```
import numpy as np
import cv2
import scipy.io
import os
from numpy.linalg import norm
from matplotlib import pyplot as plt
from numpy.linalg import det
from numpy.linalg import inv
from scipy.linalg import rq
from numpy.linalg import svd
import matplotlib.pyplot as plt
import numpy as np
import math
import random
import sys
from scipy import ndimage, spatial


from google.colab import drive

# This will prompt for authorization.
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
plt.figure(figsize=(20,10))
```

```
<Figure size 1440x720 with 0 Axes>
<Figure size 1440x720 with 0 Axes>
```

```
class Image:
    def __init__(self, img, position):

        self.img = img
        self.position = position

inlier_matchset = []
def features_matching(a,keypointlength,threshold):
  #threshold=0.2
  bestmatch=np.empty((keypointlength),dtype= np.int16)
  img1index=np.empty((keypointlength),dtype=np.int16)
  distance=np.empty((keypointlength))
  index=0
  for j in range(0,keypointlength):
    #For a descriptor fa in Ia, take the two closest descriptors fb1 and fb2 in Ib
    x=a[j]
    listx=x.tolist()
    x.sort()
    minval1=x[0]                                    # min
    minval2=x[1]                                    # 2nd min
```

```
      minval2=x[1]                              # 2nd min
      itemindex1 = listx.index(minval1)         #index of min val
      itemindex2 = listx.index(minval2)         #index of second min value
      ratio=minval1/minval2                     #Ratio Test

      if ratio<threshold:
        #Low distance ratio: fb1 can be a good match
        bestmatch[index]=itemindex1
        distance[index]=minval1
        img1index[index]=j
        index=index+1
    return  [cv2.DMatch(img1index[i],bestmatch[i].astype(int),distance[i]) for i in range(0,ind


  def compute_Homography(im1_pts,im2_pts):
    """
    im1_pts and im2_pts are 2×n matrices with
    4 point correspondences from the two images
    """
    num_matches=len(im1_pts)
    num_rows = 2 * num_matches
    num_cols = 9
    A_matrix_shape = (num_rows,num_cols)
    A = np.zeros(A_matrix_shape)
    a_index = 0
    for i in range(0,num_matches):
      (a_x, a_y) = im1_pts[i]
      (b_x, b_y) = im2_pts[i]
      row1 = [a_x, a_y, 1, 0, 0, 0, -b_x*a_x, -b_x*a_y, -b_x] # First row
      row2 = [0, 0, 0, a_x, a_y, 1, -b_y*a_x, -b_y*a_y, -b_y] # Second row

      # place the rows in the matrix
      A[a_index] = row1
      A[a_index+1] = row2

      a_index += 2

    U, s, Vt = np.linalg.svd(A)

    #s is a 1-D array of singular values sorted in descending order
    #U, Vt are unitary matrices
    #Rows of Vt are the eigenvectors of A^TA.
    #Columns of U are the eigenvectors of AA^T.
    H = np.eye(3)
    H = Vt[-1].reshape(3,3) # take the last row of the Vt matrix
    return H


  def displayplot(img,title):

    plt.figure(figsize=(15,15))
    plt.title(title)
    plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
```

```
    plt.show()


def RANSAC_alg(f1, f2, matches, nRANSAC, RANSACthresh):


    minMatches = 4
    nBest = 0
    best_inliers = []
    H_estimate = np.eye(3,3)
    global inlier_matchset
    inlier_matchset=[]
    for iteration in range(nRANSAC):

        #Choose a minimal set of feature matches.
        matchSample = random.sample(matches, minMatches)

        #Estimate the Homography implied by these matches
        im1_pts=np.empty((minMatches,2))
        im2_pts=np.empty((minMatches,2))
        for i in range(0,minMatches):
          m = matchSample[i]
          im1_pts[i] = f1[m.queryIdx].pt
          im2_pts[i] = f2[m.trainIdx].pt
          #im1_pts[i] = f1[m[0]].pt
          #im2_pts[i] = f2[m[1]].pt

        H_estimate=compute_Homography(im1_pts,im2_pts)


        # Calculate the inliers for the H
        inliers = get_inliers(f1, f2, matches, H_estimate, RANSACthresh)

        # if the number of inliers is higher than previous iterations, update the best estima
        if len(inliers) > nBest:
            nBest= len(inliers)
            best_inliers = inliers

    print("Number of best inliers",len(best_inliers))
    for i in range(len(best_inliers)):
      inlier_matchset.append(matches[best_inliers[i]])

    # compute a homography given this set of matches
    im1_pts=np.empty((len(best_inliers),2))
    im2_pts=np.empty((len(best_inliers),2))
    for i in range(0,len(best_inliers)):
      m = inlier_matchset[i]
      im1_pts[i] = f1[m.queryIdx].pt
      im2_pts[i] = f2[m.trainIdx].pt
      #im1_pts[i] = f1[m[0]].pt
      #im2_pts[i] = f2[m[1]].pt
```

```
    M=compute_Homography(im1_pts,im2_pts)
    return M


def get_inliers(f1, f2, matches, H, RANSACthresh):

  inlier_indices = []
  for i in range(len(matches)):
    queryInd = matches[i].queryIdx
    trainInd = matches[i].trainIdx

    #queryInd = matches[i][0]
    #trainInd = matches[i][1]

    queryPoint = np.array([f1[queryInd].pt[0],  f1[queryInd].pt[1], 1]).T
    trans_query = H.dot(queryPoint)


    comp1 = [trans_query[0]/trans_query[2], trans_query[1]/trans_query[2]] # normalize with r
    comp2 = np.array(f2[trainInd].pt)[:2]


    if(np.linalg.norm(comp1-comp2) <= RANSACthresh): # check against threshold
      inlier_indices.append(i)
  return inlier_indices



def ImageBounds(img, H):

    h, w= img.shape[0], img.shape[1]
    p1 = np.dot(H, np.array([0, 0, 1]))
    p2 = np.dot(H, np.array([0, h - 1, 1]))
    p3 = np.dot(H, np.array([w - 1, 0, 1]))
    p4 = np.dot(H, np.array([w - 1, h - 1, 1]))
    x1 = p1[0] / p1[2]
    y1 = p1[1] / p1[2]
    x2 = p2[0] / p2[2]
    y2 = p2[1] / p2[2]
    x3 = p3[0] / p3[2]
    y3 = p3[1] / p3[2]
    x4 = p4[0] / p4[2]
    y4 = p4[1] / p4[2]
    minX = math.ceil(min(x1, x2, x3, x4))
    minY = math.ceil(min(y1, y2, y3, y4))
    maxX = math.ceil(max(x1, x2, x3, x4))
    maxY = math.ceil(max(y1, y2, y3, y4))

    return int(minX), int(minY), int(maxX), int(maxY)


def Populate_Images(img, accumulator, H, bw):
```

```
    h, w = img.shape[0], img.shape[1]
    minX, minY, maxX, maxY = ImageBounds(img, H)

    for i in range(minX, maxX + 1):
        for j in range(minY, maxY + 1):
            p = np.dot(np.linalg.inv(H), np.array([i, j, 1]))

            x = p[0]
            y = p[1]
            z = p[2]

            _x = int(x / z)
            _y = int(y / z)

            if _x < 0 or _x >= w - 1 or _y < 0 or _y >= h - 1:
                continue

            if img[_y, _x, 0] == 0 and img[_y, _x, 1] == 0 and img[_y, _x, 2] == 0:
                continue

            wt = 1.0

            if _x >= minX and _x < minX + bw:
                wt = float(_x - minX) /bw
            if _x <= maxX and _x > maxX -bw:
                wt = float(maxX - _x) /bw

            accumulator[j, i, 3] += wt

            for c in range(3):
                accumulator[j, i, c] += img[_y, _x, c] *wt


def Image_Stitch(Imagesall, blendWidth, accWidth, accHeight, translation):
    channels=3
    #width=720

    acc = np.zeros((accHeight, accWidth, channels + 1))
    M = np.identity(3)
    for count, i in enumerate(Imagesall):
        M = i.position
        img = i.img
        M_trans = translation.dot(M)
        Populate_Images(img, acc, M_trans, blendWidth)

    height, width = acc.shape[0], acc.shape[1]

    img = np.zeros((height, width, 3))
    for i in range(height):
```

```
            for j in range(width):
                weights = acc[i, j, 3]
                if weights > 0:
                    for c in range(3):
                        img[i, j, c] = int(acc[i, j, c] / weights)


        Imagefull = np.uint8(img)
        M = np.identity(3)
        for count, i in enumerate(Imagesall):
            if count != 0 and count != (len(Imagesall) - 1):
                continue

            M = i.position

            M_trans = translation.dot(M)

            p = np.array([0.5 * width, 0, 1])
            p = M_trans.dot(p)


            if count == 0:
                x_init, y_init = p[:2] / p[2]

            if count == (len(Imagesall) - 1):
                x_final, y_final = p[:2] / p[2]


        A = np.identity(3)
        croppedImage = cv2.warpPerspective(
            Imagefull, A, (accWidth, accHeight), flags=cv2.INTER_LINEAR
        )
        displayplot(croppedImage, 'Final Stitched Image')


    '''
    left_img_pth4 = '/content/drive/MyDrive/RGB-img/img/IX-11-01917_0004_0004.JPG'
    left_img_pth3 = '/content/drive/MyDrive/RGB-img/img/IX-11-01917_0004_0005.JPG'
    left_img_pth2 = '/content/drive/MyDrive/RGB-img/img/IX-11-01917_0004_0006.JPG'
    left_img_pth1 = '/content/drive/MyDrive/RGB-img/img/IX-11-01917_0004_0007.JPG'
    left_img_pth = '/content/drive/MyDrive/RGB-img/img/IX-11-01917_0004_0008.JPG'
    mid_img_pth = '/content/drive/MyDrive/RGB-img/img/IX-11-01917_0004_0009.JPG'
    right_img_pth = '/content/drive/MyDrive/RGB-img/img/IX-11-01917_0004_0010.JPG'
    right_img_pth1 = '/content/drive/MyDrive/RGB-img/img/IX-11-01917_0004_0011.JPG'
    right_img_pth2 = '/content/drive/MyDrive/RGB-img/img/IX-11-01917_0004_0012.JPG'
    '''

    left_img_pth4 = '/content/drive/MyDrive/RGB-img/img/IX-11-01917_0004_0013.JPG'
    left_img_pth3 = '/content/drive/MyDrive/RGB-img/img/IX-11-01917_0004_0014.JPG'
    left_img_pth2 = '/content/drive/MyDrive/RGB-img/img/IX-11-01917_0004_0015.JPG'
    left_img_pth1 = '/content/drive/MyDrive/RGB-img/img/IX-11-01917_0004_0016.JPG'
    left_img_pth = '/content/drive/MyDrive/RGB-img/img/IX-11-01917_0004_0017.JPG'
```

```
mid_img_pth = '/content/drive/MyDrive/RGB-img/img/IX-11-01917_0004_0018.JPG'
right_img_pth = '/content/drive/MyDrive/RGB-img/img/IX-11-01917_0004_0019.JPG'
right_img_pth1 = '/content/drive/MyDrive/RGB-img/img/IX-11-01917_0004_0020.JPG'
right_img_pth2 = '/content/drive/MyDrive/RGB-img/img/IX-11-01917_0004_0021.JPG'


left_image_sat4= cv2.imread(left_img_pth4)
left_image_sat3= cv2.imread(left_img_pth3)
left_image_sat2= cv2.imread(left_img_pth2)
left_image_sat1= cv2.imread(left_img_pth1)
left_image_sat= cv2.imread(left_img_pth)
centre_image_sat = cv2.imread(mid_img_pth)
right_image_sat = cv2.imread(right_img_pth)
right_image_sat1= cv2.imread(right_img_pth1)
right_image_sat2= cv2.imread(right_img_pth2)


left4 = cv2.resize(left_image_sat4,None,fx=0.75, fy=0.75, interpolation = cv2.INTER_CUBIC)
left3 = cv2.resize(left_image_sat3,None,fx=0.75, fy=0.75, interpolation = cv2.INTER_CUBIC)
left2 = cv2.resize(left_image_sat2,None,fx=0.75, fy=0.75, interpolation = cv2.INTER_CUBIC)
left1 = cv2.resize(left_image_sat1,None,fx=0.75, fy=0.75, interpolation = cv2.INTER_CUBIC)
left = cv2.resize(left_image_sat,None,fx=0.75, fy=0.75, interpolation = cv2.INTER_CUBIC)
centre = cv2.resize(centre_image_sat,None,fx=0.75, fy=0.75, interpolation = cv2.INTER_CUBIC)
right = cv2.resize(right_image_sat,None,fx=0.75, fy=0.75, interpolation = cv2.INTER_CUBIC)
right1 = cv2.resize(right_image_sat1,None,fx=0.75, fy=0.75, interpolation = cv2.INTER_CUBIC)
right2 = cv2.resize(right_image_sat2,None,fx=0.75, fy=0.75, interpolation = cv2.INTER_CUBIC)


#left = cv2.cvtColor(left, cv2.COLOR_BGR2GRAY)
#centre = cv2.cvtColor(centre, cv2.COLOR_BGR2GRAY)
#right = cv2.cvtColor(right, cv2.COLOR_BGR2GRAY)



#brisk = cv2.ORB_create(nfeatures = 10000)

#print(ok)

#brisk = cv2.KAZE_create()
Threshl=60;
Octaves=6;
#PatternScales=1.0f;
brisk = cv2.BRISK_create(Threshl,Octaves)

#brisk = cv2.SIFT_create()

# find the keypoints with ORB
harris_corners_left_kp = brisk.detect(left,None)
# compute the descriptors with ORB
lpkey, lf = brisk.compute(left, harris_corners_left_kp)

# find the keypoints with ORB
```

```
harris_corners_left_kp1 = brisk.detect(left1,None)
# compute the descriptors with ORB
lpkey1, lf1 = brisk.compute(left1, harris_corners_left_kp1)

# find the keypoints with ORB
harris_corners_left_kp2 = brisk.detect(left2,None)
# compute the descriptors with ORB
lpkey2, lf2 = brisk.compute(left2, harris_corners_left_kp2)

# find the keypoints with ORB
harris_corners_left_kp3 = brisk.detect(left3,None)
# compute the descriptors with ORB
lpkey3, lf3 = brisk.compute(left3, harris_corners_left_kp3)

# find the keypoints with ORB
harris_corners_left_kp4 = brisk.detect(left4,None)
# compute the descriptors with ORB
lpkey4, lf4 = brisk.compute(left4, harris_corners_left_kp4)

#print(ok)
# find the keypoints with ORB
harris_corners_center_kp = brisk.detect(centre,None)
# compute the descriptors with ORB
cpkey, cf = brisk.compute(centre, harris_corners_center_kp)

# find the keypoints with ORB
harris_corners_right_kp = brisk.detect(right,None)
# compute the descriptors with ORB
rpkey, rf = brisk.compute(right, harris_corners_right_kp)

# find the keypoints with ORB
harris_corners_right_kp1 = brisk.detect(right1,None)
# compute the descriptors with ORB
rpkey1, rf1 = brisk.compute(right, harris_corners_right_kp1)

# find the keypoints with ORB
harris_corners_right_kp2 = brisk.detect(right2,None)
# compute the descriptors with ORB
rpkey2, rf2 = brisk.compute(right, harris_corners_right_kp2)

lp = np.asarray([[p.pt[0], p.pt[1]] for p in lpkey])
lp1 = np.asarray([[p.pt[0], p.pt[1]] for p in lpkey1])
lp2 = np.asarray([[p.pt[0], p.pt[1]] for p in lpkey2])
lp3 = np.asarray([[p.pt[0], p.pt[1]] for p in lpkey3])
lp4 = np.asarray([[p.pt[0], p.pt[1]] for p in lpkey4])

cp = np.asarray([[p.pt[0], p.pt[1]] for p in cpkey])
rp = np.asarray([[p.pt[0], p.pt[1]] for p in rpkey])
rp1 = np.asarray([[p.pt[0], p.pt[1]] for p in rpkey1])
rp2 = np.asarray([[p.pt[0], p.pt[1]] for p in rpkey2])
```

```
#Display the keypoints
im_with_keypoints = cv2.drawKeypoints(right, rpkey, np.array([]), (0,0,255))
displayplot(im_with_keypoints,'Right Image with keypoints')
```


```
#Display the keypoints
im_with_keypoints = cv2.drawKeypoints(centre, cpkey, np.array([]), (0,0,255))
displayplot(im_with_keypoints,'Reference Image with keypoints')
```


```
FLANN_INDEX_KDTREE = 0
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)
flann = cv2.FlannBasedMatcher(index_params, search_params)


rf = np.float32(rf)
cf = np.float32(cf)


matches_r_c = flann.knnMatch(rf, cf, k=2)


matches_1 = []
ratio = 0.7
# loop over the raw matches
for m in matches_r_c:
  # ensure the distance is within a certain ratio of each
  # other (i.e. Lowe’s ratio test)
  if len(m) == 2 and m[0].distance < m[1].distance * ratio:
      #matches_1.append((m[0].trainIdx, m[0].queryIdx))
      matches_1.append(m[0])


print(len(matches_1))
```

