Sreevarsha Kurella

CV ASSIGNMENT -2

1)

```
import numpy as np
from scipy import ndimage
import cv2
from PIL import Image
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure
```

[1] ✓ 1.1s

```
def gaussian_kernel(size, sigma=10):
    size = int(size) // 2
    x, y = np.mgrid[-size:size+1, -size:size+1]
    normal = 1 / (2.0 * np.pi * sigma**2)
    g = np.exp(-((x**2 + y**2) / (2.0*sigma**2))) * normal
    return g
```

[2] ✓ 0.4s

```python
def sobel_filters(img):
    Kx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], np.float32)
    Ky = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], np.float32)

    Ix = ndimage.filters.convolve(img, Kx)
    Iy = ndimage.filters.convolve(img, Ky)

    G = np.hypot(Ix, Iy)
    G = G / G.max() * 255
    theta = np.arctan2(Iy, Ix)

    return (G, theta)
```

[3] ✓ 0.3s

```python
def visualize(img,dst):
    plt.subplot(121),plt.imshow(img),plt.title('Original')
    plt.xticks([]), plt.yticks([])
    plt.subplot(122),plt.imshow(dst,cmap="gray"),plt.title('Blurred')
    plt.xticks([]), plt.yticks([])
    plt.show()
```

[4] ✓ 0.3s

```python
def Canny_detector(img):
    weak_th = None
    strong_th = None
    # conversion - image to grayscale
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)


    g=gaussian_kernel(5,5)

    img= cv2.filter2D(src=img, kernel=g, ddepth=19)

    mag,ang=sobel_filters(img)

    # setting extreme threshold

    mag_max = np.max(mag)
    if not weak_th:weak_th = mag_max * 0.1
    if not strong_th:strong_th = mag_max * 0.5

    # finding dimensions of input
    height, width = img.shape


    # img
    for i_x in range(width):
        for i_y in range(height):

            grad_ang = ang[i_y, i_x]
            grad_ang = abs(grad_ang-180) if abs(grad_ang)>180 else abs(grad_ang)
```

```
            neighb_1_x, neighb_1_y = i_x-1, i_y
            neighb_2_x, neighb_2_y = i_x + 1, i_y

            if width>neighb_1_x>= 0 and height>neighb_1_y>= 0:
                if mag[i_y, i_x]<mag[neighb_1_y, neighb_1_x]:
                    mag[i_y, i_x]= 0
                    continue

            if width>neighb_2_x>= 0 and height>neighb_2_y>= 0:
                if mag[i_y, i_x]<mag[neighb_2_y, neighb_2_x]:
                    mag[i_y, i_x]= 0

    weak_ids = np.zeros_like(img)
    strong_ids = np.zeros_like(img)
    ids = np.zeros_like(img)
    for i_x in range(width):
        for i_y in range(height):

            grad_mag = mag[i_y, i_x]

            if grad_mag<weak_th:
                mag[i_y, i_x]= 0
            elif strong_th>grad_mag>= weak_th:
                ids[i_y, i_x]= 1
            else:
                ids[i_y, i_x]= 2


    #  returning the magnitude of
    # gradients
    return mag
```

```
frame = cv2.imread('sample.jpeg')
canny_img = Canny_detector(frame)

plt.imshow(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))
```

