In [ ]:
```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow
```

In [ ]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remo
unt, call drive.mount("/content/gdrive", force_remount=True).
```

In [ ]:
```
%cd /content/gdrive/My Drive/ENPM673/Project 2
```

```
/content/gdrive/My Drive/ENPM673/Project 2
```

Task 1.

Design a video processing pipeline to find the 4 corners of the paper using Edge Detection, Hough Transform and Corner Detection.

Overlay the boundary(edges) of the paper and highlight the four corners of the paper. The boundary(edges) of the paper and highlight the four corners of the paper. (Note: that you must remove any frames which are blurry by using Variance of Laplacian and report the number of blurry frames removed). Example Pipeline:

1. Read the video and extract individual frames using OpenCV.
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.
3. Segment out the unwanted background area (example: convert to gray scale and keep white regions)
4. Detect edges pixels in each frame (you can use any edge detector)
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.
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.
10. Generate the output video in which you have to overlay the 4 blue boundary lines and 4 red corners of the paper in each frame (excluding the blurry frames).

In [91]:
```
# Read the video and extract individual frames using OpenCV

path = './videos/proj2_v2.mp4'
cap = cv2.VideoCapture(path)
if not cap.isOpened():
  print("ERROR: Could not open the video")
  exit()

# Laplacian function for Step 2
def laplacian(gray):
  laplacian_kernel = np.array([[0, 1, 0],
                               [1, -4, 1],
                               [0, 1, 0]], dtype=np.float64)
  laplacian = cv2.filter2D(gray, -1, laplacian_kernel)
  return laplacian
# Intersection function for Step 7
def find_intersection(line1, line2):
    x1, y1, x2, y2 = line1
    x3, y3, x4, y4 = line2

    denom = (x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4)
    if denom == 0:
        return None  # parallel lines
    px = ((x1*y2 - y1*x2) * (x3 - x4) - (x1 - x2) * (x3*y4 - y3*x4)) /
    py = ((x1*y2 - y1*x2) * (y3 - y4) - (y1 - y2) * (x3*y4 - y3*x4)) /
    return (px, py)

# Initialize Video Writer
fourcc = cv2.VideoWriter_fourcc(*'mp4v')  # Define the codec
output_path = './videos/output_proj2_v2.mp4'
frame_width = int(cap.get(3))
frame_height = int(cap.get(4))
out = cv2.VideoWriter(output_path, fourcc, 20.0, (frame_width, frame_h
c = 0
while True:
  ret, frame = cap.read()
  if not ret:
    print("Could not recieve the frame.")
    break

#------- Step 2 -------
#  Skip blurry frames (use Variance of the Laplacian and decide a suit
# Note: Any value below 150 for the Variance of the Laplacian, suggest
  gray_img = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
  var = laplacian(gray_img).var()

  if var<100:
    print("it's a blurry image, skipping...")
    c+=1
    continue
```

```
  else:

#------- Step 3 -------
# Segment out the unwanted background area (example: convert to gray s
    # x, thresh = cv2.threshold(gray_img,220,255, cv2.THRESH_BINARY)
    y = cv2.GaussianBlur(gray_img,(5,5), 0)
    thresh = np.where(y>220,255,0).astype(np.uint8)

#------- Step 4 -------
# Detect edges pixels in each frame (you can use any edge detector)
    edge = cv2.Canny(thresh,100,200)

#------- Step 5 & 6 -------,
# Use the detected edge pixels to extract straight lines in each frame
# Filter out "short" lines and only keep a few dominant lines
    lines = cv2.HoughLinesP(edge, 1, np.pi / 180, threshold=60, minLin
    for line in lines:
        x1, y1, x2, y2 = line[0]
        cv2.line(frame, (x1, y1), (x2, y2), (255, 0 , 0), 2)
#------- Step 7 -------
# Compute the intersections of the Hough Lines – these are the putativ
    intersections = []
    for i in range(len(lines)):
        for j in range(i+1, len(lines)):
            intersection = find_intersection(lines[i][0], lines[j][0])
            if intersection:  # Ensure the intersection point is valid
                intersections.append(intersection)
                # For visualization, mark the intersection
                # cv2.circle(frame, (int(intersection[0]), int(interse
#------- Step 8 -------
# Verify the existence of those corners with Harris corner detector. (