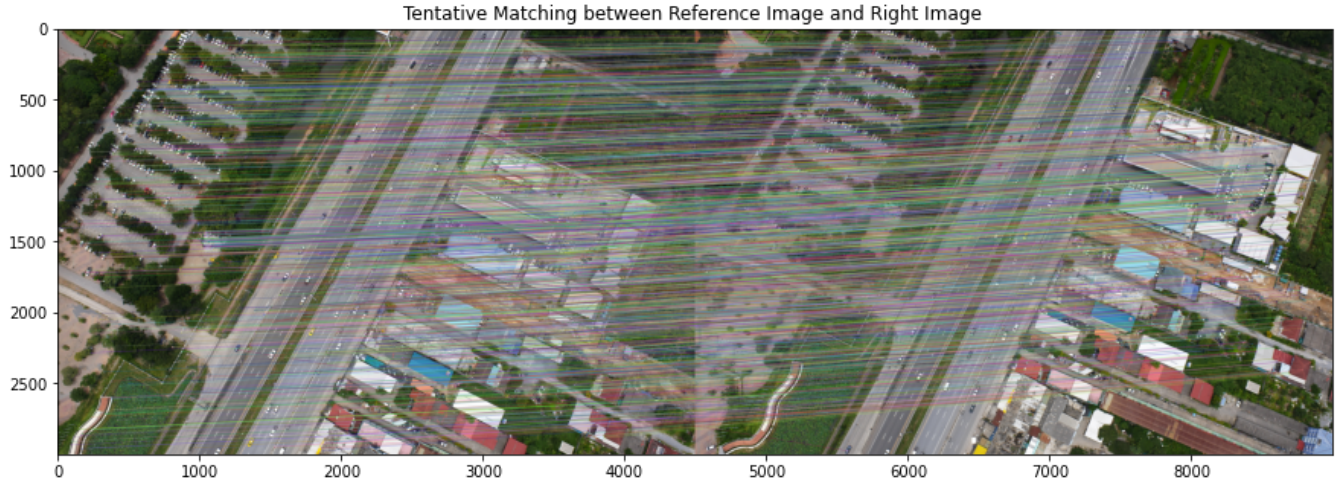
```
2102
```

```
lf = np.float32(lf)
#cf = np.float32(cf)

FLANN_INDEX_KDTREE = 0
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)
flann = cv2.FlannBasedMatcher(index_params, search_params)

matches_l_c = flann.knnMatch(lf, cf, k=2)


matches_2 = []
ratio = 0.7
# loop over the raw matches
for m in matches_l_c:
  # ensure the distance is within a certain ratio of each
  # other (i.e. Lowe’s ratio test)
  if len(m) == 2 and m[0].distance < m[1].distance * ratio:
      #matches_1.append((m[0].trainIdx, m[0].queryIdx))
      matches_2.append(m[0])


print(len(matches_2))
```

```
1807
```

