

```
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 tqdm.notebook import tqdm, trange

import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
from torch.autograd import Variable
import torchvision
from torchvision import datasets, models, transforms
from torch.utils.data import Dataset, DataLoader, ConcatDataset
from skimage import io, transform, data
from torchvision import transforms, utils
import numpy as np
import math
import glob
import matplotlib.pyplot as plt
import time
import os
import copy
import sklearn.svm
import cv2
from matplotlib import pyplot as plt
import numpy as np
from os.path import exists
import pandas as pd
import PIL
import random
from google.colab import drive
from sklearn.metrics.cluster import completeness_score
from sklearn.cluster import KMeans
from tqdm import tqdm, tqdm_notebook
from functools import partial
from torchsummary import summary
from torchvision.datasets import ImageFolder
```

```
from torch.utils.data.sampler import SubsetRandomSampler
```

```
from google.colab import drive
```

```
# This will prompt for authorization.
```

```
drive.mount('/content/drive')
```

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

```
!pip install opencv-python==3.4.2.17
```

```
!pip install opencv-contrib-python==3.4.2.17
```

Requirement already satisfied: opencv-python==3.4.2.17 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: numpy>=1.14.5 in /usr/local/lib/python3.7/dist-packages (
Requirement already satisfied: opencv-contrib-python==3.4.2.17 in /usr/local/lib/python3.7/dist-packages (
Requirement already satisfied: numpy>=1.14.5 in /usr/local/lib/python3.7/dist-packages (

```
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
```

```
        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 2xn 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 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].trainIdx

```

```

#queryInd = matches[1][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 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, best_inliers

files_all=[]
for file in os.listdir("/content/drive/MyDrive/RGB-img/img/"):
    if file.endswith(".JPG"):
        files_all.append(file)

files_all.sort()
folder_path = '/content/drive/MyDrive/RGB-img/img/'

centre_file = folder_path + files_all[50]
left_files_path_rev = []
right_files_path = []

for file in files_all[:51]:
    left_files_path_rev.append(folder_path + file)

left_files_path = left_files_path_rev[::-1]

for file in files_all[50:100]:
    right_files_path.append(folder_path + file)

gridsize = 8
clahe = cv2.createCLAHE(clipLimit=2.0,tileGridSize=(gridsize,gridsize))

images_left_bgr = []
images_right_bgr = []

images_left = []
images_right = []

for file in tqdm(left_files_path):
    left_image_sat= cv2.imread(file)
    lab = cv2.cvtColor(left_image_sat, cv2.COLOR_BGR2LAB)
    lab[...,0] = clahe.apply(lab[...,0])

```

```

left_image_sat = cv2.cvtColor(lab, cv2.COLOR_LAB2BGR)
left_img = cv2.resize(left_image_sat, None, fx=0.20, fy=0.20, interpolation = cv2.INTER_CUBIC)
images_left.append(cv2.cvtColor(left_img, cv2.COLOR_BGR2GRAY).astype('float32')/255.)
images_left_bgr.append(left_img)

for file in tqdm(right_files_path):
    right_image_sat= cv2.imread(file)
    lab = cv2.cvtColor(right_image_sat, cv2.COLOR_BGR2LAB)
    lab[...,0] = clahe.apply(lab[...,0])
    right_image_sat = cv2.cvtColor(lab, cv2.COLOR_LAB2BGR)
    right_img = cv2.resize(right_image_sat, None, fx=0.20, fy=0.20, interpolation = cv2.INTER_CUBIC)
    images_right.append(cv2.cvtColor(right_img, cv2.COLOR_BGR2GRAY).astype('float32')/255.)
    images_right_bgr.append(right_img)

100%|██████████| 51/51 [00:57<00:00, 1.13s/it]
100%|██████████| 50/50 [00:56<00:00, 1.13s/it]

images_left_bgr_no_enhance = []
images_right_bgr_no_enhance = []

for file in tqdm(left_files_path):
    left_image_sat= cv2.imread(file)
    left_img = cv2.resize(left_image_sat, None, fx=0.20, fy=0.20, interpolation = cv2.INTER_CUBIC)
    images_left_bgr_no_enhance.append(left_img)

for file in tqdm(right_files_path):
    right_image_sat= cv2.imread(file)
    right_img = cv2.resize(right_image_sat, None, fx=0.20, fy=0.20, interpolation = cv2.INTER_CUBIC)
    images_right_bgr_no_enhance.append(right_img)

100%|██████████| 51/51 [00:20<00:00, 2.45it/s]
100%|██████████| 50/50 [00:20<00:00, 2.47it/s]

surf = cv2.xfeatures2d.SURF_create(40)
keypoints_all_left_surf = []
descriptors_all_left_surf = []
points_all_left_surf=[]

keypoints_all_right_surf = []
descriptors_all_right_surf = []
points_all_right_surf=[]

for imgs in tqdm(images_left_bgr):
    kpt = surf.detect(imgs, None)
    kpt, descrip = surf.compute(imgs, kpt)
    keypoints_all_left_surf.append(kpt)
    descriptors_all_left_surf.append(descrip)
    points_all_left_surf.append(np.asarray([[p.pt[0], p.pt[1]] for p in kpt]))

```

```

for imgs in tqdm(images_right_bgr):
    kpt = surf.detect(imgs, None)
    kpt, descrip = surf.compute(imgs, kpt)
    keypoints_all_right_surf.append(kpt)
    descriptors_all_right_surf.append(descrip)
    points_all_right_surf.append(np.asarray([[p.pt[0], p.pt[1]] for p in kpt]))

```

```

100%|██████████| 51/51 [02:40<00:00, 3.14s/it]
100%|██████████| 50/50 [02:30<00:00, 3.02s/it]

```

```

def compute_homography_fast(matched_pts1, matched_pts2, thresh=4):
    #matched_pts1 = cv2.KeyPoint_convert(matched_kp1)
    #matched_pts2 = cv2.KeyPoint_convert(matched_kp2)

    # Estimate the homography between the matches using RANSAC
    H, inliers = cv2.findHomography(matched_pts1,
                                    matched_pts2,
                                    cv2.RANSAC, ransacReprojThreshold =thresh)

    inliers = inliers.flatten()
    return H, inliers

```

```

def get_Hmatrix(imgs, keypts, pts, descriptors, ratio=0.8, thresh=4, disp=False):
    FLANN_INDEX_KDTREE = 2
    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(descriptors[0])
    lff = np.float32(descriptors[1])

```

```

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

```

```

print("\nNumber of matches", len(matches_lf1_lf))

```

```

matches_4 = []
ratio = ratio
# 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 After Lowe's Ratio", len(matches_4))

```

```

matches_idx = np.array([m.queryIdx for m in matches_4])
img1_pts = np.array([keypts[0][idx] for idx in matches_idx])

```

```

imm1_pts = np.array([keypts[0][idx].pt for idx in matches_idx])
matches_idx = np.array([m.trainIdx for m in matches_4])
imm2_pts = np.array([keypts[1][idx].pt for idx in matches_idx])
...

# Estimate homography 1
#Compute H1
# 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) = keypts[0][m.queryIdx].pt
    (b_x, b_y) = keypts[1][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, best_inliers=RANSAC_alg(keypts[0] ,keypts[1], matches_4, nRANSAC=1000, RANSACthresh=6)
...

Hn,inliers = compute_homography_fast(imm1_pts,imm2_pts,thresh)
inlier_matchset = np.array(matches_4)[inliers.astype(bool)].tolist()
print("Number of Robust matches",len(inlier_matchset))
print("\n")
...

if len(inlier_matchset)<50:
    matches_4 = []
    ratio = 0.67
    # loop over the raw matches
    for m in matches_1f1_1f:
        # 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 After Lowe's Ratio New",len(matches_4))

    matches_idx = np.array([m.queryIdx for m in matches_4])
    imm1_pts = np.array([keypts[0][idx].pt for idx in matches_idx])
    matches_idx = np.array([m.trainIdx for m in matches_4])
    imm2_pts = np.array([keypts[1][idx].pt for idx in matches_idx])
    Hn,inliers = compute_homography_fast_other(imm1_pts,imm2_pts)
    inlier_matchset = np.array(matches_4)[inliers.astype(bool)].tolist()
    print("Number of Robust matches New",len(inlier_matchset))
    print("\n")
    ...

#H=compute_Homography(imm1_pts,imm2_pts)
#Robustly estimate Homography 1 using RANSAC
#Hn=RANSAC_alg(keypts[0] ,keypts[1], matches_4, nRANSAC=1500, RANSACthresh=6)

```



```

#global inlier_matchset

if disp==True:
    dispimg1=cv2.drawMatches(imgs[0], keypts[0], imgs[1], keypts[1], inlier_matchset, None,fl
    displayplot(dispimg1,'Robust Matching between Reference Image and Right Image ')

return Hn/Hn[2,2], len(matches_lf1_lf), len(inlier_matchset)

from functools import partial
from tqdm import tqdm
tqdm = partial(tqdm, position=0, leave=True)

H_left_surf = []
H_right_surf = []

num_matches_surf = []
num_good_matches_surf = []

for j in tqdm(range(len(images_left))):
    if j==len(images_left)-1:
        break

    H_a,matches,gd_matches = get_Hmatrix(images_left_bgr[j:j+2][::-1],keypoints_all_left_surf[j
    H_left_surf.append(H_a)
    num_matches_surf.append(matches)
    num_good_matches_surf.append(gd_matches)

for j in tqdm(range(len(images_right))):
    if j==len(images_right)-1:
        break

    H_a,matches,gd_matches = get_Hmatrix(images_right_bgr[j:j+2][::-1],keypoints_all_right_surf
    H_right_surf.append(H_a)

    Number of matches 14215
    Number of matches After Lowe's Ratio 2077
    Number of Robust matches 1347

    22%|██████    | 11/50 [00:15<00:53,  1.37s/it]
    Number of matches 12152
    Number of matches After Lowe's Ratio 1278
    Number of Robust matches 597

    24%|██████    | 12/50 [00:16<00:47,  1.25s/it]
    Number of matches 12197
    Number of matches After Lowe's Ratio 848
    Number of Robust matches 348

    26%|██████    | 13/50 [00:17<00:43,  1.18s/it]

```

Number of matches 11876
 Number of matches After Lowe's Ratio 1962
 Number of Robust matches 1220

28%|██████ | 14/50 [00:18<00:40, 1.12s/it]
 Number of matches 14590
 Number of matches After Lowe's Ratio 1434
 Number of Robust matches 906

30%|██████ | 15/50 [00:19<00:39, 1.12s/it]
 Number of matches 13513
 Number of matches After Lowe's Ratio 2615
 Number of Robust matches 1885

32%|██████ | 16/50 [00:20<00:37, 1.11s/it]
 Number of matches 14484
 Number of matches After Lowe's Ratio 2912
 Number of Robust matches 2187

34%|██████ | 17/50 [00:22<00:37, 1.13s/it]
 Number of matches 13797
 Number of matches After Lowe's Ratio 2282
 Number of Robust matches 1642

36%|██████ | 18/50 [00:23<00:35, 1.12s/it]
 Number of matches 13179
 Number of matches After Lowe's Ratio 2332
 Number of Robust matches 1645

38%|██████ | 19/50 [00:24<00:34, 1.10s/it]
 Number of matches 12609
 Number of matches After Lowe's Ratio 1713
 Number of Robust matches 1164

```
def warpnImages(images_left, images_right,H_left,H_right):
    #img1-centre,img2-left,img3-right

    h, w = images_left[0].shape[:2]

    pts_left = []
    pts_right = []

    pts_centre = np.float32([[0, 0], [0, h], [w, h], [w, 0]]).reshape(-1, 1, 2)

    for j in range(len(H_left)):
        pts = np.float32([[0, 0], [0, h], [w, h], [w, 0]]).reshape(-1, 1, 2)
        pts_left.append(pts)
```

```

for j in range(len(H_right)):
    pts = np.float32([[0, 0], [0, h], [w, h], [w, 0]]).reshape(-1, 1, 2)
    pts_right.append(pts)

pts_left_transformed=[]
pts_right_transformed=[]

```

```

for j,pts in enumerate(pts_left):
    if j==0:
        H_trans = H_left[j]
    else:
        H_trans = H_trans@H_left[j]
    pts_ = cv2.perspectiveTransform(pts, H_trans)
    pts_left_transformed.append(pts_)

```

```

for j,pts in enumerate(pts_right):
    if j==0:
        H_trans = H_right[j]
    else:
        H_trans = H_trans@H_right[j]
    pts_ = cv2.perspectiveTransform(pts, H_trans)
    pts_right_transformed.append(pts_)

```

```
print('Step1:Done')
```

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

```
pts_concat = np.concatenate((pts_centre,np.concatenate(np.array(pts_left_transformed),axi
```

```

[xmin, ymin] = np.int32(pts_concat.min(axis=0).ravel() - 0.5)
[xmax, ymax] = np.int32(pts_concat.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')
```

```
return xmax,xmin,ymax,ymin,t,h,w,Ht
```

```
def final_steps_left(images_left,images_right,H_left,H_right,xmax,xmin,ymax,ymin,t,h,w,Ht):
```

```
warp_imgs_left = []
```

```

for j,H in enumerate(H_left):
    if j==0:

```

```

        H_trans = Ht@H
    else:
        H_trans = H_trans@H
    result = cv2.warpPerspective(images_left[j+1], H_trans, (xmax-xmin, ymax-ymin))

    if j==0:
        result[t[1]:h+t[1], t[0]:w+t[0]] = images_left[0]

    warp_imgs_left.append(result)

print('Step31:Done')

return warp_imgs_left

def final_steps_right(images_left,images_right,H_left,H_right,xmax,xmin,ymax,ymin,t,h,w,Ht):

    warp_imgs_right = []

    for j,H in enumerate(H_right):
        if j==0:
            H_trans = Ht@H
        else:
            H_trans = H_trans@H
        result = cv2.warpPerspective(images_right[j+1], H_trans, (xmax-xmin, ymax-ymin))

        warp_imgs_right.append(result)

    print('Step32:Done')

    return warp_imgs_right

def final_steps_union(warp_imgs_left,warp_imgs_right):
    #Union

    warp_images_all = warp_imgs_left + warp_imgs_right

    warp_img_init = warp_images_all[0]

    #warp_final_all=[]

    for j,warp_img in enumerate(warp_images_all):
        if j==len(warp_images_all)-1:
            break
        black_pixels = np.where((warp_img_init[:, :, 0] == 0) & (warp_img_init[:, :, 1] == 0) &

        warp_img_init[black_pixels] = warp_images_all[j+1][black_pixels]

        #warp_final = np.maximum(warp_img_init,warp_images_all[j+1])
        #warp_img_init = warp_final
        #warp_final_all.append(warp_final)

```

```
print('Step4:Done')
```

```
return warp_img_init
```

```
def final_steps_left_union(images_left,H_left,xmax,xmin,ymax,ymin,t,h,w,Ht):
```

```
    for j,H in enumerate(H_left):
```

```
        if j==0:
```

```
            H_trans = Ht@H
```

```
        else:
```

```
            H_trans = H_trans@H
```

```
        input_img = images_left[j+1]
```

```
        result = np.zeros((ymax-ymin,xmax-xmin,3),dtype='uint8')
```

```
        cv2.warpPerspective(src = np.uint8(input_img), M = H_trans, dsize = (xmax-xmin, ymax-ym
        warp_img_init_curr = result
```

```
        if j==0:
```

```
            result[t[1]:h+t[1], t[0]:w+t[0]] = images_left[0]
```

```
            warp_img_init_prev = result
```

```
            continue
```

```
        black_pixels = np.where((warp_img_init_prev[:, :, 0] == 0) & (warp_img_init_prev[:, :,
```

```
        warp_img_init_prev[black_pixels] = warp_img_init_curr[black_pixels]
```

```
print('Step31:Done')
```

```
return warp_img_init_prev
```

```
def final_steps_right_union(warp_img_prev,images_right,H_right,xmax,xmin,ymax,ymin,t,h,w,Ht):
```

```
    for j,H in enumerate(H_right):
```

```
        if j==0:
```

```
            H_trans = Ht@H
```

```
        else:
```

```
            H_trans = H_trans@H
```

```
        input_img = images_right[j+1]
```

```
        result = np.zeros((ymax-ymin,xmax-xmin,3),dtype='uint8')
```

```
        cv2.warpPerspective(src = np.uint8(input_img), M = H_trans, dsize = (xmax-xmin, ymax-ym
        warp_img_init_curr = result
```

```
        black_pixels = np.where((warp_img_prev[:, :, 0] == 0) & (warp_img_prev[:, :, 1] == 0) &
```

```
        warp_img_prev[black_pixels] = warp_img_init_curr[black_pixels]
```

```
print('Step32:Done')

return warp_img_prev

xmax,xmin,ymax,ymin,t,h,w,Ht = warpnImages(images_left_bgr_no_enhance, images_right_bgr_no_en

Step1:Done
Step2:Done

warp_imgs_left = final_steps_left_union(images_left_bgr_no_enhance,H_left_surf,xmax,xmin,ymax

warp_imgs_all_surf = final_steps_right_union(warp_imgs_left,images_right_bgr_no_enhance,H_rig

Step32:Done

fig,ax =plt.subplots()
fig.set_size_inches(20,20)
ax.imshow(cv2.cvtColor(warp_imgs_all_surf , cv2.COLOR_BGR2RGB))
ax.set_title('100-Images Mosaic-surf')
```

 9s completed at 22:32

 