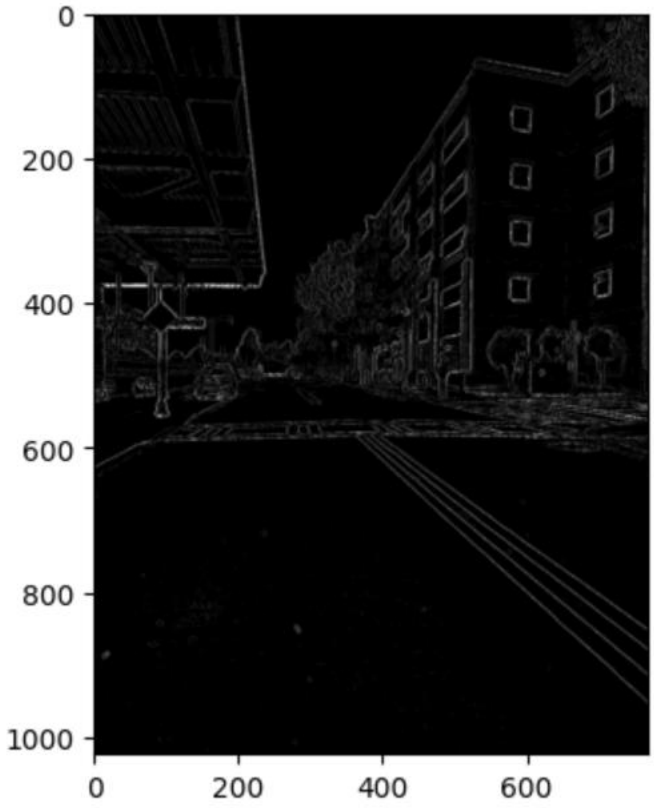
✓ 5.0s


0
200
400
600
800
1000
0
200
400
600

```
plt.imshow(canny_img,cmap='gray')
```

✓ 0.2s

```
<matplotlib.image.AxesImage at 0x2139017d630>
```

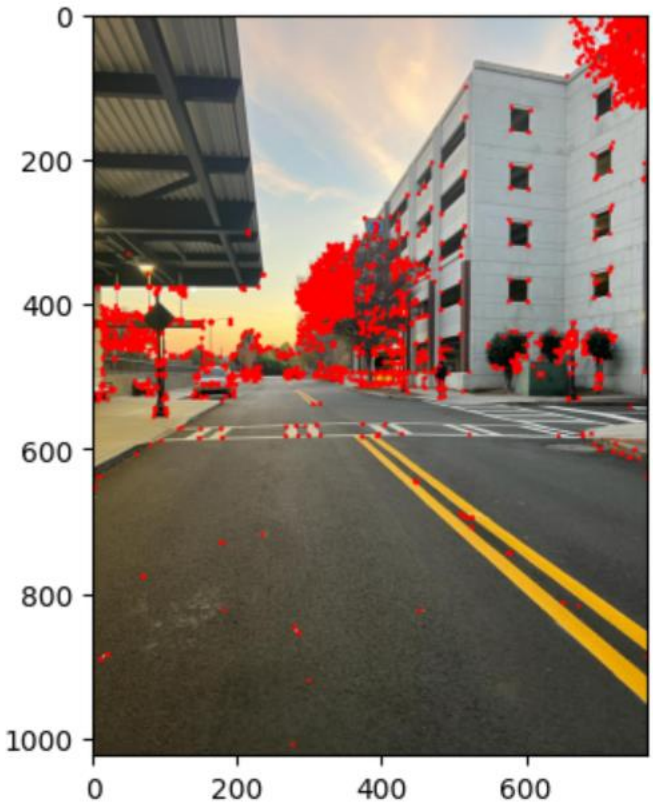


Harris Edge Detection

```
image = frame
operatedImage = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
operatedImage = np.float32(operatedImage)
dest = cv2.cornerHarris(operatedImage, 2, 3, 0.07)
dest = cv2.dilate(dest,gaussian_kernel(5,5))
image[dest > 0.01 * dest.max()]=[0, 0, 255]
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
if cv2.waitKey(0) & 0xff == 27:
    cv2.destroyAllWindows()
image
```