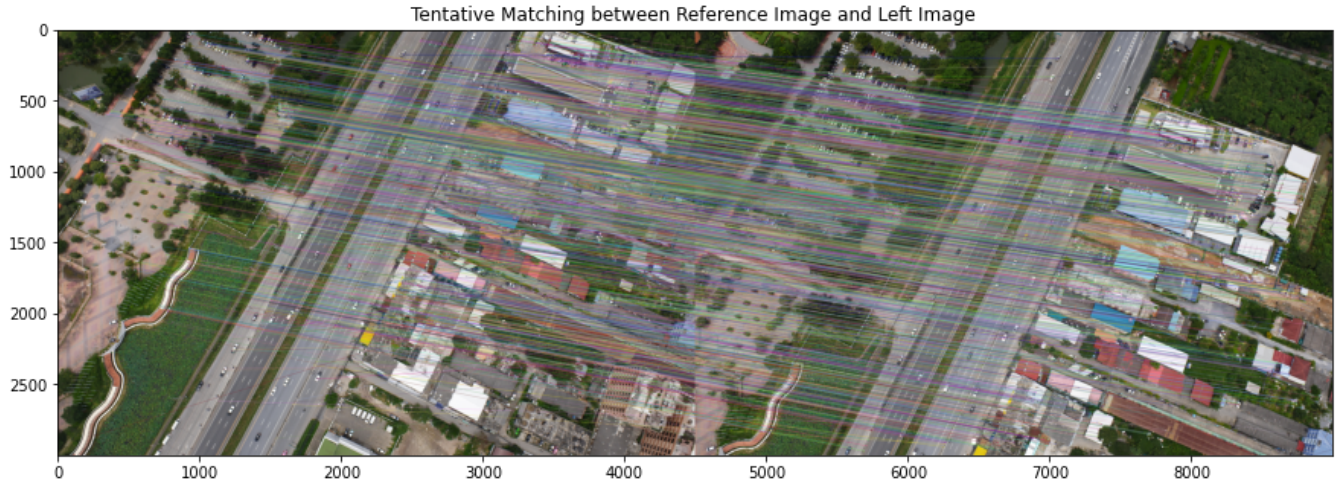
```
# Euclidean distance to compute pairwise distances between the BRISK descriptors of refernce
#p=scipy.spatial.distance.cdist(rf,cf, metric='sqeuclidean')
#matches1=features_matching(p,17000,0.7)
print("Number of matches",len(matches_1))
dispimg1=cv2.drawMatches( right, rpkey,centre, cpkey,matches_1, None,flags=2)
displayplot(dispimg1,'Tentative Matching between Reference Image and Right Image ')
```

```
Number of matches 2102
```



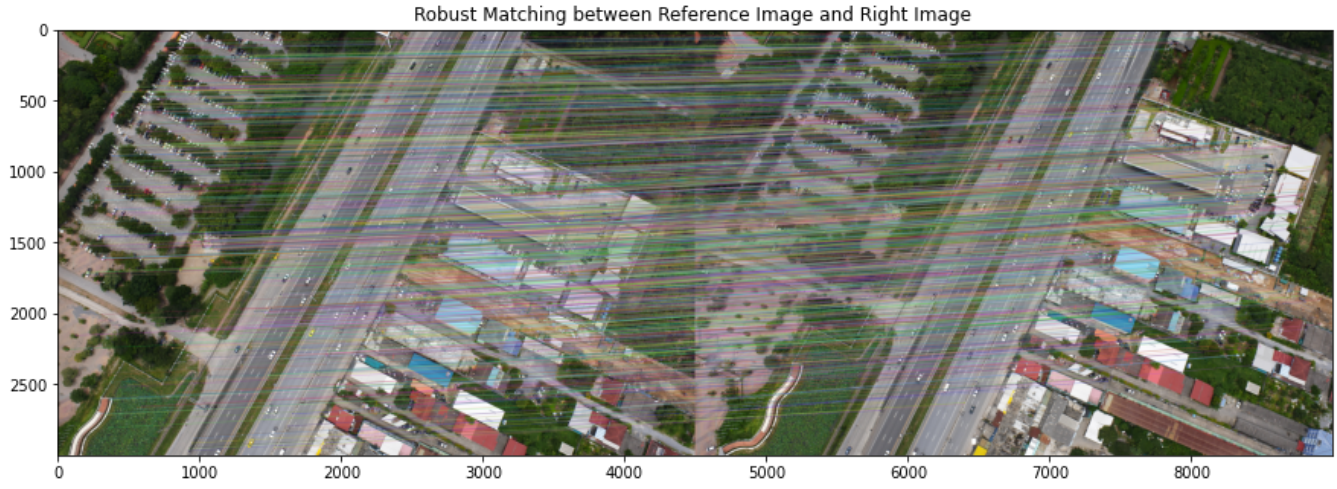
```
# Euclidean distance to compute pairwise distances between the BRISK descriptors of refernce
#p=scipy.spatial.distance.cdist(lf,cf, metric='sqeuclidean')
#matches2=features_matching(p,17000,0.5)
print("Numberof matches",len(matches_2))
dispimg2=cv2.drawMatches(left, lpkey,centre, cpkey, matches_2, None,flags=2)
displayplot(dispimg2,'Tentative Matching between Reference Image and Left Image ')
```

```
Numberof matches 1807
```



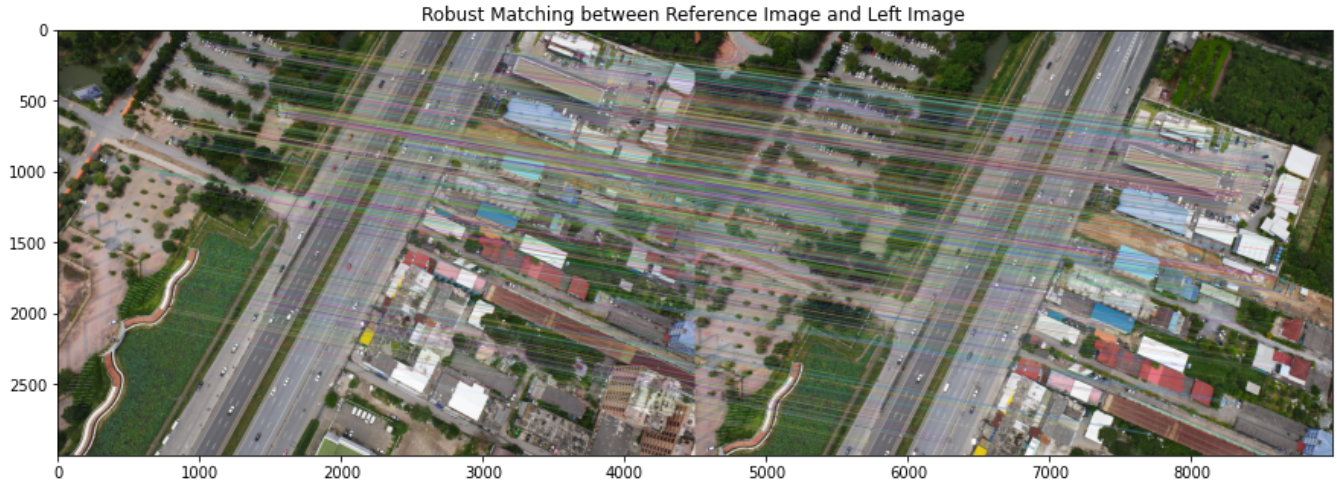

```
# Estimate homography 1
#Compute H1
im1_pts=np.empty((len(matches_1),2))
im2_pts=np.empty((len(matches_1),2))
for i in range(0,len(matches_1)):
  m = matches_1[i]
  (a_x, a_y) = rpkey[m.queryIdx].pt
  (b_x, b_y) = cpkey[m.trainIdx].pt
  #(a_x, a_y) = rpkey[m[0]].pt
  #(b_x, b_y) = cpkey[m[1]].pt
  im1_pts[i]=(a_x, a_y)
  im2_pts[i]=(b_x, b_y)
H=compute_Homography(im1_pts,im2_pts)
#Robustly estimate Homography 1 using RANSAC
H1=RANSAC_alg(rpkey ,cpkey, matches_1,  nRANSAC=1000, RANSACthresh=4)
global inlier_matchset
dispimg1=cv2.drawMatches(right, rpkey, centre, cpkey, inlier_matchset, None,flags=2)
displayplot(dispimg1,'Robust Matching between Reference Image and Right Image ')

#print(ok)
```

```
Number of best inliers 1605
```




```
# Estimate homography 2
#Compute H2
im1_pts=np.empty((len(matches_2),2))
im2_pts=np.empty((len(matches_2),2))
for i in range(0,len(matches_2)):
  m = matches_2[i]
  (a_x, a_y) = lpkey[m.queryIdx].pt
  (b_x, b_y) = cpkey[m.trainIdx].pt
  im1_pts[i]=(a_x, a_y)
  im2_pts[i]=(b_x, b_y)
H=compute_Homography(im1_pts,im2_pts)
#Robustly estimate Homography 2 suing RANSAC
H2=RANSAC_alg(lpkey ,cpkey, matches_2,  nRANSAC=1000, RANSACthresh=4)
dispimg2=cv2.drawMatches(left, lpkey, centre, cpkey,inlier_matchset, None,flags=2)
displayplot(dispimg2,'Robust Matching between Reference Image and Left Image ')
```

```
Number of best inliers 1191
```



```
H1=H1/H1[2,2]
H2=H2/H2[2,2]


def warpThreeImages(img1, img2,img3, H1,H2):
    #img1-centre,img2-left,img3-right
    h1, w1 = img1.shape[:2]
    h2, w2 = img2.shape[:2]
    h3, w3 = img3.shape[:2]

    pts1 = np.float32([[0, 0], [0, h1], [w1, h1], [w1, 0]]).reshape(-1, 1, 2)
    pts2 = np.float32([[0, 0], [0, h2], [w2, h2], [w2, 0]]).reshape(-1, 1, 2)
    pts3 = np.float32([[0, 0], [0, h3], [w3, h3], [w3, 0]]).reshape(-1, 1, 2)

    pts2_ = cv2.perspectiveTransform(pts2, H1)
    pts3_ = cv2.perspectiveTransform(pts3, H2)

    #pts = np.concatenate((pts1, pts2_), axis=0)
    pts = np.concatenate((pts1, pts2_,pts3_), axis=0)

    [xmin, ymin] = np.int32(pts.min(axis=0).ravel() - 0.5)
    [xmax, ymax] = np.int32(pts.max(axis=0).ravel() + 0.5)
    t = [-xmin, -ymin]
    Ht = np.array([[1, 0, t[0]], [0, 1, t[1]], [0, 0, 1]])  # translate

    result = cv2.warpPerspective(img2, Ht@H1, (xmax-xmin, ymax-ymin))

    result2 = cv2.warpPerspective(img3, Ht@H2, (xmax-xmin, ymax-ymin))

    result[t[1]:h1+t[1], t[0]:w1+t[0]] = img1
    result2[t[1]:h1+t[1], t[0]:w1+t[0]] = img1

    #indices = np.where(result==0)[0]

    #print(len(indices))

    #result[indices] = result2[indices]
    result3 = result | result2

    #result[:t[1], :t[0]] = result2[:t[1], :t[0]]

    return result,result2,result3


# Function to simulate OR Gate
#def OR(A, B): #A->Img1,B->Img2

    #A[np.where(B!=0)[0]] = B[np.where(B!=0)[0]]
    #plt.imshow(A)


#c= OR(2,3)
```

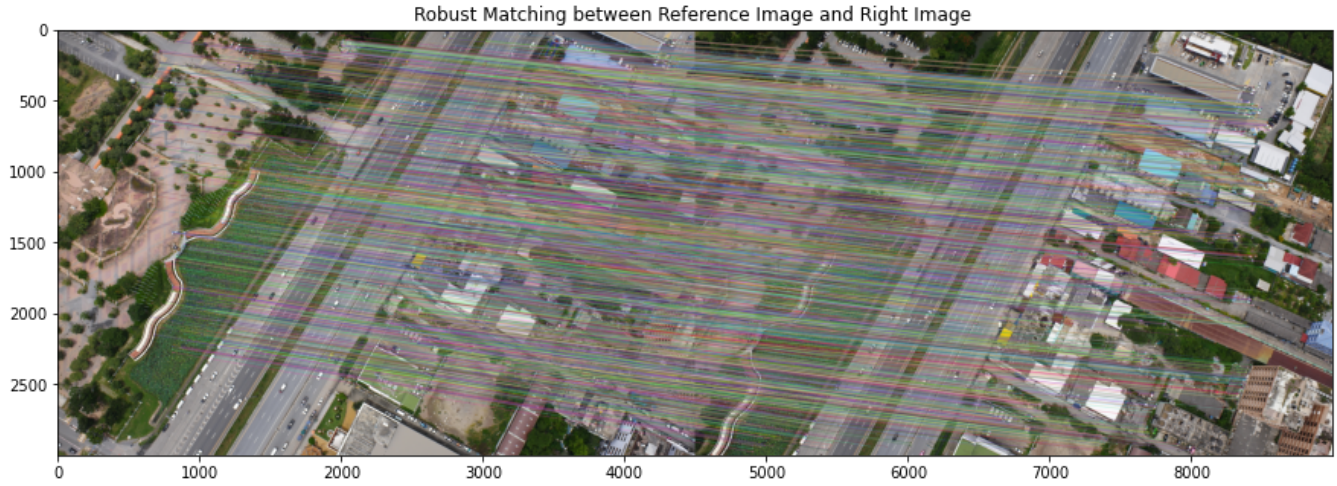
```
#print(c)


def warp4Images(img1, img2,img3,img42,img53,img64,img75,img86,img97, H1,H2,H3,H4,H5,H6,H7,H8)
    #img1-centre,img2-left,img3-right
    h1, w1 = img1.shape[:2]
    h2, w2 = img2.shape[:2]
    h3, w3 = img3.shape[:2]
    h4, w4 = img42.shape[:2]
    h5, w5 = img53.shape[:2]
    h6, w6 = img64.shape[:2]
    h7, w7 = img75.shape[:2]
    h8, w8 = img86.shape[:2]
    h9, w9 = img97.shape[:2]


    pts1 = np.float32([[0, 0], [0, h1], [w1, h1], [w1, 0]]).reshape(-1, 1, 2)
    pts2 = np.float32([[0, 0], [0, h2], [w2, h2], [w2, 0]]).reshape(-1, 1, 2)
    pts3 = np.float32([[0, 0], [0, h3], [w3, h3], [w3, 0]]).reshape(-1, 1, 2)
    pts4 = np.float32([[0, 0], [0, h4], [w4, h4], [w4, 0]]).reshape(-1, 1, 2)
    pts5 = np.float32([[0, 0], [0, h5], [w5, h5], [w5, 0]]).reshape(-1, 1, 2)
    pts6 = np.float32([[0, 0], [0, h6], [w6, h6], [w6, 0]]).reshape(-1, 1, 2)
    pts7 = np.float32([[0, 0], [0, h7], [w7, h7], [w7, 0]]).reshape(-1, 1, 2)
    pts8 = np.float32([[0, 0], [0, h8], [w8, h8], [w8, 0]]).reshape(-1, 1, 2)
    pts9 = np.float32([[0, 0], [0, h9], [w9, h9], [w9, 0]]).reshape(-1, 1, 2)


    pts2_ = cv2.perspectiveTransform(pts2, H1)
    pts3_ = cv2.perspectiveTransform(pts3, H2)
    pts4_ = cv2.perspectiveTransform(pts4, H1@H3)
    pts5_ = cv2.perspectiveTransform(pts5, H2@H4)
    pts6_ = cv2.perspectiveTransform(pts6, H1@H3@H5)
    pts7_ = cv2.perspectiveTransform(pts7, H2@H4@H6)
    pts8_ = cv2.perspectiveTransform(pts8, H1@H3@H5@H7)
    pts9_ = cv2.perspectiveTransform(pts9, H1@H3@H5@H7@H8)

    print('Step1:Done')


    #pts = np.concatenate((pts1, pts2_), axis=0)
    pts = np.concatenate((pts1, pts2_,pts3_,pts4_,pts5_,pts6_,pts7_,pts8_,pts9_), axis=0)

    [xmin, ymin] = np.int32(pts.min(axis=0).ravel() - 0.5)
    [xmax, ymax] = np.int32(pts.max(axis=0).ravel() + 0.5)
    t = [-xmin, -ymin]
    Ht = np.array([[1, 0, t[0]], [0, 1, t[1]], [0, 0, 1]])  # translate

    print('Step2:Done')

    result = cv2.warpPerspective(img2, Ht@H1, (xmax-xmin, ymax-ymin))
```

```
    result2 = cv2.warpPerspective(img3, Ht@H2, (xmax-xmin, ymax-ymin))

    result4 = cv2.warpPerspective(img42, Ht@H1@H3, (xmax-xmin, ymax-ymin))
    result6= cv2.warpPerspective(img53, Ht@H2@H4, (xmax-xmin, ymax-ymin))
    result8= cv2.warpPerspective(img64, Ht@H1@H3@H5, (xmax-xmin, ymax-ymin))
    result10= cv2.warpPerspective(img75, Ht@H2@H4@H6, (xmax-xmin, ymax-ymin))
    result12= cv2.warpPerspective(img86, Ht@H1@H3@H5@H7, (xmax-xmin, ymax-ymin))
    result14= cv2.warpPerspective(img97, Ht@H1@H3@H5@H7@H8, (xmax-xmin, ymax-ymin))


    result[t[1]:h1+t[1], t[0]:w1+t[0]] = img1
    #result2[t[1]:h1+t[1], t[0]:w1+t[0]] = img1


    #Union

    #result3 = result | result2
    result3 = np.maximum(result,result2)

    #result5 = result3 | result4
    result5 = np.maximum(result3,result4)


    #result7 = result5 | result6
    result7 = np.maximum(result5,result6)

    #result7 = OR(result3,result4)

    #result9 = result7 | result8
    result9 = np.maximum(result7,result8)

    #result11 = result9 | result10
    result11 = np.maximum(result9,result10)

    result13 = np.maximum(result11,result12)

    result15 = np.maximum(result13,result14)


    #result6,result7=[],[]


    #result[:t[1], :t[0]] = result2[:t[1], :t[0]]

    return result,result2,result3,result4,result5,result6,result7,result8,result9,result10,re


print(left1.shape,lf1.shape,lp1.shape)
```

```
 (3000, 4500, 3) (42088, 64) (42088, 2)
```

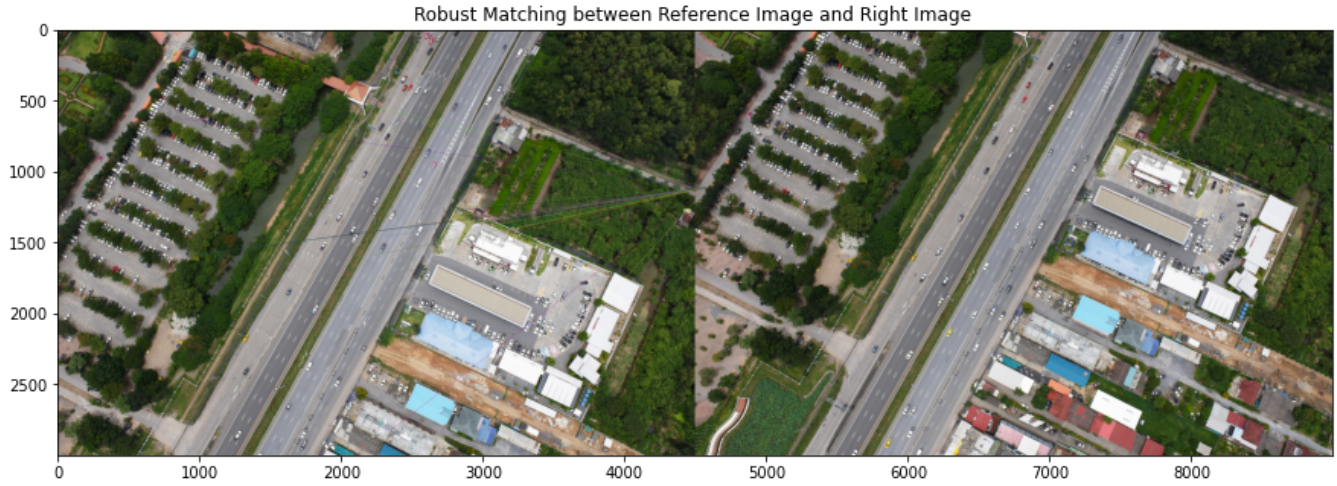
```
def get_H(imgl1,imgl,lppkey1,lpp1,lff1,lppkey,lpp,lff):
  #FLANN_INDEX_KDTREE = 1
  #index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
  #search_params = dict(checks=50)
  #flann = cv2.FlannBasedMatcher(index_params, search_params)
  flann = cv2.BFMatcher()


  lff1 = np.float32(lff1)
  lff = np.float32(lff)


  matches_lf1_lf = flann.knnMatch(lff1, lff, k=2)

  print(len(matches_lf1_lf))

  matches_4 = []
  ratio = 0.85
  # loop over the raw matches
  for m in matches_lf1_lf:
    # ensure the distance is within a certain ratio of each
    # other (i.e. Lowe’s ratio test)
    if len(m) == 2 and m[0].distance < m[1].distance * ratio:
        #matches_1.append((m[0].trainIdx, m[0].queryIdx))
        matches_4.append(m[0])

  print("Number of matches",len(matches_4))

  # Estimate homography 1
  #Compute H1
  imm1_pts=np.empty((len(matches_4),2))
  imm2_pts=np.empty((len(matches_4),2))
  for i in range(0,len(matches_4)):
    m = matches_4[i]
    (a_x, a_y) = lppkey1[m.queryIdx].pt
    (b_x, b_y) = lppkey[m.trainIdx].pt
    imm1_pts[i]=(a_x, a_y)
    imm2_pts[i]=(b_x, b_y)
  H=compute_Homography(imm1_pts,imm2_pts)
  #Robustly estimate Homography 1 using RANSAC
  Hn=RANSAC_alg(lppkey1 ,lppkey, matches_4,  nRANSAC=1500, RANSACthresh=6)

  dispimg1=cv2.drawMatches(imgl1, lppkey1, imgl, lppkey, inlier_matchset, None,flags=2)
  displayplot(dispimg1,'Robust Matching between Reference Image and Right Image ')

  return Hn/Hn[2,2]


H3 = get_H(left1,left,lpkey1,lp1,lf1,lpkey,lp,lf)
```

```
42088
Number of matches 5466
Number of best inliers 3130
```



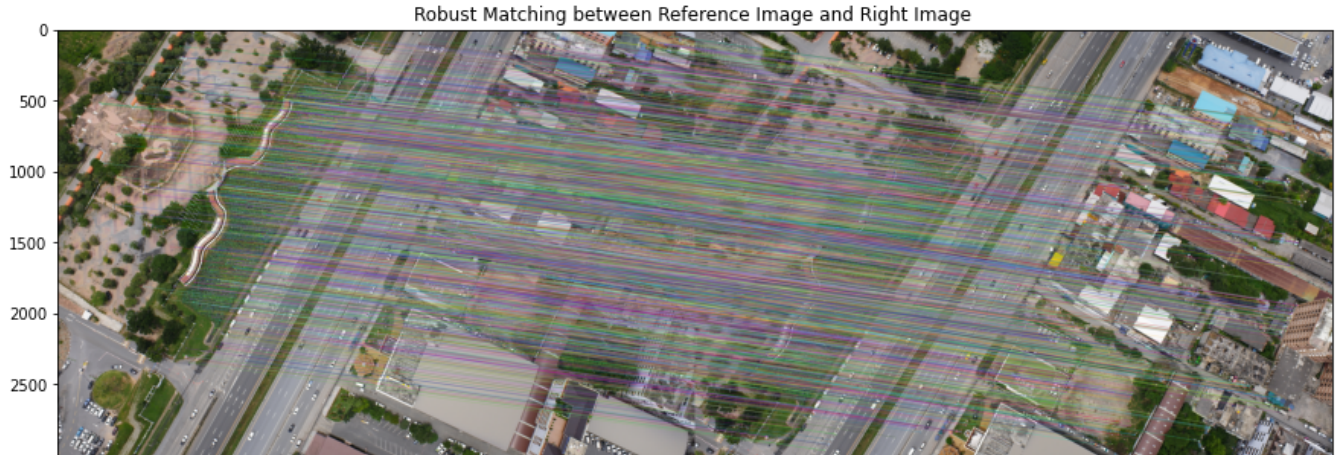

```
H4 = get_H(right1,right,rpkey1,rp1,rf1,rpkey,rp,rf)
```

```
29178
Number of matches 859
Number of best inliers 28
```



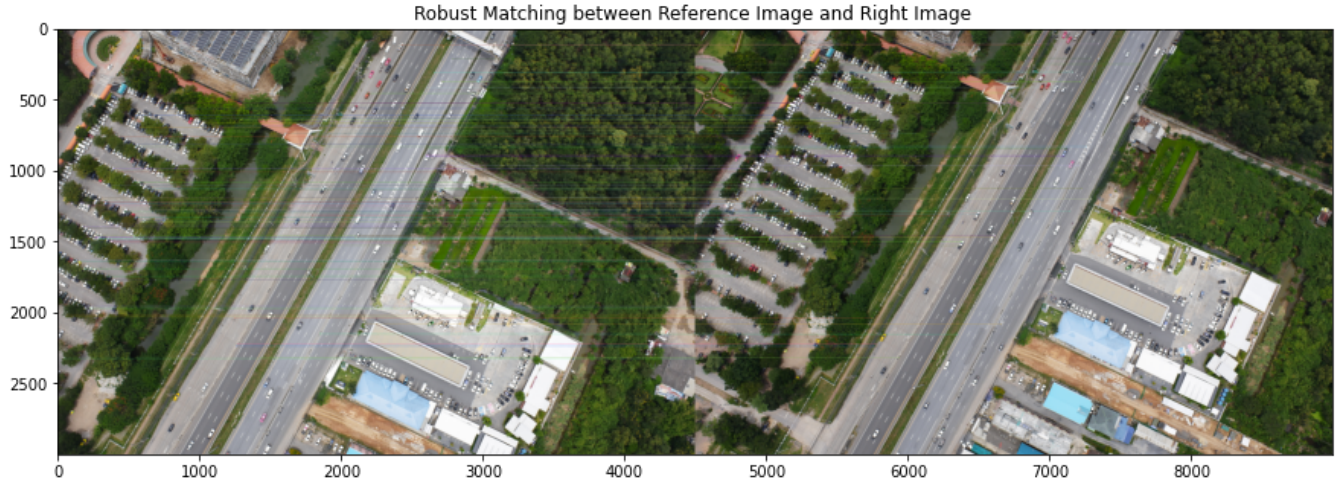

```
H5 = get_H(left2,left1,lpkey2,lp2,lf2,lpkey1,lp1,lf1)
```

```
42852
Number of matches 5594
Number of best inliers 3239
```




```
H6 = get_H(right2,right1,rpkey2,rp2,rf2,rpkey1,rp1,rf1)
```

```
28411
Number of matches 1118
Number of best inliers 87
```

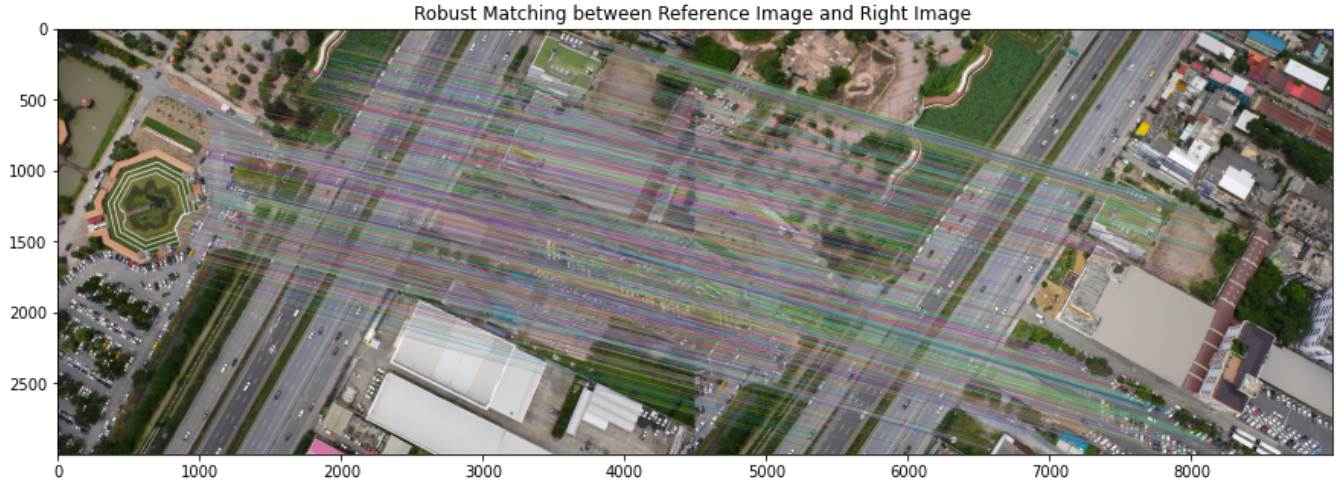



```
H7 = get_H(left3,left2,lpkey3,lp3,lf3,lpkey2,lp2,lf2)
```

```
47110
Number of matches 5696
Number of best inliers 3221
```


```
H8 = get_H(left4,left3,lpkey4,lp4,lf4,lpkey3,lp3,lf3)
```

```
37911
Number of matches 4016
Number of best inliers 2222
```




```
left_centre_warp,right_centre_warp,combined_warp3,left_left_warp,combined_warp4,right_right_w
combined_warp5,left_left_left_warp,combined_warp6,right_right_right_warp,combined_warp7,left_
combined_warp8,left_5_warp,combined_warp9 = warp4Images(centre, left, right,left1,right1,left
```

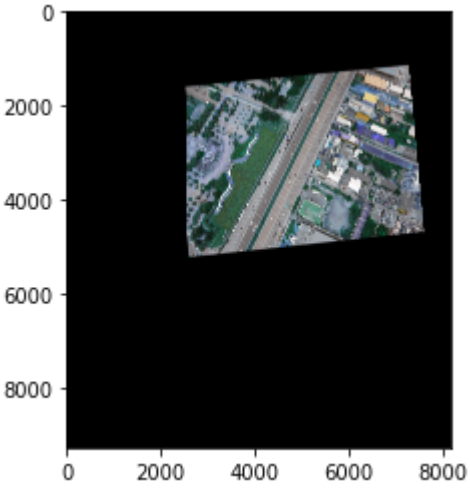
```
Step1:Done
Step2:Done
```

```
plt.imshow(combined_warp3)
```

```
<matplotlib.image.AxesImage at 0x7fcc08646450>
```

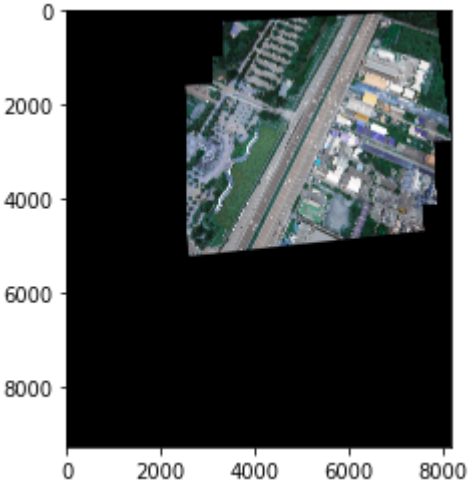

```
plt.imshow(left_left_warp)
```

```
<matplotlib.image.AxesImage at 0x7fcc11706d50>
```