#------- Step 9 -------
# Filter out remaining extraneous corners that are not the 4 corners o

    dest = cv2.cornerHarris(gray_img, 2, 3, 0.04)
    corners = np.argwhere(dest>0.5*dest.max())
    # dest_dilated = cv2.dilate(dest, None) # Results are marked throu
    # frame[dest_dilated > 0.01 * dest_dilated.max()]=[255, 0, 0]  # R
    # Verify intersections with Harris corners
    verified_intersections = []
    for intersection in intersections:
      for corner in corners:
        if np.linalg.norm(intersection - corner[::-1])<20:
          verified_intersections.append(intersection)
    for i in verified_intersections:
      cv2.circle(frame, (int(i[0]),int(i[1])),  5, (0, 0, 255), -1)


#------- Step 10 -------
```

```
# Generate the output video in which you have to overlay the 4 blue bo
# # and 4 red corners of the paper in each frame (excluding the blurry
    out.write(frame)

    cv2_imshow(frame)
    print(c, ' frames skipped. ')
  if cv2.waitKey(1) == ord('q'):
    break
out.release()  # Finalize the video file
cap.release()
cv2.destroyAllWindows()
```

Task 2.

Part A

Given four overlapping images of a far-way building taken from the same camera (images may be taken with both rotation and slight translation). Design an image processing pipeline to stitch these images to create a panoramic image. (Note: You can consider far away objects/features to be approximately on the same plane.)

Part B

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

Example pipeline:

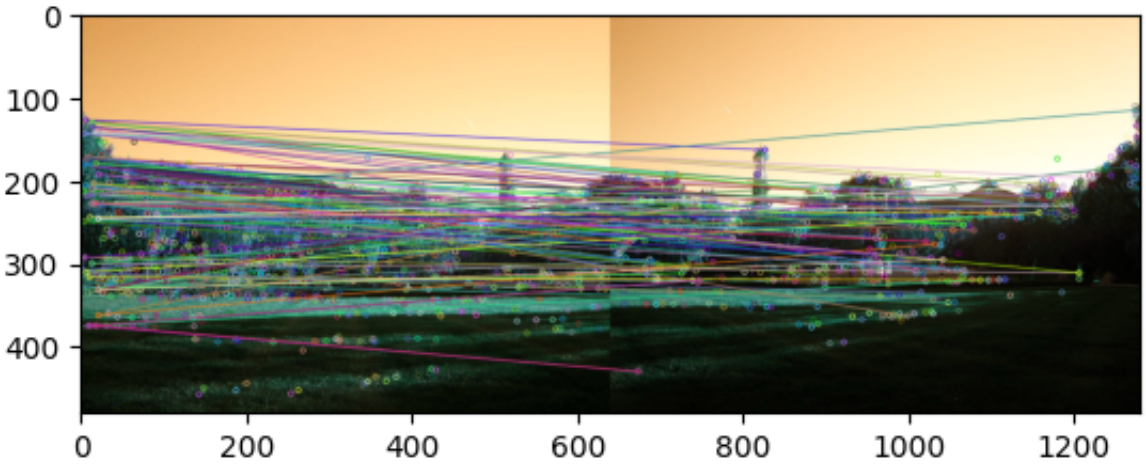
1. Extract features from each frame (You can use any feature extractor and justify).
2. Match the features between each consecutive image and visualize them. (hint: Use RANSAC)
3. Compute the homographies between the pairs of images.
4. Combine these frames together using the computed homographies.

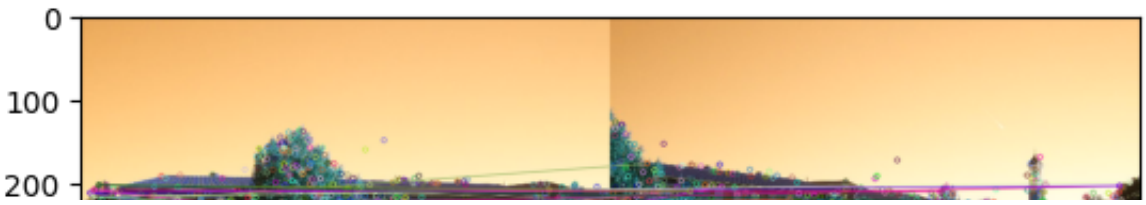
In [ ]:
```
#import libraries
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt
import random
from itertools import groupby, product
from scipy.spatial.transform import Rotation as Ro
from google.colab.patches import cv2_imshow