✓ 0.3s

```
Output exceeds the size limit. Open the full output data in a text editor
array([[[ 83, 106, 128],
        [ 84, 107, 129],
        [ 85, 110, 130],
        ...,
        [ 27,  49,  55],
        [ 29,  53,  59],
        [ 48,  72,  78]],

       [[ 83, 106, 128],
        [ 84, 107, 129],
        [ 84, 109, 129],
        ...,
        [ 47,  69,  75],
        [ 58,  82,  88],
        [ 73,  97, 103]],

       [[ 83, 104, 125],
        [ 83, 107, 127],
        [ 83, 108, 128],
        ...,
        [ 60,  82,  88],
        [ 64,  86,  92],
        [ 60,  82,  88]],

       ...,
...
```



```
canny_img
```

✓ 0.4s

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

---

3)Integral Image

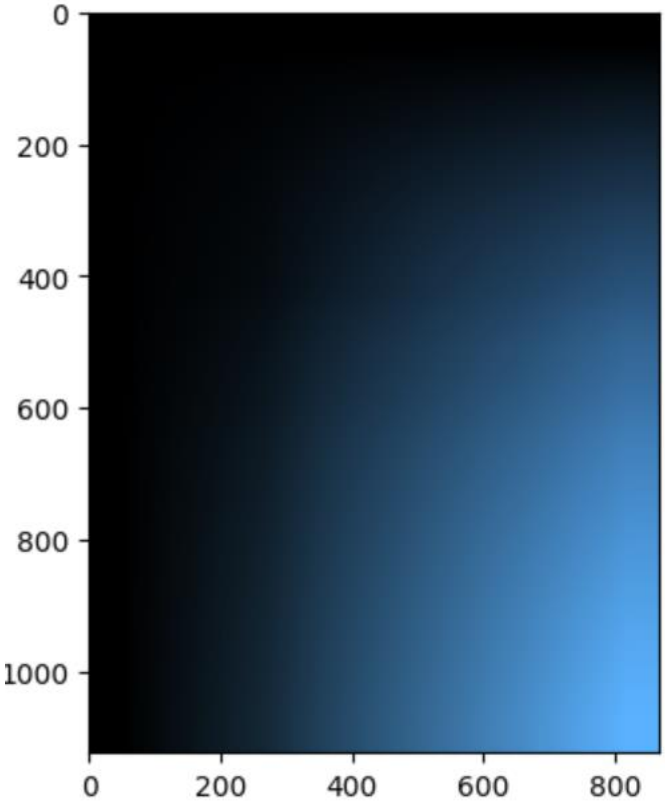
```python
def integral_image(image, *, dtype=None):
    if dtype is None and image.real.dtype.kind == 'f':
        dtype = np.promote_types(image.dtype, np.float64)

    S = image
    for i in range(image.ndim):
        S = S.cumsum(axis=i, dtype=dtype)
    return S


def integrate(ii, start, end):
    start = np.atleast_2d(np.array(start))
    end = np.atleast_2d(np.array(end))
    rows = start.shape[0]

    total_shape = ii.shape
    total_shape = np.tile(total_shape, [rows, 1])

    start_negatives = start < 0
    end_negatives = end < 0
    start = (start + total_shape) * start_negatives + \
            start * ~(start_negatives)
    end = (end + total_shape) * end_negatives + \
          end * ~(end_negatives)

    if np.any((end - start) < 0):
        raise IndexError('end coordinates must be greater or equal to start')
```

```
    S = np.zeros(rows)
    bit_perm = 2 ** ii.ndim
    width = len(bin(bit_perm - 1)[2:])
    for i in range(bit_perm):
        binary = bin(i)[2:].zfill(width)
        bool_mask = [bit == '1' for bit in binary]

        sign = (-1)**sum(bool_mask)

        bad = [np.any(((start[r] - 1) * bool_mask) < 0)
               for r in range(rows)]

        corner_points = (end * (np.invert(bool_mask))) + \
                        ((start - 1) * bool_mask)

        S += [sign * ii[tuple(corner_points[r])] if(not bad[r]) else 0
              for r in range(rows)]
    return S
```

```
frame = cv2.imread('sample.jpeg')
frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
frame= cv2.copyMakeBorder(frame, 50, 50, 50, 50, cv2.BORDER_CONSTANT, (0,0,0))
frame=integral_image(frame)
frame = frame/np.amax(frame)
frame = np.clip(frame, 0,200)
plt.imshow(frame)
```

