Prateek Gupta

gupta198@purdue.edu

ECE 661 Homework 4

**Theory Questions**

1. Conceptually speaking, how do we differentiate between the inliers and the outliers when using RANSAC for solving the homography estimation problem using the interest points extracted from two different photos of the same scene?

To differentiate between inliers and outliers, we use RANSAC. This process involves utilizing linear regression and attempting to find a representation that encompasses most of the correspondences. Essentially, we start with a model and verify how many correspondences fit that model within some number of pixels 𝛿, and then update the model until we get the most amount of correspondences. We will not be able to find a model that has all the correspondences, since some will be an error, and our goal is to eliminate those.

1. As you will see in Lecture 12, the Gradient-Descent (GD) is a reliable method for minimizing a cost function, but it can be excruciatingly slow. At the other extreme, we have the much faster Gauss-Newton (GN) method, but it can be numerically unstable. Explain in your own words how the Levenberg-Marquardt (LM) algorithm combines the best of GD and GN to give us a method that is reasonably fast and numerically stable at the same time.

The Gradient-Descent method is very slow because it attempts to minimize the error by constantly adjusting the parameters in the opposite direction of the gradient. The Gauss Newton method does this in one step, but it is unstable, so it is not guaranteed to work. The LM method is the best to use since it takes the best features of both approaches by combining the two methods. It essentially uses gradient descent when the parameters are far away from the minimized value. But as the parameters get closer to optimal, the step sizes get smaller with gradient descent. So in order to use larger steps and get to the solution faster, LM acts like the Gauss-Newton method.

**Overview**

In this report we will stitch together five images of the same scene and construct a panorama. This will be done by utilizing the techniques from previous reports, as well as some new things. First, we will use the Harris corner algorithm to identify points of interest, and establish correspondences using the NCC algorithm. As we found, this yields a descent set, however, it is not 100% accurate. There are usually a decent number of outliers. The RANSAC algorithm helps find the inliers (within certain thresholds) and discards the outliers. This allows us to accurately have a set of correspondences to construct a homography with. The homography will be refined using the Linear Least-Squares method. Finally, the homography will be applied to every pixel in all the images and will be stitched together. Since many of the techniques are from previous reports, a lot of what is here will be taken from there.

**Harris Corner Detector**

A corner is defined as any pixel having varied gray levels in multiple directions. We will start by finding the corners on each image in a pair. In order to do this, we will convert the image into grayscale, so we can detect the changing gray values. Now, we will need to construct a Haar filter. This is done based on the scale σ we want to use. We will take the scale, multiply it by 4, then round up to the nearest even number. This will allow us to construct the filters for estimating the x-derivative dx and the y-derivative dy. Below is an example for σ = 1.2:

This means our Dx and Dy will be 6x6 (Note that Dx and Dy will be the filters to help estimate dx and dy). The filters will be constructed in the following manner:

We now need to apply each of these filters to an area around each pixel. This area is determined by multiplying σ by 5 and rounding up to the nearest odd number. So, in our case, we would convolve each filter to a 7x7 area around each pixel. We will let the result of the convolution be denoted as dx and dy respectively. We will then use this result to construct a matrix C. This will help us determine if that pixel is a corner.

Now, we will compute the Harris response. This will tell us whether the pixel is a corner or not.

Where:

We apply this to every pixel in the gray image and use this to construct a list of corners.

**Normalized Cross Correlation**

After running through the corner detection algorithm, we have a list of corners on each image in a pair. We now need to establish a correspondence between those corners and determine if they are interest points. Again, we will utilize the grayscale images for this. We compare the corners on each image, and if they exceed a certain threshold value, their correspondence can be confirmed. There are two techniques for this, however, we will use the Normalized Cross Correlation (NCC). It is explained by the equation below.

