SIFT(img4_gray)

#Drawing the keypoints in the images
sift1 = cv2.drawKeypoints(img1, kp1, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
sift2 = cv2.drawKeypoints(img2, kp2, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
sift3 = cv2.drawKeypoints(img3, kp3, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
sift4 = cv2.drawKeypoints(img4, kp4, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

#Display
cv2_imshow(sift1)
cv2_imshow(sift2)
cv2_imshow(sift3)
cv2_imshow(sift4)
```







Matching the features between consecutive images using Knn

Image1 and Image2

Image2 and Image3

Image3 and Image4

Taking only good matches using ratio test

Finding the Homographies using cv2.findHomography

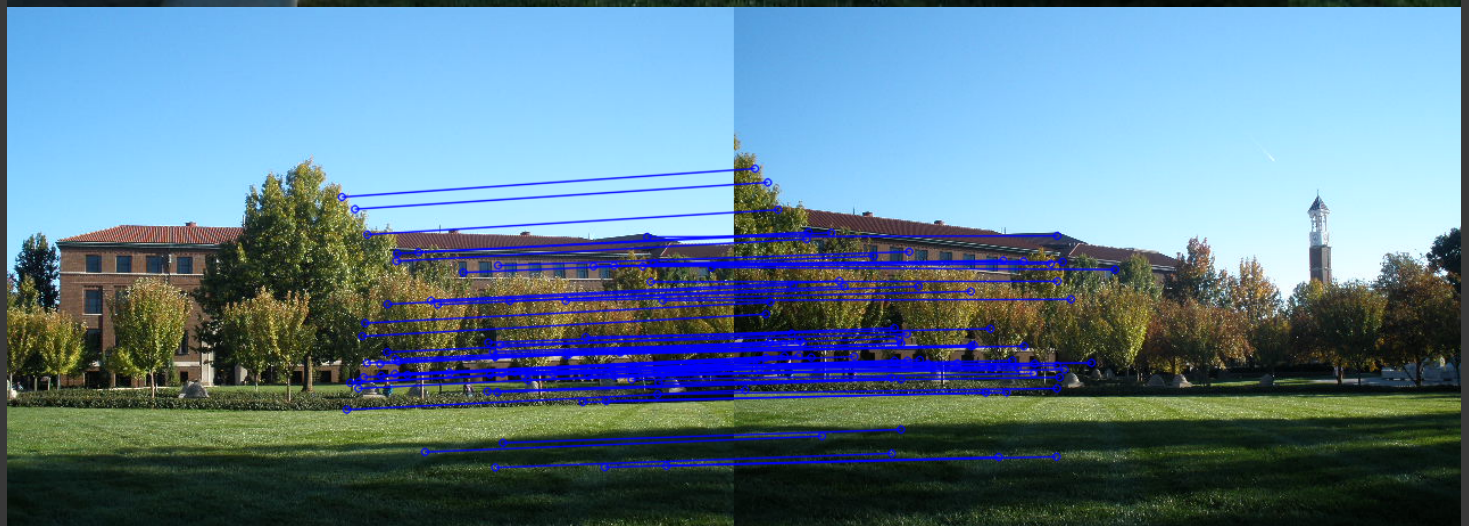
WArping is done using cv2.warpPerspective

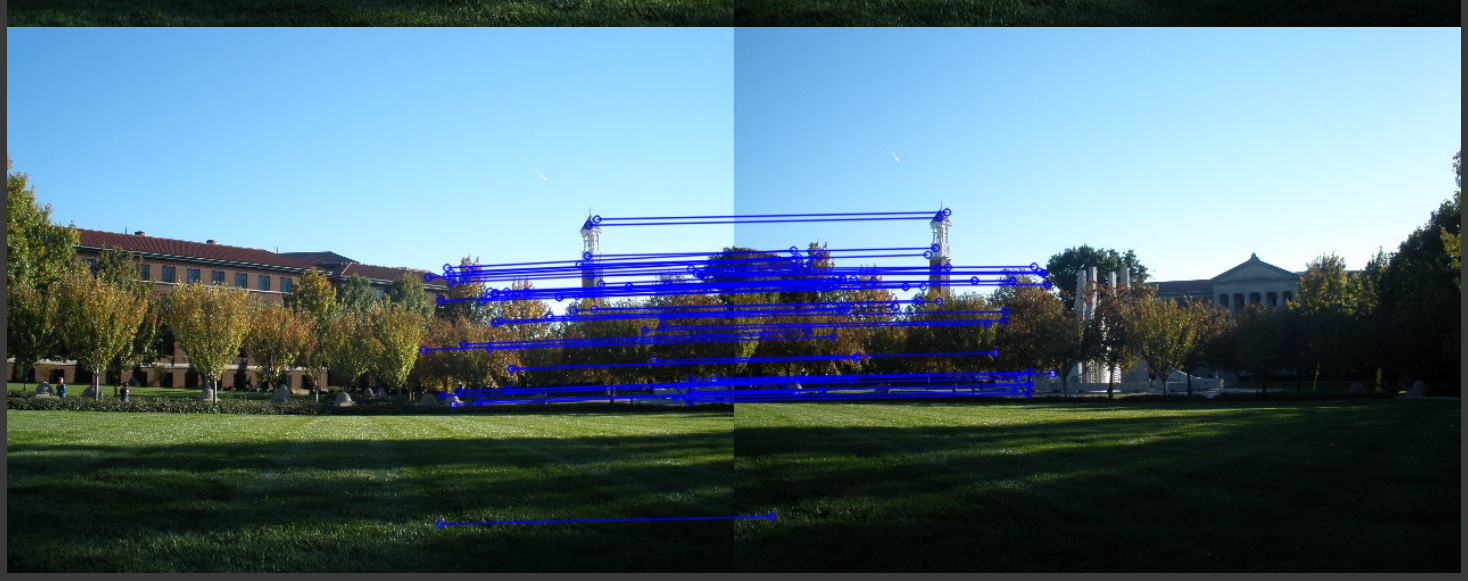
remove_black_pixels function removes the black formed pixel clusters

```
import numpy as np

#KNN Matcher function to match
def KNNmatcher(kp1, des1, kp2, des2):
    # BFMatcher
    bf = cv2.BFMatcher()

    #Matches list
    matches = bf.knnMatch(des1,des2, k=2)

    #Filter out only good matches from ratio test method with distance as 0.5
    good_match = []
    for m,n in matches:
        if m.distance < 0.5*n.distance:
            good_match.append([m])

    return good_match

#Calling matchers for all keypoints
matches12 = KNNmatcher(kp1, des1, kp2, des2)
matches23 = KNNmatcher(kp2, des2, kp3, des3)
matches34 = KNNmatcher(kp3, des3, kp4, des4)

#Plot the matcher to check the results
temp1 = cv2.drawMatchesKnn(img1, kp1, img2, kp2, matches12, None, (255,0,0) , None, flags=2)
temp2 = cv2.drawMatchesKnn(img2, kp2, img3, kp3, matches23, None, (255,0,0) , None, flags=2)
temp3 = cv2.drawMatchesKnn(img3, kp3, img4, kp4, matches34, None, (255,0,0) , None, flags=2)

#display
cv2_imshow(temp1)
cv2_imshow(temp2)
cv2_imshow(temp3)

#Declare query and train idx lists for all matchers
src_pts12 = []
des_pts12 = []

src_pts23 = []
des_pts23 = []

src_pts34 = []
des_pts34 = []
```

```
#Check if matches is greater than 10 to qualify as good match
if len(matches12) > 10:
    src_pts12 = np.float32([ kp1[m[0].queryIdx].pt for m in matches12 ]).reshape(-1,1,2)
    des_pts12 = np.float32([ kp2[m[0].trainIdx].pt for m in matches12 ]).reshape(-1,1,2)

if len(matches23) > 10:
    src_pts23 = np.float32([ kp2[m[0].queryIdx].pt for m in matches23 ]).reshape(-1,1,2)
    des_pts23 = np.float32([ kp3[m[0].trainIdx].pt for m in matches23 ]).reshape(-1,1,2)

if len(matches34) > 10:
    src_pts34 = np.float32([ kp3[m[0].queryIdx].pt for m in matches34 ]).reshape(-1,1,2)
    des_pts34 = np.float32([ kp4[m[0].trainIdx].pt for m in matches34 ]).reshape(-1,1,2)