✓ 0.4s

Integral Image



2)Stitching

```
class Image_Stitching():
    def __init__(self) :
        self.ratio=0.85
        self.min_match=10
        self.sift=cv2.SIFT_create()
        self.smoothing_window_size=800

    def registration(self,img1,img2):
        kp1, des1 = self.sift.detectAndCompute(img1, None)
        kp2, des2 = self.sift.detectAndCompute(img2, None)
        matcher = cv2.BFMatcher()
        raw_matches = matcher.knnMatch(des1, des2, k=2)
        good_points = []
        good_matches=[]
        for m1, m2 in raw_matches:
            if m1.distance < self.ratio * m2.distance:
                good_points.append((m1.trainIdx, m1.queryIdx))
                good_matches.append([m1])
        img3 = cv2.drawMatchesKnn(img1, kp1, img2, kp2, good_matches, None, flags=2)
        cv2.imwrite('matching.jpg', img3)
        if len(good_points) > self.min_match:
            image1_kp = np.float32(
                [kp1[i].pt for (_, i) in good_points])
            image2_kp = np.float32(
                [kp2[i].pt for (i, _) in good_points])
            H, status = cv2.findHomography(image2_kp, image1_kp, cv2.RANSAC,5.0)
        return H
```

```python
def blending(self,img1,img2):
    H = self.registration(img1,img2)
    height_img1 = img1.shape[0]
    width_img1 = img1.shape[1]
    width_img2 = img2.shape[1]
    height_panorama = height_img1
    width_panorama = width_img1 +width_img2

    panorama1 = np.zeros((height_panorama, width_panorama, 3))
    mask1 = self.create_mask(img1,img2,version='left_image')
    panorama1[0:img1.shape[0], 0:img1.shape[1], :] = img1
    panorama1 *= mask1
    mask2 = self.create_mask(img1,img2,version='right_image')
    panorama2 = cv2.warpPerspective(img2, H, (width_panorama, height_panorama))*mask2
    result=panorama1+panorama2

    rows, cols = np.where(result[:, :, 0] != 0)
    min_row, max_row = min(rows), max(rows) + 1
    min_col, max_col = min(cols), max(cols) + 1
    final_result = result[min_row:max_row, min_col:max_col, :]
    return final_result
```

Building 1 Stitching-

```
d='b1'
img1=cv2.cvtColor(cv2.imread(d+'/1.jpg'), cv2.COLOR_BGR2RGB)
img2=cv2.cvtColor(cv2.imread(d+'/2.jpg'), cv2.COLOR_BGR2RGB)
img3=cv2.cvtColor(cv2.imread(d+'/3.jpg'), cv2.COLOR_BGR2RGB)
```

✓ 0.5s

```
stitcher = cv2.Stitcher_create()
(status, stitched) = stitcher.stitch([img1,img2,img3])
print(status)
```

✓ 1.6s

```
plt.imshow(stitched)
```

✓ 2.4s


0
500
1000
1500
2000
2500
3000
3500
0
500
1000
1500
2000
2500
3000
3500
4000

Building 2 stitching

```
d='b2'
img1=cv2.cvtColor(cv2.imread(d+'/1.jpg'), cv2.COLOR_BGR2RGB)
img2=cv2.cvtColor(cv2.imread(d+'/2.jpg'), cv2.COLOR_BGR2RGB)
img3=cv2.cvtColor(cv2.imread(d+'/3.jpg'), cv2.COLOR_BGR2RGB)
```