# Function for extracting the features
def feature_extractor(img1, img2):
  sift = cv.SIFT_create()  # Create a SIFT object
  kp1, ds1 = sift.detectAndCompute(img1, None)
  kp2, ds2 = sift.detectAndCompute(img2, None)
  return kp1, ds1, kp2, ds2 # Return keypoints and descriptors for bot



# Function for matching the features
def feature_matcher_1_2(k1, k2, d1, d2, img1, img2):
  FLANN_INDEX_KDTREE = 1
  index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
  search_params = dict(checks=50)
  flann = cv.FlannBasedMatcher(index_params,search_params) # Create FL
  matches = flann.knnMatch(d1,d2, k =2)
```

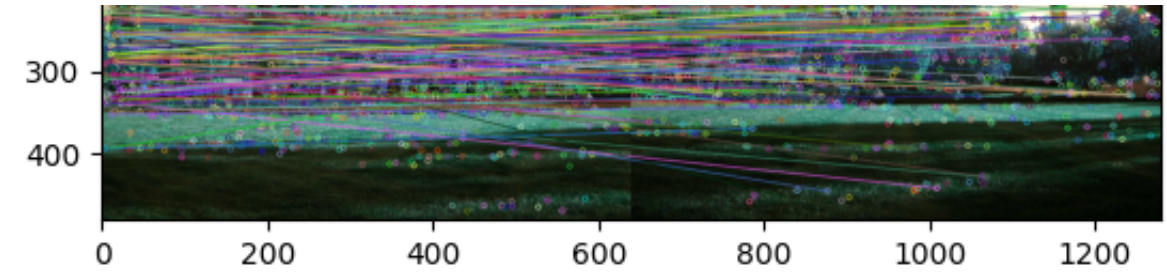
```
  good_matches = []# List to store good matches

  for m, n in matches:
      if m.distance < 0.8 * n.distance:
          good_matches.append(m)

  top_matches = matches[:50]

  img12 = cv.drawMatchesKnn(img1,k1,img2,k2,top_matches,None)
  src_pts = np.float32([k1[m.queryIdx].pt for m in good_matches]).resh
  dst_pts = np.float32([k2[m.trainIdx].pt for m in good_matches]).resh
  klist= []
  for mat in good_matches:

    (x1, y1) = k1[mat.queryIdx].pt
    (x2, y2) = k2[mat.trainIdx].pt

    klist.append([(x1, y1), (x2, y2)])# Append point pairs to the list
  plt.imshow(img12)
  plt.show()
  return top_matches, img12, klist, src_pts, dst_pts


# Function for cropping the black part in the stitched image
def trim_black_part(img):

  if not np.sum(img[0]):
    return trim_black_part(img[1:])
  elif not np.sum(img[-1]):
    return trim_black_part(img[:-1])
  elif not np.sum(img[:,0]):
    return trim_black_part(img[:,1:])
  elif not np.sum(img[:,-1]):
    return trim_black_part(img[:,:-1])

  return img

image_1 = cv.imread("./videos/img1.jpg", cv.IMREAD_COLOR)
image_2 = cv.imread("./videos/img2.jpg", cv.IMREAD_COLOR)
image_3 = cv.imread("./videos/img3.jpg", cv.IMREAD_COLOR)
image_4 = cv.imread("./videos/img4.jpg", cv.IMREAD_COLOR)
# image 3 and 4
# 1) Extract features from each frame (You can use any feature extract
key_3, des3, key_4, des4 = feature_extractor(image_3, image_4)

# 2) Match the features between each consecutive image and visualize t
top, image_34, key_list_34, src_pts, dst_pts = feature_matcher_1_2(key