# Homography mat -- RANSAC
H12, mask = cv2.findHomography(des_pts12, src_pts12, cv2.RANSAC, 5.0)
H23, mask = cv2.findHomography(des_pts23, src_pts23, cv2.RANSAC, 5.0)
H34, mask = cv2.findHomography(des_pts34, src_pts34, cv2.RANSAC, 5.0)

# Print the estimated homography matrix
print("\nHomography Matrix 12\n:")
print(H12)
print("\nHomography Matrix 23\n:")
print(H23)
print("\nHomography Matrix 34\n:")
print(H34)

#Function to check if entire row or col is black to remove it
def remove_black_blocks(Img):

    #check from top row
    while not np.any(Img[0]):
      Img = Img[1:]
    #check from bottom row
    while not np.any(Img[-1]):
      Img = Img[:-2]
    #check from col from left
    while not np.any(Img[:,0]):
      Img = Img[:,1:]
    #check from col from right
    while not np.any(Img[:,-1]):
      Img = Img[:,:-2]

    return Img

#Function to stitch the images
def panorama_img(img2, img1, H):
    #Calcuate width
    width = img2.shape[1] + img1.shape[1]
    #Calculate height
    height = max(img2.shape[0], img1.shape[0])

    #warping the image using H
    final_img = cv2.warpPerspective(img2, H, (width,height))
    final_img[0:img1.shape[0], 0:img1.shape[1]] = img1

    return final_img

#stitch 1 and 2 image
out12 = panorama_img(img2, img1, H12)
```

```
#stitch 3 and 4th image
out34 = panorama_img(img4, img3, H34)

#display
cv2_imshow(out12)
cv2_imshow(out34)

#convert one coord to other
new_H = np.dot(H23, H12)

out12_wo_black = remove_black_blocks(out12)
out34_wo_black = remove_black_blocks(out34)

#Stitching the final panorama images
stitch_final = panorama_img(out34_wo_black, out12_wo_black, new_H)

#Display the final image
cv2_imshow(stitch_final)
```





```
Homography Matrix 12
:
[[ 8.58188937e-01  2.76681994e-02  2.90024910e+02]
 [-1.07552917e-01  9.37377886e-01  2.75769887e+01]
 [-2.23927853e-04 -5.44238027e-05  1.00000000e+00]]

Homography Matrix 23
:
[[ 8.65345186e-01  3.21554924e-02  2.72746650e+02]
 [-1.11953079e-01  9.30232669e-01  3.39350809e+01]
 [-1.92017342e-04 -6.65764113e-05  1.00000000e+00]]

Homography Matrix 34
:
[[ 8.38147614e-01  2.27158217e-02  3.32331585e+02]
 [-1.08141084e-01  9.50444078e-01  2.57756876e+01]
 [-2.56593750e-04 -2.53890970e-05  1.00000000e+00]]
```

In general, why does panoramic mosaicing work better when the camera is only allowed to rotate at its camera center?

We cound see that image 2 and 3 and 4 are not rotated at center so it caused improper stitching and poor 3 and 4 image qualities If it is rotated as its camera center we could avoid improper image registration, better image quality

When the camera rotates around its center, the images taken have less distortion and are easier to align. This is because the main change between the images is rotation, which is simpler to account for than other types of movement.

Because the images are easier to align, the final stitched panorama looks better. It has fewer visible seams or misalignments, resulting in a higher-quality image with sharper details and fewer visual errors.

When the camera rotates around its center, it means that it's spinning without moving from its spot. This keeps everything in the picture in the same place relative to each other. So when you put the pictures together, things line up better because they haven't moved around as much.

Because everything stays in the same place, objects at different distances from the camera don't look like they've shifted position. This makes the final picture look more natural because things that are far away still look like they're far away, and things that are close still look close.