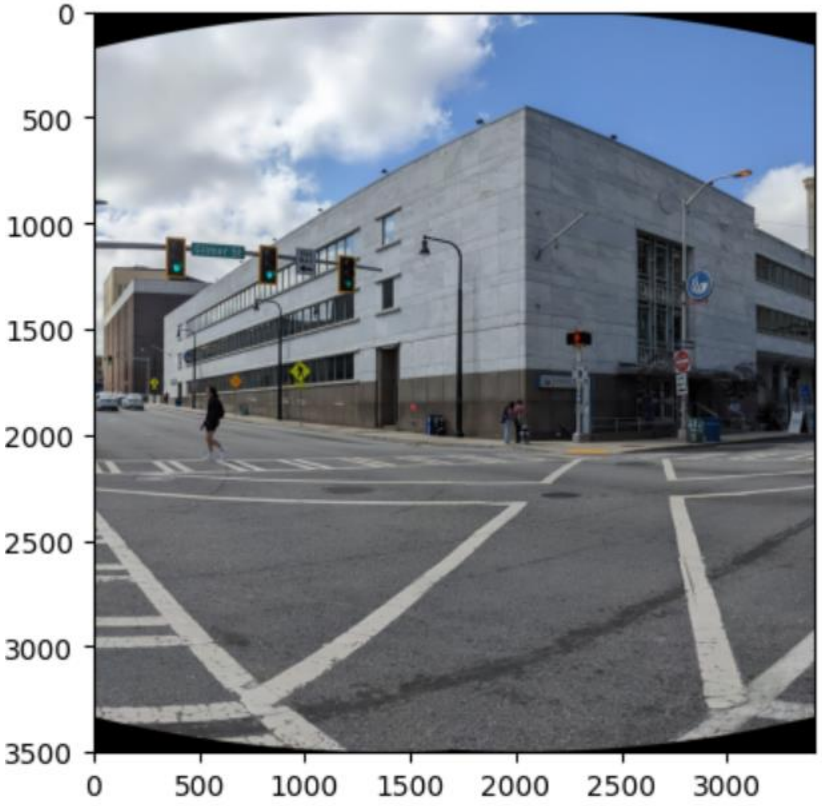
✓ 0.4s

```
stitcher = cv2.Stitcher_create()
(status, stitched) = stitcher.stitch([img1,img2,img3])
print(status)
```

✓ 0.8s

```
plt.imshow(stitched)
```

✓ 1.5s


0
500
1000
1500
2000
2500
3000
3500
0
500
1000
1500
2000
2500
3000

Building 3 stitching

```
d='b3'
img1=cv2.cvtColor(cv2.imread(d+'/1.jpeg'), cv2.COLOR_BGR2RGB)
img2=cv2.cvtColor(cv2.imread(d+'/2.jpeg'), cv2.COLOR_BGR2RGB)
img3=cv2.cvtColor(cv2.imread(d+'/3.jpeg'), cv2.COLOR_BGR2RGB)
```

