**ECE 661 Homework 5**

**Weichen Xu**

**xu1363@purdue.edu**

**Theory Questions**

1 How do we differentiate between the inliers and the outliers using RANSAC?

Usually inliers are correspondences of points with merely noise, outliers are only false correspondences. Since using Linear Least-Square method to estimate homography only tolerates inliers, the outliers should be removed.

To view the problem of differentiating between the inliers and the outliers as the linear regression problem. Consider and as two random variables, linear regression estimates a 1-D affine transformation , since at least two points are needed to form a line, RANSAC will randomly pick two points constructing a line, then the support is measure by the number of points that lie within certain distance threshold of the line. The points within the threshold distance of the line are inliers, points outside the threshold distance of the line are outliers.

Applying to the case of 2-D image, at least four pairs of correspondences are randomly selected, then using the linear least-square method to estimate the homography. We calculate the Euclidean distance between transformed point and its original corresponding point, if the value is below certain threshold, the pair is viewed as inliers, otherwise is outliers.

2 How Levenberg-Marquardt (LM) algorithm combines the best of Gradient-Descent (GD) and Gauss-Newton (GN)?

If the starting point is far from the minimum, LM algorithm behaves like GD, as the solution point approaches the minimum, it behaves like GN.

For GN’s solution, it is given by

We can rewrite it as

If were purely diagonal, the solution of GN will be the same as GD. We can heuristically extend the equation to the following form

The LM method add a damping coefficient , making it equal to GN if , close to GD if is much larger than elements of .

**Scale Invariant Feature Transform (SIFT)**

Since we have covered the SIFT method from the last homework, here the details of SIFT method is left out. Initially the SIFT algorithm will extract 1000 feature points with corresponding descriptor vectors, the correspondences are built through choosing pairs with least Euclidean distance. Here the first 100 pairs of correspondences of least Euclidean distances are kept as the valid correspondence pairs.

**Random Sample Consensus (RANSAC)**

The correspondence pairs return by the SIFT algorithm still have some false matching pairs, which are called outliers. Since the next step Linear Least-Square method requires a relatively correct point location to estimate homography, the outliers should be removed to improve the homography. The implementation procedure of RANSAC is following:

1. Select the probability that the ratio of outliers in the initial correspondences, here I choose . Then the total trials of conduct is expressed as , where is the probability of at least one of the trials has no outliers in the following calculation. is the number of correspondences used to estimate the homography for trials. The minimum number of accepted inliers considered as a successful trial , where is the number of all initial correspondences.
2. For repetitive trials, a set of random 4 correspondences are selected, thus an estimated homography can be obtained. Transform all points in the correspondences in domain 1 to the domain 2.
3. Calculating the distance between the transformed point in domain 2 with its corresponding point in domain 2. Only if the distance is less than the threshold , such correspondence pair can be viewed as inliers, otherwise outliers.
4. After trials, the set resulting at the greatest number of inliers pairs is returned to be the result with inliers of RANSAC (If there are sets that has the number of inliers larger than )

**Linear Least-Square Method to Estimate Homography**

Recall on previous homework, the transformation of a point by homography matrix can be expressed as

While

We can then transform to the following equations

For one correspondence pair, we have two equations for eight unknowns, if we have the correspondence pairs that are more than 4, we will have more than eight equations for eight unknowns. In other words, we are solving an over-determined system.

To look for the , where is the vector form of the homography , that minimizes

**Homography Refinement with LM**

The Levenberg-Marquardt (LM) algorithm combines the best of Gradient-Descent (GD) and Gauss-Newton (GN), its equation is given by

While

The optimization goal is to minimize the error between the loss in the estimation

Following are the steps for the LM algorithm:

1. First estimate the initial damping coefficient , it is given by

where .

1. Compute according to the following equation
2. Compute the ratio where
3. The updated damping coefficient for the next iteration is given by
4. Iterate until the loss stop decreasing.

**Stitching Images**

We have five images with overlapping areas, to project them into a common frame, we need to assign the destination frame is the center image , while denotes the th image. Then we can get the homography matrix from the previous steps, let denote is the homography matrix between and .

Since , can directly obtained from the previous steps, we also need to get the , , which can be calculated as

Then we can map each image with its corresponding matrix to the frame of image 3. It’s noted that the stitched image has the larger size than original image 3, and the image 3 should be centered on the stitched image.

**Experiment Results**

A picture containing indoor, sitting, pair, metal

Description automatically generated

Figure 1: Input image 1

A picture containing indoor, sitting, pair, bicycle

Description automatically generated

Figure 2: Input image 2

A picture containing indoor, scissors, pair, metal

Description automatically generated

Figure 3: Input image 3

A picture containing indoor, sitting, pair, sink

Description automatically generated

Figure 4: Input image 4

A close up of a sink

Description automatically generated

Figure 5: Input image 5



Figure 6: Correspondences between image 1 and image 2, while the yellow lines are inliers, the red lines are outliers

A picture containing ware

Description automatically generated

Figure 7: Correspondences between image 2 and image 3, while the yellow lines are inliers, no outliers is detected in this case

A picture containing ware

Description automatically generated

Figure 8: Correspondences between image 3 and image 4, while the yellow lines are inliers, the red lines are outliers

A picture containing box

Description automatically generated

Figure 9: Correspondences between image 4 and image 5, while the yellow lines are inliers, the red lines are outliers

A close up of a piece of paper

Description automatically generated

Figure 10: Stitched panorama image

**Optimal Parameters**

|  |  |  |
| --- | --- | --- |
| SIFT | # of features | 1000 |
| SSD | Threshold of # of best matching pairs | 100 |
| RANSAC | Probability of outliers in all pairs | 0.1 |
| # of pairs used to estimate | 4 |
| Probability of at least one trial has no outliers | 0.99 |

**Code**

#!/usr/bin/env python

# coding: utf-8

import numpy as np

import cv2

import matplotlib.pyplot as plt

from scipy import signal

def GetSSD(patch1, patch2):

# calculate the SDD value from the formula

sum = np.sum(np.sum(np.square(patch1 - patch2)))

return sum

def Correspondence\_SSD (img1, corners1, img2, corners2, window\_size = 21):

# get the relevant correspondent pairs with SSD

value\_min = []

index\_min = []

h = int(window\_size / 2)

corners1 = GetValidCorners(corners1, img1, window\_size)

corners2 = GetValidCorners(corners2, img2, window\_size)

for pt1 in corners1:

x1 = pt1[0]

y1 = pt1[1]

patch1 = img1[y1 - h: y1 + h, x1 - h: x1 + h]

SSD\_all = []

for pt2 in corners2:

x2 = pt2[0]

y2 = pt2[1]

patch2 = img2[y2 - h: y2 + h, x2 - h: x2 + h]

SSD\_all.append(GetSSD(patch1, patch2))

# calculate the current image patch's minimum SSD value

idx = np.argmin(SSD\_all)

value\_min.append(SSD\_all[idx])

index\_min.append(idx)

# set the threshold of keeping pairs

threshold = np.min(value\_min) \* 3

#print(np.min(value\_min))

#print(np.max(value\_min))

corners1\_corr = []

corners2\_corr = []

for i in range(len(value\_min)):

if value\_min[i] < threshold:

corners1\_corr.append(corners1[i])

corners2\_corr.append(corners2[index\_min[i]])

return corners1\_corr, corners2\_corr

def getSIFT(img, nfeatures = 2000):

# get the desired number of interest points and descriptive vectors

sift = cv2.xfeatures2d.SIFT\_create(nfeatures = nfeatures)

img\_gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

kps, des = sift.detectAndCompute(img\_gray, None)

return kps, des

def Correspondence\_SIFT(img1, kps1, des1, img2, kps2, des2):

# Keep only the 100 most relavant pairs of interest points

corr1 = []

corr2 = []

dist\_min\_all = []

for i in range(len(kps1)):

pt1 = kps1[i].pt

pt1 = np.round(pt1)

# measure the Euclidean distance between two descriptor vector

dist\_min = np.sqrt(np.sum(np.multiply(des1[i] - des2[0], des1[i] - des2[0])))

for j in range(len(kps2)):

pt2 = kps2[j].pt

pt2 = np.round(pt2)

dist = np.sqrt(np.sum(np.multiply(des1[i] - des2[j], des1[i] - des2[j])))

if dist <= dist\_min:

dist\_min = dist

pt1\_corr = pt2

corr1.append(pt1)

corr2.append(pt1\_corr)

dist\_min\_all.append(dist\_min)

dist\_sorted = sorted(dist\_min\_all)

corr1\_sorted = []

corr2\_sorted = []

for i in range(100):

idx = np.where(dist\_min\_all == dist\_sorted[i])

idx = idx[0][0]

corr1\_sorted.append(corr1[idx])

corr2\_sorted.append(corr2[idx])

return corr1\_sorted, corr2\_sorted

def get\_homography(target,source):

A = np.zeros((8,8))

b = np.zeros((8,1))

for i in range(4):

x1 = source[i][0]

y1 = source[i][1]

x2 = target[i][0]

y2 = target[i][1]

A[i\*2,:] = [x1,y1,1,0,0,0,-x1\*x2,-y1\*x2]

A[i\*2+1,:] = [0,0,0,x1,y1,1,-x1\*y2,-y1\*y2]

b[i\*2] = x2

b[i\*2+1] = y2

h = np.dot(np.linalg.inv(A),b)

h = np.append(h,1)

H = np.reshape(h,(3,3))

return H

def get\_random\_set(pts1, pts2,n):

# return random set for homography estimation in RANSAC

order = np.random.permutation(len(pts1))

output1 = []

output2 = []

for i in range(n):

output1.append(pts1[order[i]])

output2.append(pts2[order[i]])

return output1, output2

def get\_inliers(pts1, pts2, H, delta):

# obtain inliers for distance below setting threshold

temp = np.array([pts1], dtype='float32')

pts1\_trans = cv2.perspectiveTransform(temp, H)

inliers1 = []

inliers2 = []

outliers1 = []

outliers2 = []

num = 0

for i in range(len(pts1)):

pt1 = pts1\_trans[0][i]

pt2 = pts2[i]

dist = np.sqrt((pt1[0] - pt2[0])\*\*2 + (pt1[1] - pt2[1])\*\*2)

if dist < delta:

num = num + 1

inliers1.append(pts1[i])

inliers2.append(pts2[i])

if dist >= delta:

outliers1.append(pts1[i])

outliers2.append(pts2[i])

return num, inliers1, inliers2, outliers1, outliers2

def RANSAC(pts1, pts2, epsilon, delta):

# M is the number of inliers, N is the number of trials

M = int((1 - epsilon) \* len(pts1))

N = int(np.log(1 - 0.99) / np.log(1 - (1 - epsilon)\*\*4 ))

inliers1 = []

inliers2 = []

for i in range(N):

# randomly pick subset of four points to construct homography

pts1\_4, pts2\_4 = get\_random\_set(pts1, pts2, 4)

H = get\_homography(pts2\_4, pts1\_4)

num, inliers1, inliers2, outliers1, outliers2 = get\_inliers(pts1, pts2, H, delta)

if num > M:

#print(num, 'pairs of inliers found')

break

return inliers1, inliers2, outliers1, outliers2

def get\_homography\_LLS(pts1, pts2):

# whole process for getting homography using multiple correspondences

# using Linear Least-Square method

num = len(pts1)

A = np.zeros((2\*num, 8))

b = np.zeros((2\*num, 1))

for i in range(num):

x1, y1 = pts1[i]

x2, y2 = pts2[i]

A[i\*2,:] = [x1,y1,1,0,0,0,-x1\*x2,-y1\*x2]

A[i\*2+1,:] = [0,0,0,x1,y1,1,-x1\*y2,-y1\*y2]

b[i\*2] = x2

b[i\*2+1] = y2

h = np.dot(np.dot(np.linalg.inv(np.dot(A.transpose(), A) ), A.transpose()), b)

h = np.append(h, 1)

H = h.reshape((3, 3))

return H

def get\_homography\_all(img1, img2):

kps1, des1 = getSIFT(img1)

kps2, des2 = getSIFT(img2)

l1, l2 = Correspondence\_SIFT(img1, kps1, des1, img2, kps2, des2)

inliers1, inliers2, outliers1, outliers2 = RANSAC(l1, l2, epsilon, delta)

H = get\_homography\_LLS(inliers1, inliers2)

return H

def plot\_inliers\_outliers(img1, img2):

kps1, des1 = getSIFT(img1)

kps2, des2 = getSIFT(img2)

l1, l2 = Correspondence\_SIFT(img1, kps1, des1, img2, kps2, des2)

inliers1, inliers2, outliers1, outliers2 = RANSAC(l1, l2, epsilon, delta)

h, w, c = img1.shape

img\_synthetic = np.zeros((h, 2\*w, c), dtype = np.uint8)

img\_synthetic[:, 0:w, :] = img1

img\_synthetic[:, w:2\*w, :] = img2

plt.figure(dpi=1200)

plt.imshow(cv2.cvtColor(img\_synthetic, cv2.COLOR\_BGR2RGB))

for i in range(len(inliers1)):

plt.scatter(inliers1[i][0], inliers1[i][1], color = 'yellow', s = 0.3)

plt.scatter(inliers2[i][0] + w, inliers2[i][1], color = 'yellow', s = 0.3)

plt.plot([inliers1[i][0], inliers2[i][0] + w], [inliers1[i][1], inliers2[i][1]], color = 'yellow', linewidth = 0.1)

for i in range(len(outliers1)):

plt.scatter(outliers1[i][0], outliers1[i][1], color = 'red', s = 0.3)

plt.scatter(outliers2[i][0] + w, outliers2[i][1], color = 'red', s = 0.3)

plt.plot([outliers1[i][0], outliers2[i][0] + w], [outliers1[i][1], outliers2[i][1]], color = 'red', linewidth = 0.1)

def mapping(img\_target,H):

P\_distort = np.array([0,0,1])

Q\_distort = np.array([0,img\_target.shape[0]-1,1])

R\_distort = np.array([img\_target.shape[1]-1,img\_target.shape[0]-1,1])

S\_distort = np.array([img\_target.shape[1]-1,0,1])

P\_world = np.matmul(H,P\_distort)

P\_world = P\_world / P\_world[2]

Q\_world = np.matmul(H,Q\_distort)

Q\_world = Q\_world / Q\_world[2]

R\_world = np.matmul(H,R\_distort)

R\_world = R\_world / R\_world[2]

S\_world = np.matmul(H,S\_distort)

S\_world = S\_world / S\_world[2]

xmin = np.int32(np.round(np.amin([P\_world[0],Q\_world[0],R\_world[0],S\_world[0]])))

xmax = np.int32(np.ceil(np.amax([P\_world[0],Q\_world[0],R\_world[0],S\_world[0]])))

ymin = np.int32(np.round(np.amin([P\_world[1],Q\_world[1],R\_world[1],S\_world[1]])))

ymax = np.int32(np.ceil(np.amax([P\_world[1],Q\_world[1],R\_world[1],S\_world[1]])))

xlen = xmax-xmin

ylen = ymax-ymin

img\_new = np.zeros((ylen,xlen,3), dtype=np.uint8)

print('The output image size is',xlen,ylen)

Hinv = np.linalg.inv(H)

for i in range(xlen):

for j in range(ylen):

input = np.array([i+xmin,j+ymin,1])

output = np.matmul(Hinv,input)

x = np.int(np.round(output[0]/output[2]))

y = np.int(np.round(output[1]/output[2]))

if x>0 and x<img\_target.shape[1]-1 and y>0 and y<img\_target.shape[0]-1:

img\_new[j,i,:] = img\_target[y,x,:]

return img\_new

def get\_bounday(img, H):

# get the four corners' coordinate in transformed domain

P\_distort = np.array([0,0,1])

Q\_distort = np.array([0,img.shape[0]-1,1])

R\_distort = np.array([img.shape[1]-1,img.shape[0]-1,1])

S\_distort = np.array([img.shape[1]-1,0,1])

P\_world = np.matmul(H,P\_distort)

P\_world = P\_world / P\_world[2]

Q\_world = np.matmul(H,Q\_distort)

Q\_world = Q\_world / Q\_world[2]

R\_world = np.matmul(H,R\_distort)

R\_world = R\_world / R\_world[2]

S\_world = np.matmul(H,S\_distort)

S\_world = S\_world / S\_world[2]

xmin = np.int32(np.round(np.amin([P\_world[0],Q\_world[0],R\_world[0],S\_world[0]])))

xmax = np.int32(np.ceil(np.amax([P\_world[0],Q\_world[0],R\_world[0],S\_world[0]])))

ymin = np.int32(np.round(np.amin([P\_world[1],Q\_world[1],R\_world[1],S\_world[1]])))

ymax = np.int32(np.ceil(np.amax([P\_world[1],Q\_world[1],R\_world[1],S\_world[1]])))

return xmin, ymin, xmax - xmin, ymax - ymin

def get\_panorama\_boundary(img\_center, img\_all, H\_all):

# Calculate multiple images mapping to the selected frame

# then obtain the overall boundary for all mapped images

h, w, c = img\_center.shape

xmin = 0

ymin = 0

xmax = w

ymax = h

for i in range(len(img\_all)):

x, y, w, h = get\_bounday(img\_all[i], H\_all[i])

xmin = np.min([x, xmin])

ymin = np.min([y, ymin])

xmax = np.max([xmax, x + w])

ymax = np.max([ymax, y + h])

return xmin, ymin, xmax - xmin, ymax - ymin

def plot\_panorama(panorama, img\_ori, img\_trans, H, x0, y0):

# plot each transformed images in the synthetic panorama

x, y, w, h = get\_bounday(img\_ori, H)

for i in range(y-y0 , y-y0+h):

for j in range(x-x0 , x-x0+w):

if np.sum(panorama[i,j] == np.zeros(3)):

panorama[i,j] = img\_trans[i-(y-y0), j-(x-x0)]

elif np.sum(img\_trans[i-(y-y0), j-(x-x0)] == np.zeros(3)):

panorama[i,j] = panorama[i,j]

else:

panorama[i,j] = (panorama[i,j] + img\_trans[i-(y-y0), j-(x-x0)]) / 2

############# Main ################

epsilon = 0.9

delta = 10

directory = "/home/xu1363/Documents/ECE 661/hw5/input/"

file1 = "1.jpg"

file2 = "2.jpg"

file3 = "3.jpg"

file4 = "4.jpg"

file5 = "5.jpg"

img1 = cv2.imread(directory+file1,cv2.IMREAD\_COLOR)

img2 = cv2.imread(directory+file2,cv2.IMREAD\_COLOR)

img3 = cv2.imread(directory+file3,cv2.IMREAD\_COLOR)

img4 = cv2.imread(directory+file4,cv2.IMREAD\_COLOR)

img5 = cv2.imread(directory+file5,cv2.IMREAD\_COLOR)

# resize the input image to a smaller size for acceleration

h = int(img1.shape[1]\*0.5)

w = int(img1.shape[0]\*0.5)

img1 = cv2.resize(img1, (h, w), interpolation = cv2.INTER\_AREA)

img2 = cv2.resize(img2, (h, w), interpolation = cv2.INTER\_AREA)

img3 = cv2.resize(img3, (h, w), interpolation = cv2.INTER\_AREA)

img4 = cv2.resize(img4, (h, w), interpolation = cv2.INTER\_AREA)

img5 = cv2.resize(img5, (h, w), interpolation = cv2.INTER\_AREA)

H12 = get\_homography\_all(img1, img2)

H23 = get\_homography\_all(img2, img3)

H43 = get\_homography\_all(img4, img3)

H54 = get\_homography\_all(img5, img4)

H13 = np.dot(H23,H12)

H53 = np.dot(H54, H43)

plot\_inliers\_outliers(img1, img2)

plt.axis('off')

plt.savefig("/home/xu1363/Documents/ECE 661/hw5/img12.jpeg")

plot\_inliers\_outliers(img2, img3)

plt.axis('off')

plt.savefig("/home/xu1363/Documents/ECE 661/hw5/img23.jpeg")

plot\_inliers\_outliers(img3, img4)

plt.axis('off')

plt.savefig("/home/xu1363/Documents/ECE 661/hw5/img34.jpeg")

plot\_inliers\_outliers(img4, img5)

plt.axis('off')

plt.savefig("/home/xu1363/Documents/ECE 661/hw5/img45.jpeg")

img13 = mapping(img1, H13)

img23 = mapping(img2, H23)

img43 = mapping(img4, H43)

img53 = mapping(img5, H53)

img\_all = [img1, img2, img4, img5]

H\_all = [H13, H23, H43, H53]

x, y, w, h = get\_panorama\_boundary(img3, img\_all, H\_all)

panorama = np.zeros((h, w, 3), dtype = np.uint8)

panorama[ 0-y : 0-y+img3.shape[0] , 0-x : 0-x+img3.shape[0], :] = img3

plot\_panorama(panorama, img1, img13, H13, x, y)

plot\_panorama(panorama, img5, img53, H53, x, y)

plot\_panorama(panorama, img2, img23, H23, x, y)

plot\_panorama(panorama, img4, img43, H43, x, y)

plt.figure(dpi=1200)

plt.imshow(cv2.cvtColor(panorama, cv2.COLOR\_BGR2RGB), cmap='jet')

plt.axis('off')

plt.savefig("/home/xu1363/Documents/ECE 661/hw5/panorama.jpeg")