# 3) Compute the homographies between the pairs of images.
H_34, mask = cv.findHomography(src_pts, dst_pts, cv.RANSAC)
```

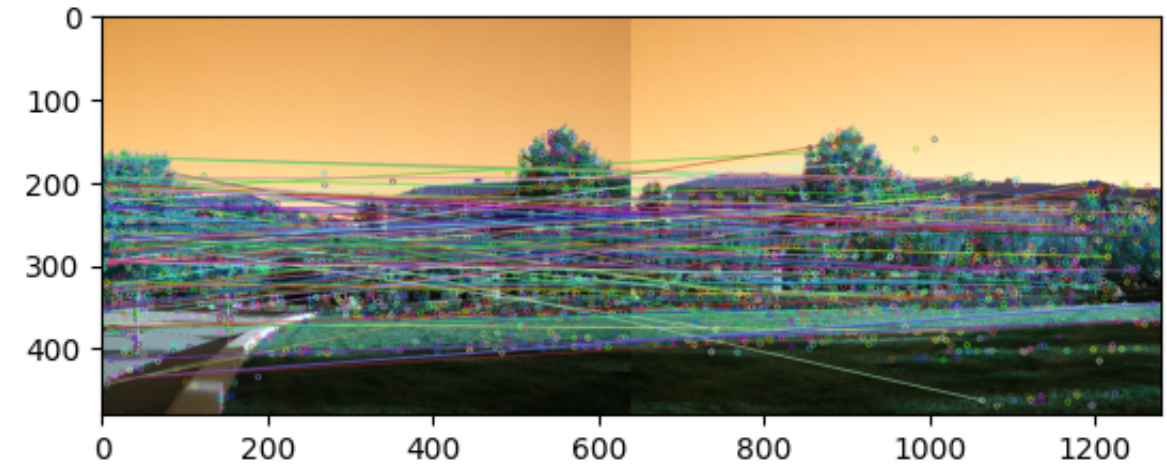
```
# 4) Combine these frames together using the computed homographies.
H_43 = np.linalg.inv(H_34)
width_3 = image_3.shape[1]
height_3 = image_3.shape[0]
width_4 = image_4.shape[1]

result_34 = cv.warpPerspective(image_4, H_43, (width_3 + width_4, heig

result_34[0:image_3.shape[0], 0:image_3.shape[1]] = image_3

result_34 = trim_black_part(result_34)

cv2_imshow(result_34)
#image 2 and 3
# 1) Extract features from each frame (You can use any feature extract
key_2, des2, key_3, des3 = feature_extractor(image_2, image_3)

# 2) Match the features between each consecutive image and visualize t
top, image_23, key_list_23, src_pts, dst_pts = feature_matcher_1_2(key

# 3) Compute the homographies between the pairs of images.
H_23, mask = cv.findHomography(src_pts, dst_pts, cv.RANSAC)

# 4) Combine these frames together using the computed homographies.
H_32 = np.linalg.inv(H_23)
width_2 = image_2.shape[1]
height_2 = image_2.shape[0]
width_3 = result_34.shape[1]

result_23 = cv.warpPerspective(result_34, H_32, (width_2 + width_3, he

result_23[0:image_2.shape[0], 0:image_2.shape[1]] = image_2

result_23 = trim_black_part(result_23)
cv2_imshow(result_23)

#image 1 and 2

# 1) Extract features from each frame (You can use any feature extract

key_1, des1, key_2, des2 = feature_extractor(image_1, image_2)

# 2) Match the features between each consecutive image and visualize t
top, image_12, key_list_12, src_pts, dst_pts = feature_matcher_1_2(key
# 3) Compute the homographies between the pairs of images.
H_12, mask = cv.findHomography(src_pts, dst_pts, cv.RANSAC)

# 4) Combine these frames together using the computed homographies.
H_21 = np.linalg.inv(H_12)
width_1 = image_1.shape[1]
```

```
height_1 = image_1.shape[0]
width_2 = result_23.shape[1]

result_12 = cv.warpPerspective(result_23, H_21, (width_1 + width_2, he

result_12[0:image_1.shape[0], 0:image_1.shape[1]] = image_1

result_12 = trim_black_part(result_12)
cv2_imshow(result_12)

cv.waitKey() & 0xFF == ord("q")
cv.destroyAllWindows()
```






300
400
0
200
400
600
800
1000
1200


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

In [ ]: