r08521609_handson8

May 7, 2021

[ ]:
```
%matplotlib inline
import numpy as np
import cv2
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow
import random
from matplotlib import pyplot as plt
import pylab
pylab.rcParams['figure.figsize'] = (20, 15)

#In case your Open CV version do not support SIFT
!pip install opencv-contrib-python==3.4.2.17
```

```
Requirement already satisfied: opencv-contrib-python==3.4.2.17 in
/usr/local/lib/python3.7/dist-packages (3.4.2.17)
Requirement already satisfied: numpy>=1.14.5 in /usr/local/lib/python3.7/dist-
packages (from opencv-contrib-python==3.4.2.17) (1.19.5)
```

[ ]:
```
#Download the left and right perspective of site images
!gdown --id '1RNCdBF9a4fIdcyPvjelyx7dsheLgRoaQ' --output leftSite.jpg
!gdown --id '1zHAtikO9dVHpLPJ-KQPdvKEC-wLB1hHM' --output rightSite.jpg
```

```
Downloading…
From: https://drive.google.com/uc?id=1RNCdBF9a4fIdcyPvjelyx7dsheLgRoaQ
To: /content/leftSite.jpg
100% 576k/576k [00:00<00:00, 79.5MB/s]
Downloading…
From: https://drive.google.com/uc?id=1zHAtikO9dVHpLPJ-KQPdvKEC-wLB1hHM
To: /content/rightSite.jpg
100% 554k/554k [00:00<00:00, 72.1MB/s]
```

[ ]:
```
#Read the left and right perspective of site images
img1_bgr = cv2.imread('leftSite.jpg')
img2_bgr  = cv2.imread('rightSite.jpg')

#Resize images for convenience
def resizeimg (img):
```

```
    img_resize = cv2.resize(img, (int(img.shape[1]*0.5),int(img.shape[0]*0.5)),␣
 ↪interpolation = cv2.INTER_AREA)
    return img_resize

img1_bgr = resizeimg(img1_bgr)
img2_bgr = resizeimg(img2_bgr)

#display the images
cv2_imshow(img1_bgr)
cv2_imshow(img2_bgr)
```

[ ]:
```
# Converting images to gray scale
img1 = cv2.cvtColor(img1_bgr, cv2.COLOR_BGR2GRAY)
img2 = cv2.cvtColor(img2_bgr, cv2.COLOR_BGR2GRAY)

# create a SIFT detector
sift = cv2.xfeatures2d.SIFT_create()

# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)

# matching descriptor vectors with a FLANN based matcher
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)
matches = flann.knnMatch(des1,des2,k=2)
pts1 = []
pts2 = []
# Filter matches using the Lowe's ratio test
for i,(m,n) in enumerate(matches):
    if m.distance < 0.8*n.distance:
        pts1.append(kp1[m.queryIdx].pt)
```

```
        pts2.append(kp2[m.trainIdx].pt)

pts1 = np.int32(pts1)
pts2 = np.int32(pts2)
print(pts1)
```

```
[[  9 216]
 [ 23 373]
 [ 24 370]
 …
 [586 148]
 [587 372]
 [588  64]]
```

[ ]:
```
print(img1.shape)
print(np.max(pts1[:, 0]))
print(np.max(pts1[:, 1]))
```

```
(400, 600)
588
396
```

- Part A: Estimate the fundamental matrix F automatically using normalized 8-point algorithm with SVD
- Part B: Estimate the fundamental matrix F automatically using RANSAC

https://www.cc.gatech.edu/classes/AY2016/cs4476_fall/results/proj3/html/sdai30/index.html

https://github.com/AdityaNair111/RANSAC-based-scene-geometry

- Part C: Estimate the homography H automatically using normalized 4-point algorithm with SVD

https://engineering.purdue.edu/kak/courses-i-teach/ECE661.08/solution/hw4_s1.pdf

https://github.com/AdityaNair111/RANSAC-based-scene-geometry/tree/master/code

[ ]:
```
def random_matches(pts1, pts2, N):
    """
    Choose 8 sets of matching points that are well separated
    """
    c = list(zip(pts1,pts2))
    se = random.sample(c, N)
    se_pts1 = np.array([a for (a, b) in se])
    se_pts2 = np.array([b for (a, b) in se])
    return se_pts1, se_pts2



def normalize_pts(pts1,pts2):
    """
```

```
    recenter the point correspondences pts1 and pts2 in both images
    to their respective centroids before proceeding to compute
    the least-squares solution for f.

    Return:
    -   normalized set of pts1 and pts2
    -   transformation matrices T1 and T2
    """
    count = pts1.shape[0]

    mean_1 = np.mean(pts1[:], axis=1)
    S1 = np.sqrt(2) / np.sqrt(np.var(pts1[:,0]) + np.var(pts1[:,1]))
    T1 = np.array([[S1, 0, -S1 * mean_1[0]],
                   [0, S1, -S1 * mean_1[1]],
                   [0, 0, 1]])
    pts1_homo = np.concatenate((pts1, np.ones((count,1))), axis=1) # the 2D (u,v)␣
↪are represented in homogeneous coordinates as a 3-vector
    pts1_norm = np.dot(T1, pts1_homo.T)

    mean_2 = np.mean(pts2[:], axis=1)
    S2 = np.sqrt(2) / np.sqrt(np.var(pts2[:,0]) + np.var(pts2[:,1]))
    T2 = np.array([[S2, 0, -S2 * mean_2[0]],
                   [0, S2, -S2 * mean_2[1]],
                   [0, 0, 1]])
    pts2_homo = np.concatenate((pts2, np.ones((count,1))), axis=1)
    pts2_norm = np.dot(T2, pts2_homo.T)

    return pts1_norm.T, pts2_norm.T, T1, T2


def singularize(F):
    """
    Reducing the rank of the matrix from 3 to 2
    """
    U, S, V = np.linalg.svd(F)
    S[2] = 0
    F = np.dot(U, np.dot(np.diag(S), V))
    F /= F[2,2]
    #print('rank of F after zeroing out last singular value :', np.linalg.
↪matrix_rank(F))
    return F


def unscaled_mat(F, T1, T2):
    """
    Obtaining the unscaled fundamental matrix
    """
```

```python
  F = np.dot(T1.T, np.dot(F, T2))
  F /= F[2,2]
  return F


def estimate_fundamental_matrix(pts1, pts2, ransac = False):
  """
  Calculates the fundamental matrix.
  Normalize  coordinates through linear transformations before computing the␣
↪fundamental matrix.

  Args:
  -   pts1: A numpy array of shape (N, 2) representing the 2D points in img1
  -   pts2: A numpy array of shape (N, 2) representing the 2D points in img2

  Returns:
  -   F: A numpy array of shape (3, 3) representing the fundamental matrix
  """
  if ransac == False:
    selected_matching_pts = random_matches(pts1, pts2, 8)
  else:
    selected_matching_pts = [pts1, pts2]

  pts1_norm, pts2_norm, T1, T2 = normalize_pts(selected_matching_pts[0],␣
↪selected_matching_pts[1])

  A = np.zeros((pts1_norm.shape[0], 9))

  for i in range(pts1_norm.shape[0]):
    A[i, :] = [ pts2_norm[i,0] * pts1_norm[i,0], pts2_norm[i,0] *␣
↪pts1_norm[i,1], pts2_norm[i,0],
                 pts2_norm[i,1] * pts1_norm[i,0], pts2_norm[i,1] *␣
↪pts1_norm[i,1], pts2_norm[i,1],
                 pts1_norm[i,0], pts1_norm[i,1], 1 ]

  # Solve A*f = 0 using least squares.
  U, S, V = np.linalg.svd(A)
  #print(S)
  F = V[-1].reshape(3, 3)

  # Constrain F to rank 2 by zeroing out last singular value
  F = singularize(F)
  #print('\nNormalized Fundamental Matrix : \n', F)

  # Denormalize using transformation matrices T1 and T2
  unscaled_F = unscaled_mat(F, T1, T2)
  #print('\nUnscaled Fundamental Matrix : \n', unscaled_F)
```

```
    return unscaled_F

def ransac_fundamental_matrix(pts1, pts2):
    """
    Find the best fundamental matrix using RANSAC on potentially matching points.
    RANSAC loop should contain a call to estimate_fundamental_matrix().

    Args:
    -   pts1: A numpy array of shape (N, 2) representing the coordinates of␣
↪possibly matching points from img1
    -   pts2: A numpy array of shape (N, 2) representing the coordinates of␣
↪possibly matching points from img2

    Returns:
    -   best_F: A numpy array of shape (3, 3) representing the best fundamental␣
↪matrix estimation
    -   inlie1: A numpy array of shape (M, 2) representing the subset of␣
↪corresponding points from
                    img1 that are inliers with respect to best_F
    -   inlie2: A numpy array of shape (M, 2) representing the subset of␣
↪corresponding points from
                    img2 that are inliers with respect to best_F
    """

    N = 1000 # iteration times
    S = pts2.shape[0] # count of potential matching points
    r = np.random.randint(S, size=(N,8)) # randomly select N sets of 8 potential␣
↪matching points

    m1 = np.ones((3,S))
    m1[0:2,:] = pts1.T

    m2 = np.ones((3,S))
    m2[0:2,:] = pts2.T

    count = np.zeros(N)
    cost = np.zeros(S)

    t = 1e-2

    # iterate N times of estimate_fundamental_matrix()
    # estimate the total inliers in each iteration
    for i in range(N):
        F = estimate_fundamental_matrix(pts1[r[i,:],:], pts2[r[i,:],:], ransac =␣
↪True)
```

```
    for j in range(S):
      cost[j] = np.dot(np.dot(m2[:,j].T, F), m1[:,j])
    inlie = np.absolute(cost) < t
    count[i] = np.sum(inlie + np.zeros(S), axis = None)

  # sort the total inlier counts of each iteration
  # the iteration with the most inliers serves as the best matching result
  index = np.argsort(-count)
  best = index[0]
  #print("best: ", best)

  # calculate the fundamental matrix based on the 8 best matching points
  best_F = estimate_fundamental_matrix(pts1[r[best,:], :], pts2[r[best,:], :],␣
↪ransac = True)

  for j in range(S):
    cost[j] = np.dot(np.dot(m2[:,j].T, best_F), m1[:,j])

  # filter 100 best matching points cross img1 and img2 and store them as␣
↪inlie1 and inlie2
  confidence = np.absolute(cost)
  index = np.argsort(confidence)
  pts2 = pts2[index]
  pts1 = pts1[index]

  inlie1 = pts1[:100, :]
  inlie2 = pts2[:100, :]

  return best_F, inlie1, inlie2

def homography(pts1, pts2, ransac = False):
  """
  Find H such that H * fp = tp.

  H has eight degrees of freedom, so this needs at least 4 points in fp and tp.
  Calculates the homography using normalized 4-point algorithm.
  Normalize coordinates through linear transformations before computing the␣
↪homography.

  Args:
  -   pts1: A numpy array of shape (N, 2) representing the 2D points in img1
  -   pts2: A numpy array of shape (N, 2) representing the 2D points in img2

  Returns:
  -   F: A numpy array of shape (3, 3) representing the homography
  """
```

```
    if ransac == False:
      selected_matching_pts = random_matches(pts1, pts2, 4)
    else:
      selected_matching_pts = [pts1, pts2]

    pts1_norm, pts2_norm, C1, C2 = normalize_pts(selected_matching_pts[0],␣
↪selected_matching_pts[1])

    # create matrix for linear method, 2 rows for each correspondence pair
    correspondences_count = pts1_norm.shape[0]
    A = np.zeros((2 * correspondences_count, 9))
    for i in range(correspondences_count):
      A[2 * i ] = [-pts1_norm[i][0], -pts1_norm[i][1], -1, 0, 0, 0,
                      pts2_norm[i][0]  * pts1_norm[i][0], pts2_norm[i][0] *␣
↪pts1_norm[i][1], pts2_norm[i][0]]
      A[2 * i + 1] = [0, 0, 0, -pts1_norm[i][0], -pts1_norm[i][1], -1,
                      pts2_norm[i][1]  * pts1_norm[i][0], pts2_norm[i][1] *␣
↪pts1_norm[i][1], pts2_norm[i][1]]

    U, S, V = np.linalg.svd(A)
    H = V[-1].reshape((3, 3))

    # denormalized
    H = np.dot(np.linalg.inv(C2), np.dot(H, C1))
    return H / H[2, 2]
```

[ ]:

```
# Find the best fundamental matrix using RANSAC on potentially matching points
best_F, inlie1, inlie2 = ransac_fundamental_matrix(pts1, pts2)
print(best_F)
```

```
[[-6.66047511e-07  2.03573162e-06  1.57421416e-04]
 [ 6.88274761e-06  2.49891692e-07 -3.73839757e-03]
 [-1.74109919e-03 -6.10724384e-04  1.00000000e+00]]
```

[ ]:

```
# Calculates the fundamental matrix using normalized 8-point algorithm
unscaled_F = estimate_fundamental_matrix(pts1, pts2, ransac = False)
print(unscaled_F)
```

```
[[-1.62291099e-06 -1.73743246e-05  2.94166128e-03]
 [ 1.72438009e-05 -2.51584417e-06 -5.02823109e-03]
 [-5.98076920e-03  6.18048216e-03  1.00000000e+00]]
```

[ ]:

```
# Calculates the homography using normalized 4-point algorithm
unscaled_F = homography(pts1, pts2, ransac = False)
print(unscaled_F)
```

```
[[-4.90314336e+00  3.32443937e+00  9.00427820e+02]
 [-3.12461525e+00  6.91650970e-01  6.91871789e+02]
 [-1.33838482e-02  1.17935223e-02  1.00000000e+00]]
```

[ ]:

```

```

Part D:

Build up VisualSFM or OpenSfM system on your own device and reconstruct a 3D point cloud with arbitrary set of images.

Dataset : Neilstreet building

- Containing aerial images with two different altitudes.
  upper part: 132 images (100 meters high)
  lower part: 216 images (85 meters high)