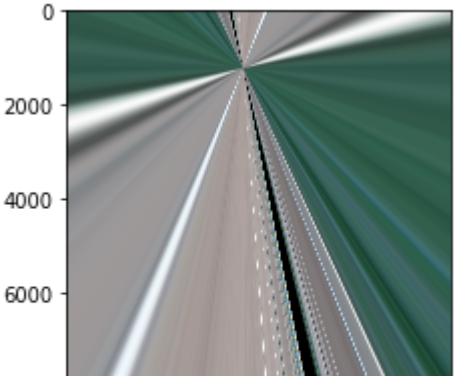

```
plt.imshow(combined_warp4)
```

```
<matplotlib.image.AxesImage at 0x7fcc08db6a10>
```



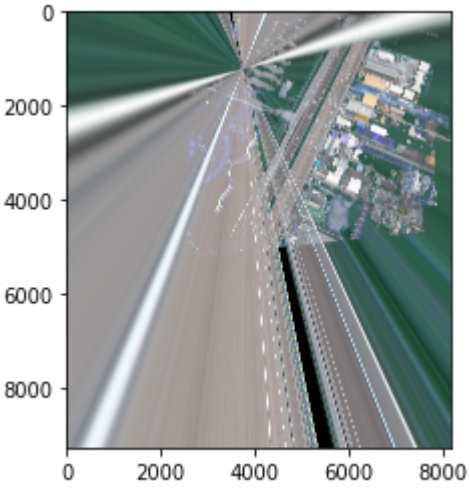

```
plt.imshow(right_right_warp)
```

```
<matplotlib.image.AxesImage at 0x7fcc08440c50>
```



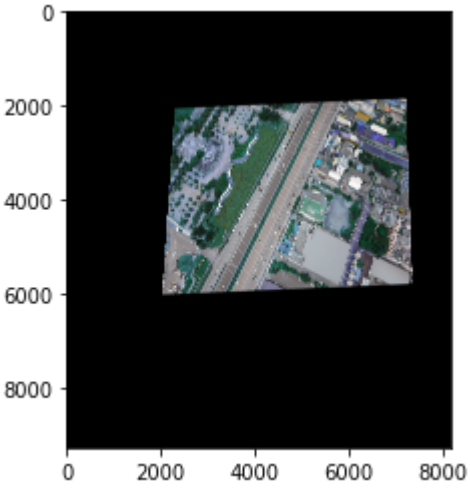

```
plt.imshow(combined_warp5)
```

```
<matplotlib.image.AxesImage at 0x7fcc084263d0>
```



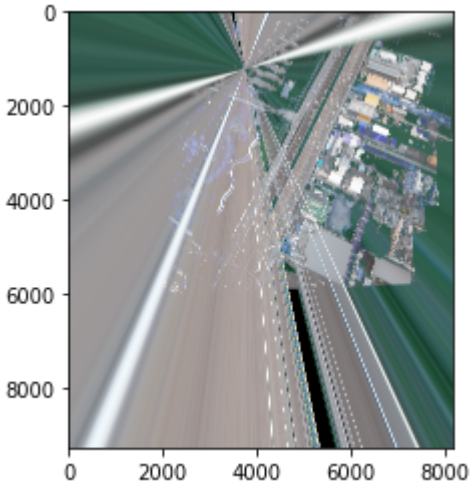

```
plt.imshow(left_left_left_warp)
```

```
<matplotlib.image.AxesImage at 0x7fcc084e2810>
```



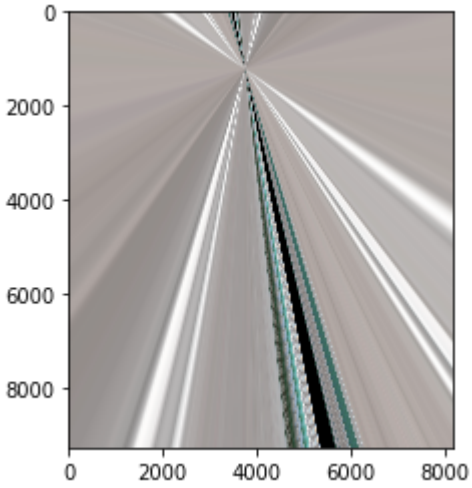

```
plt.imshow(combined_warp6)
```

```
<matplotlib.image.AxesImage at 0x7fcc0837fb50>
```




```
plt.imshow(right_right_right_warp)
```

```
<matplotlib.image.AxesImage at 0x7fcc083653d0>
```

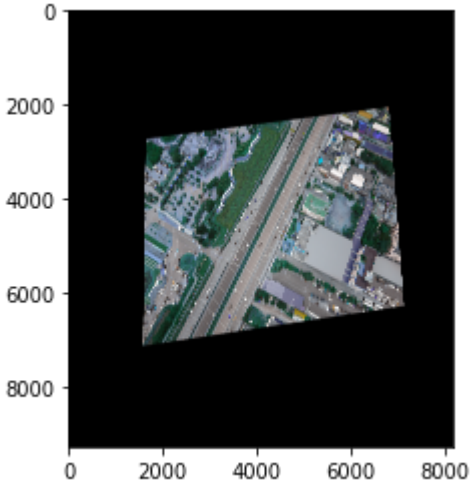



```
plt.imshow(combined_warp7)
```

```
<matplotlib.image.AxesImage at 0x7fcc082bedd0>
```

```
plt.imshow(left_4_warp)
```

```
<matplotlib.image.AxesImage at 0x7fcc082a7810>
```




```
plt.figure(figsize = (25,20))
plt.subplot(331)
plt.imshow(left4)
plt.title("1st Image")
plt.subplot(332)
plt.imshow(left3)
plt.title("2nd Image")
plt.subplot(333)
plt.imshow(left2)
plt.title("3rd Image")
plt.subplot(334)
plt.imshow(left1)
plt.title("4th Image")
plt.subplot(335)
plt.imshow(left)
plt.title("5th Image")
plt.subplot(336)
plt.imshow(centre)
plt.title("6th Image")
plt.subplot(337)
plt.imshow(right)
plt.title("7th Image")
plt.subplot(338)
plt.imshow(right1)
plt.title("8th Image")
plt.subplot(339)
plt.imshow(right2)
plt.title("9th Image")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-ae43045b49db> in <module>()
----> 1 plt.figure(figsize = (25,20))
      2 plt.subplot(331)
      3 plt.imshow(left4)
      4 plt.title("1st Image")
      5 plt.subplot(332)

NameError: name 'plt' is not defined
```

```
plt.figure(figsize = (25,20))
plt.subplot(241)
plt.imshow(combined_warp3)
plt.title("3-Images Mosaic")
plt.subplot(242)
plt.imshow(combined_warp4)
plt.title("4-Images Mosaic")
plt.subplot(243)
plt.imshow(combined_warp5)
plt.title("5-Images Mosaic")
plt.subplot(244)
plt.imshow(combined_warp6)
plt.title("6-Images Mosaic")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-62d18541a749> in <module>()
----> 1 plt.figure(figsize = (25,20))
      2 plt.subplot(241)
      3 plt.imshow(combined_warp3)
      4 plt.title("3-Images Mosaic")
      5 plt.subplot(242)

NameError: name 'plt' is not defined
```

SEARCH STACK OVERFLOW

```
plt.figure(figsize = (25,20))
plt.subplot(245)
plt.imshow(combined_warp7)
plt.title("7-Images Mosaic")
plt.subplot(246)
plt.imshow(combined_warp8)
plt.title("8-Images Mosaic")
plt.subplot(247)
plt.imshow(combined_warp9)
plt.title("9-Images Mosaic")
```

```
Text(0.5, 1.0, '9-Images Mosaic')
```