✓ 0.8s

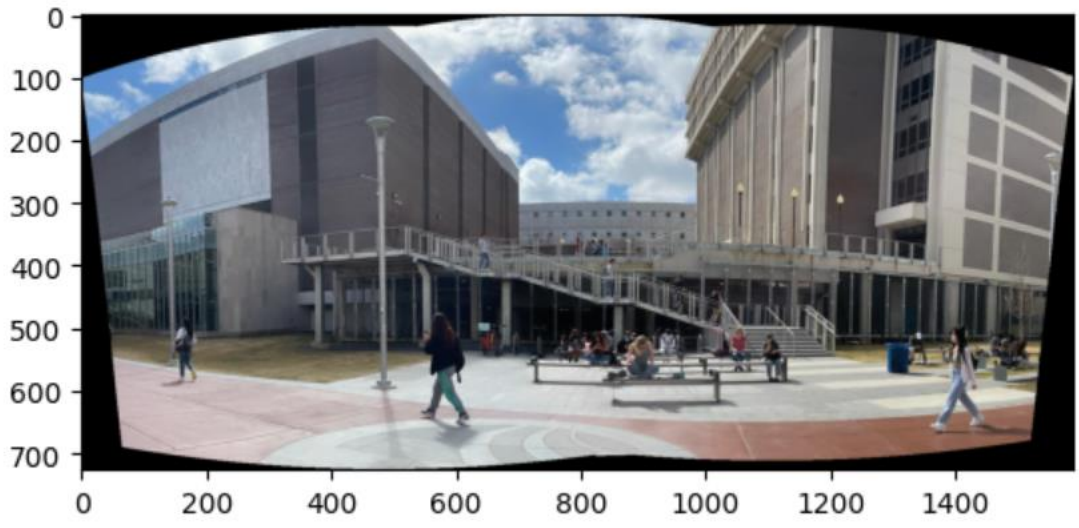
```
stitcher = cv2.Stitcher_create()
(status, stitched) = stitcher.stitch([img1,img2,img3])
print(status)
```

✓ 0.4s

```
0
```

```
plt.imshow(stitched)
```

✓ 0.3s



Building 4 stitching

```
d='b4'
img1=cv2.cvtColor(cv2.imread(d+'/1.jpeg'), cv2.COLOR_BGR2RGB)
img2=cv2.cvtColor(cv2.imread(d+'/2.jpeg'), cv2.COLOR_BGR2RGB)
img3=cv2.cvtColor(cv2.imread(d+'/3.jpeg'), cv2.COLOR_BGR2RGB)
```

✓ 0.8s

```
stitcher = cv2.Stitcher_create()
(status, stitched) = stitcher.stitch([img1,img2,img3])
print(status)
```

✓ 0.5s

ə

```
plt.imshow(stitched)
```

✓ 0.3s


0
200
400
600
800
0
200
400
600
800
1000
RIALTO

Building 5 Stitching

```
d='b5'
img1=cv2.cvtColor(cv2.imread(d+'/1.jpg'), cv2.COLOR_BGR2RGB)
img2=cv2.cvtColor(cv2.imread(d+'/2.jpg'), cv2.COLOR_BGR2RGB)
img3=cv2.cvtColor(cv2.imread(d+'/3.jpg'), cv2.COLOR_BGR2RGB)
```

✓ 0.4s

```
stitcher = cv2.Stitcher_create()
(status, stitched) = stitcher.stitch([img1,img2,img3])
print(status)
```

✓ 1.2s

ə

```
plt.imshow(stitched)
```

✓ 2.7s


0
500
1000
1500
2000
2500
3000
3500
0
1000
2000
3000
4000
5000
STATE PLAYS HERE

---

4)

```python
import cv2
import depthai as dai
import time
from depthai_sdk.fps import FPSHandler
# Create pipeline
pipeline = dai.Pipeline()

# Define source and output
camRgb = pipeline.create(dai.node.ColorCamera)
xoutVideo = pipeline.create(dai.node.XLinkOut)

xoutVideo.setStreamName("video")

# Properties
camRgb.setBoardSocket(dai.CameraBoardSocket.RGB)
camRgb.setResolution(dai.ColorCameraProperties.SensorResolution.THE_1080_P)
camRgb.setVideoSize(1080,720)

xoutVideo.input.setBlocking(False)
xoutVideo.input.setQueueSize(1)

# Linking
camRgb.video.link(xoutVideo.input)
# Connect to device and start pipeline
start_time = time.time()
x = 1
counter = 0
count=0
imagesf=[]

with dai.Device(pipeline) as device:

```

```
    video = device.getOutputQueue(name="video", maxSize=1, blocking=False)

    while True:
        videoIn = video.get()
        Frame=videoIn.getCvFrame()
        counter+=1
        cv2.imshow("video", Frame)
        if cv2.waitKey(1) == ord('i'):
            imagesf.append(Frame)
            counter+=1

        if cv2.waitKey(1) == ord('p'):
            print('Images have been stitched to make a panaroma')

            if counter < 2:
                print('Not enough pictures to create a panaroma')
            else:
                stitcher=cv2.Stitcher.create()
                image,panaromaview =stitcher.stitch(imagesf)
                if image != cv2.STITCHER_OK:
                    print("could not stitch the images to create a panaroma")
                else:
                    print('Images stitched. Yayyyy Panaroma.')
                    cv2.imshow('Panaroma of the images clicked',panaromaview)
                    cv2.imwrite('panaroma.jpg', panaromaview)

        if cv2.waitKey(1) == ord('q'):
            break
```

---

5)

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt
import imageio
import imutils
cv2.ocl.setUseOpenCL(False)
def Main_Ponts_Func(image):
    descriptor = cv2.ORB_create()
    kps, features = descriptor.detectAndCompute(image, None)
    return (kps, features)
def Match_The_Key_Points_Func(features_train, features_query, ratio):
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck = False)
    raw_match = bf.knnMatch(features_train, features_query, 2)
    matches = []
    for m, n in raw_match:
        if m.distance < n.distance * ratio:
            matches.append(m)
    return matches
def Find_Homography(kps_train, kps_query, matches, reprojThresh):
    kpsA = np.float32([kp.pt for kp in kps_train])
    kpsB = np.float32([kp.pt for kp in kps_query])
    if len(matches) > 4:
        ptsA = np.float32([kpsA[m.queryIdx] for m in matches])
        ptsB = np.float32([kpsB[m.trainIdx] for m in matches])
        (Homography, status) = cv2.findHomography(ptsA, ptsB, cv2.RANSAC, reprojThresh)
        return(matches, Homography, status)
    else:
        return None
def Transform_To_Gray_Scale_Func(result):
    gray = cv2.cvtColor(result, cv2.COLOR_BGR2GRAY)
    thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY)[1]
    # To retrieve the contours in the binary image
    cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

```
    # TO get the maximum contour area
    c = max(cnts, key=cv2.contourArea)
    (x, y, w, h) = cv2.boundingRect(c)
    result = result[y:y + h, x:x + w]
    return result
#Read img4 and img3 which are to be stiched
#training Img
image_to_train = imageio.imread(r"C:\Users\sreev\OneDrive\Documents\Computer Vision\ASSIGNMENT_2\b4\1.jpeg")
#queried Img
image_to_be_queried = imageio.imread(r"C:\Users\sreev\OneDrive\Documents\Computer Vision\ASSIGNMENT_2\b4\2.jpeg")
gray_image_to_train = cv2.cvtColor(image_to_train, cv2.COLOR_RGB2GRAY)
gray_image_to_be_queried = cv2.cvtColor(image_to_be_queried, cv2.COLOR_RGB2GRAY)
kps_train, features_train = Main_Ponts_Func(gray_image_to_train) #kps and features of 1
kps_query, features_query = Main_Ponts_Func(gray_image_to_be_queried) #kps and features of 2
matches = Match_The_Key_Points_Func(features_train, features_query, 0.75)
temp_img = cv2.drawMatches(image_to_train, kps_train, image_to_be_queried, kps_query, np.random.choice(matches,100), None, flags=cv2.DrawMa
M = Find_Homography(kps_train, kps_query, matches, 4)
matches, homography, status = M
width = image_to_train.shape[1] + image_to_be_queried.shape[1]
height = image_to_train.shape[0] + image_to_be_queried.shape[0]
result = cv2.warpPerspective(image_to_train, homography, (width, height))
result[0:image_to_be_queried.shape[0], 0:image_to_be_queried.shape[1]] = image_to_be_queried
result = Transform_To_Gray_Scale_Func(result)
plt.figure(figsize=(20,10))
plt.imshow(result)
plt.show()

```