**Random Sample Consensus**

After running through the Harris corner detection and NCC, we have a decent set of correspondences. However, they are not perfect. There are many correspondences that are not correct and are outliers. Our goal is to eliminate the outliers and to find a model that gives the largest set of inliers. We will allow for a small threshold δ to fall within our model. So, by setting δ to 3 means we will allow for a deviation of 3 pixels from the homography estimate to the actual pixel location.

We will want to select the homography that yields the most inliers. However, we need a finite amount of trials to take. This value will be N. We will determine this value using the following

Where p is the probability that at least one of the trials will not have any outliers. We will set p to 0.99 so that we can ensure we get good homographies.

We need one final value before we can run through RANSAC. We need a minimum number M of trials to conduct. This will be computed using the following

Where *ntotal* is the total number of correspondences, and (1 – ϵ) is the probability of there being an inlier. Once we set these three values, we are ready to run through RANSAC. We will do N trials by randomly selecting a set of correspondences, compute the homography from those correspondences, and then seeing how many of the correspondences the homography represents, within δ pixels (using the Euclidian distance). We do this N times and take the homography that has the most inliers.

**Estimating the Homography**

The homography for each trial can be estimated in two ways. The first way is to use the least-squares method, and the second is to use the Levenburg-Marquardt Nonlinear Least-Squares. Each of these methods will be discussed below.

*Linear Least Squares Minimization*

Given a set of points that form a correspondence, x and x’, where and , we can set up a homography matrix using the following system.

We will let

, A = ,

To get the least square approximation, we use

This will allow us to plug into H and get the homography

*Levenburg-Marquardt Nonlinear Least-Squares*

LM can be explained by the following, where p is the same as above, and J is the Jacobian

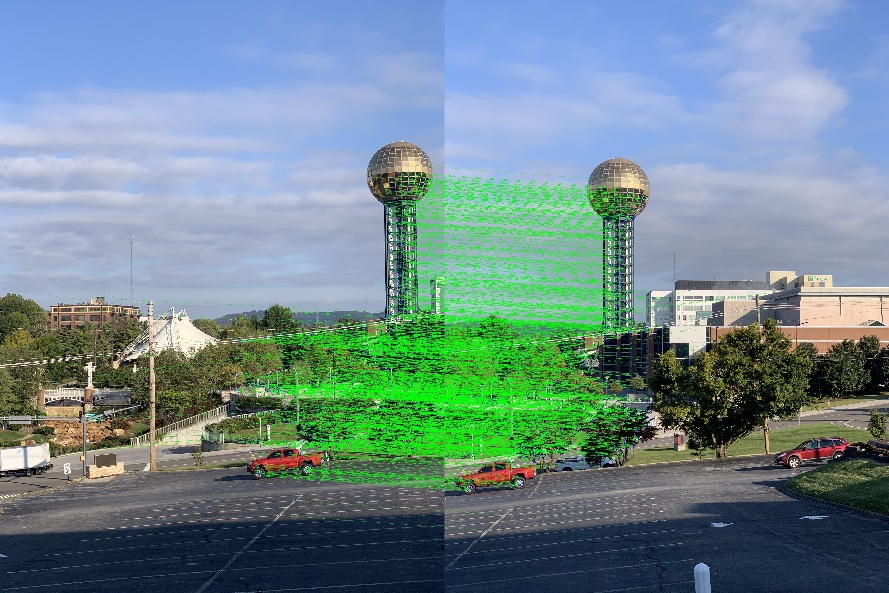
**Results**

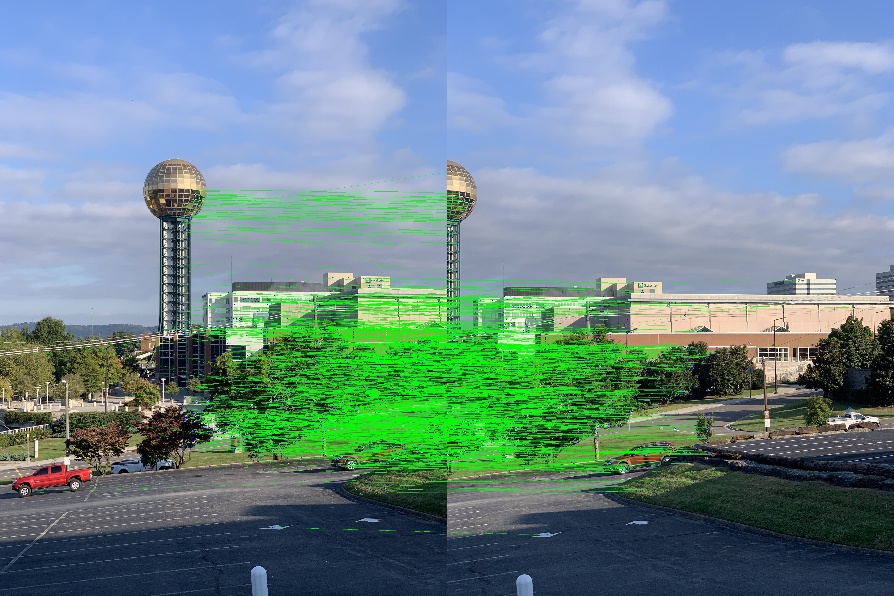
Input Images:

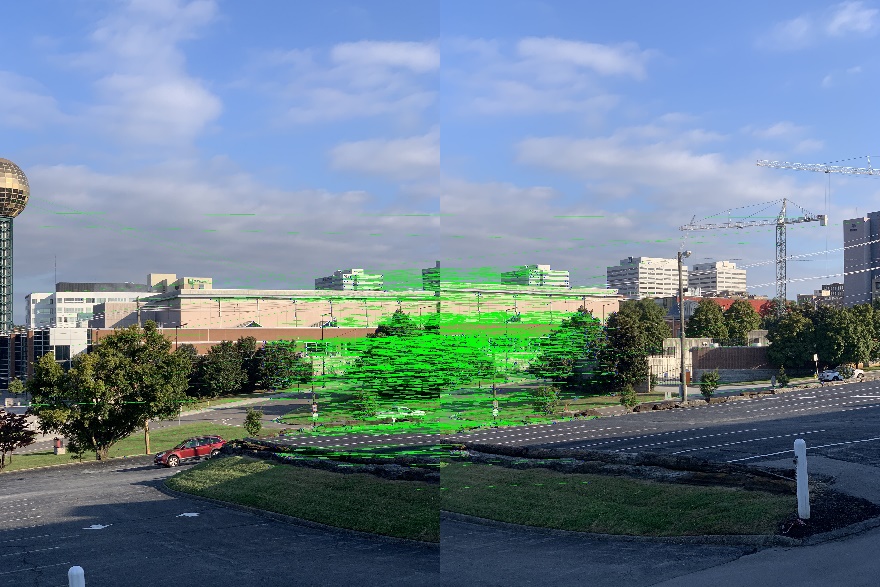
  

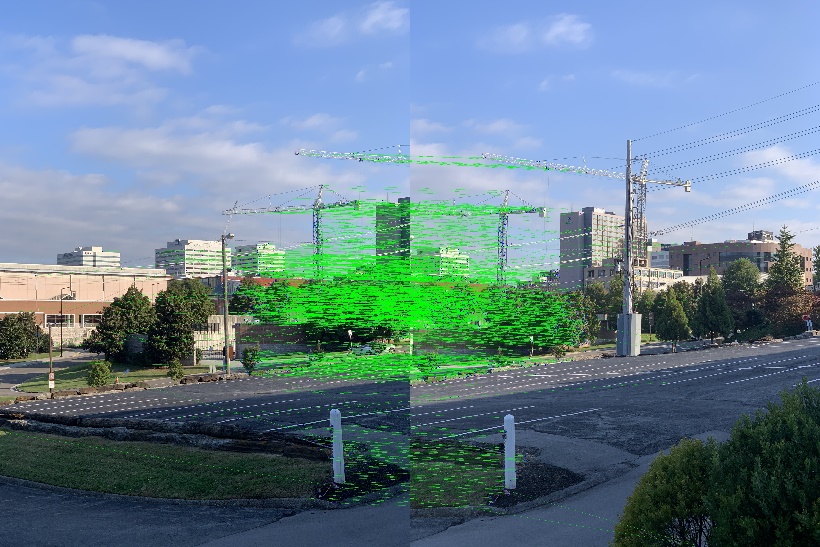
 

Match with outliers

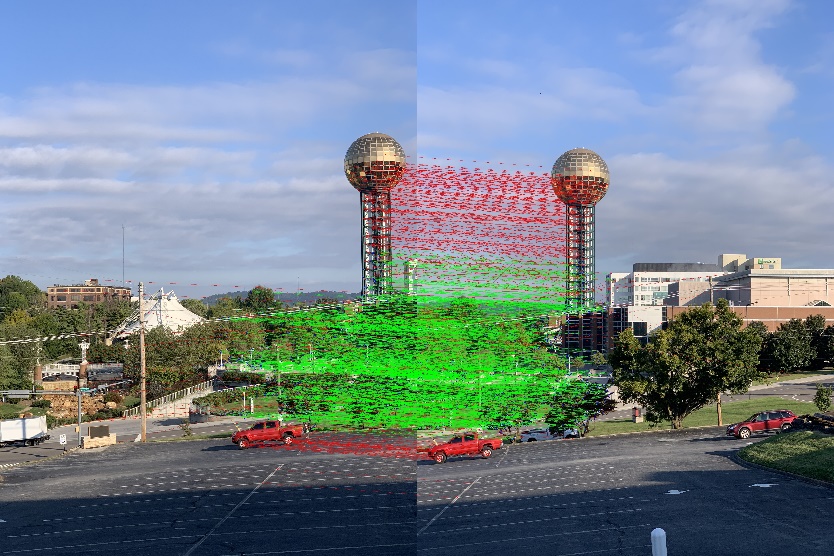


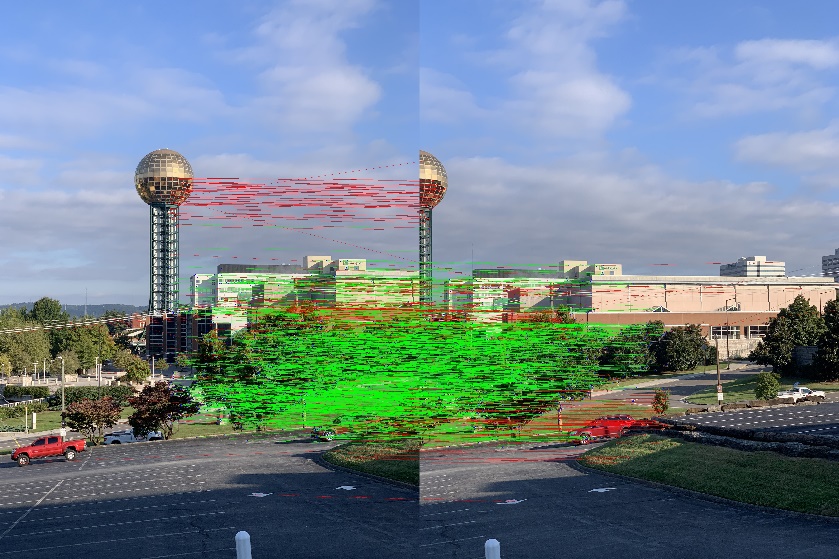






Match showing outlier and inliers:











import numpy as np

import cv2

import math

def harrisCorners(im, sigma):

dim = np.ceil(sigma \* 4)

if dim % 2 != 0:

dim+=1

oneMat = np.ones((int(dim/2), int(dim/2)), dtype=int)

negMat = -1 \* oneMat

D\_x = np.hstack((negMat, oneMat))

D\_x = np.vstack((D\_x, D\_x))

D\_y = np.vstack((oneMat, negMat))

D\_y = np.hstack((D\_y, D\_y))

dx = cv2.filter2D(im, -1, D\_x)

dy = cv2.filter2D(im, -1, D\_y)

C\_size = int(np.ceil(sigma \* 5))

if (C\_size % 2 == 0):

C\_size+=1

C\_size\_half = int(C\_size / 2)

threshold = np.zeros(im.shape)

for i in range(im.shape[0] - C\_size):

for j in range(im.shape[1] - C\_size):

dx\_kernal = dx[i:i+C\_size, j:j+C\_size]

dy\_kernal = dy[i:i+C\_size, j:j+C\_size]

dx2 = np.sum(np.square(dx\_kernal))

dy2 = np.sum(np.square(dy\_kernal))

dxdy = np.sum(np.multiply(dx\_kernal, dy\_kernal))

C\_det = np.subtract(np.multiply(dx2, dy2), np.square(dxdy))

C\_tr = np.add(dx2, dy2)

threshold[i + C\_size\_half, j + C\_size\_half] = C\_det - 0.05 \* np.square(C\_tr)

thresh\_size = 29

thresh\_size\_half = int(thresh\_size / 2)

thresholdAvg = np.mean(abs(threshold))

corners=[]

for i in range(im.shape[0]-thresh\_size):

for j in range(im.shape[1]-thresh\_size):

if threshold[i+thresh\_size\_half,j+thresh\_size\_half] > 0 and abs(threshold[i+thresh\_size\_half,j+thresh\_size\_half]) > thresholdAvg:

thresh = threshold[i:i+thresh\_size,j:j+thresh\_size]

if threshold[i+thresh\_size\_half,j+thresh\_size\_half] == np.max(thresh):

corners.append([i+thresh\_size\_half,j+thresh\_size\_half])

return np.array(corners)

def NCC(im1, im2, cnr1, cnr2):

NCC = np.zeros((len(cnr1), len(cnr2)))

for i in range(len(cnr1)):

for j in range(len(cnr2)):

f1 = im1[cnr1[i, 0] - 12 : cnr1[i, 0] + 13, cnr1[i, 1] - 12 : cnr1[i, 1] + 13]

f2 = im2[cnr2[j, 0] - 12 : cnr2[j, 0] + 13, cnr2[j, 1] - 12 : cnr2[j, 1] + 13]

ncc = np.sum(np.multiply(np.subtract(f1, np.mean(f1)), np.subtract(f2, np.mean(f2))))

ncc /= np.sqrt(np.sum(np.square(np.subtract(f1, np.mean(f1)))) \* np.sum(np.square(np.subtract(f2, np.mean(f2)))))

NCC[i, j] = ncc

pts=[]

for i in range(len(cnr1)):

for j in range(len(cnr2)):

if NCC[i,j]==np.max(NCC[i,:]) and NCC[i,j]>0.8:

pts.append([[cnr1[i,0],cnr1[i,1]],[cnr2[j,0],cnr2[j,1]]])

return np.array(pts)

def getHomography(im1Pts, im2Pts):

px = im1Pts[0, 1]

py = im1Pts[0, 0]

qx = im1Pts[1, 1]

qy = im1Pts[1, 0]

rx = im1Pts[2, 1]

ry = im1Pts[2, 0]

sx = im1Pts[3, 1]

sy = im1Pts[3, 0]

pxp = im2Pts[0, 1]

pyp = im2Pts[0, 0]

qxp = im2Pts[1, 1]

qyp = im2Pts[1, 0]

rxp = im2Pts[2, 1]

ryp = im2Pts[2, 0]

sxp = im2Pts[3, 1]

syp = im2Pts[3, 0]

t = np.array([[pxp],[pyp],[qxp],[qyp],[rxp],[ryp],[sxp],[syp]])

P = np.array([[px, py, 1, 0, 0, 0, -px\*pxp, -py\*pxp],

[0, 0, 0, px, py, 1, -px\*pyp, -py\*pyp],

[qx, qy, 1, 0, 0, 0, -qx\*qxp, -qy\*qxp],

[0, 0, 0, qx, qy, 1, -qx\*qyp, -qy\*qyp],

[rx, ry, 1, 0, 0, 0, -rx\*rxp, -ry\*rxp],

[0, 0, 0, rx, ry, 1, -rx\*ryp, -ry\*ryp],

[sx, sy, 1, 0, 0, 0, -sx\*sxp, -sy\*sxp],

[0, 0, 0, sx, sy, 1, -sx\*syp, -sy\*syp]])

pInv = np.linalg.inv(P)

h = pInv.dot(t)

H = np.array([[h[0, 0], h[1, 0], h[2, 0]], [h[3, 0], h[4, 0], h[5, 0]], [h[6, 0], h[7, 0], 1]])

return H

def testHomography(pts, H):

xp = np.linalg.matmul(H, np.append(pt[0], 1))

xp /= xp[-1]

dist = math.sqrt(np.sum((xp[0] - pt[1][0])\*\*2, (xp[1] - pt[1][1])\*\*2))

def RANSAC(pts):

pts1 = pts[:0]

pts2 = pts[:1]

N = 96

M = (len(pts)\*0.6)

maxNum = 0

bestH = np.zeros(3,3)

inliers = []

outliers = []

for i in range(N):

rand = np.random.randint(0,len(pts),6)

test1 = pts[0][rand]

test2 = pts[1][rand]

rand = np.random.randint(0,len(pts),6)

test3 = pts[0][rand]

test4 = pts[1][rand]

rand = np.random.randint(0,len(pts),6)

test5 = pts[0][rand]

test6 = pts[1][rand]

rand = np.random.randint(0,len(pts),6)

test7 = pts[0][rand]

test8 = pts[1][rand]

Htest = getHomography(np.array([test1, test3, test5, test7]), np.array([test2, test4, test6, test8]))

num = 0

tempInliers = []

tempOutliers = []

for j in range(pts):

dist = testHomography(pts[j], Htest)

if dist < 3:

num+=1

tempInliers.append(pts[j])

else:

tempOutliers.append(pts[j])

if num > maxNum:

maxNum = num

bestH = Htest

inliers = tempInliers

outliers = tempOutliers

return bestH, inliers, outliers

def showPoints(im1, im2, corners):

im = np.zeros((max(im1.shape[0], im2.shape[0]), im1.shape[1] + im2.shape[1],3))

im[0 : im1.shape[0], 0 : im1.shape[1]] = im1

im[0 : im2.shape[0], im1.shape[1] : im1.shape[1] + im2.shape[1]] = im2

for cnr in corners:

cv2.circle(im,(cnr[0, 1], cnr[0, 0]),4,(0,255,0),1)

cv2.circle(im,(im1.shape[1]+cnr[1, 1],cnr[1, 0]),4,(0,255,0),1)

cv2.line(im,(cnr[0, 1],cnr[0, 0]),(im1.shape[1]+cnr[1, 1],cnr[1, 0]), (0,255,0))

return im

def showInliers(im1, im2, inliers, outliers):

im = np.zeros((max(im1.shape[0], im2.shape[0]), im1.shape[1] + im2.shape[1],3))

im[0 : im1.shape[0], 0 : im1.shape[1]] = im1

im[0 : im2.shape[0], im1.shape[1] : im1.shape[1] + im2.shape[1]] = im2

for cnr in inliers:

cv2.circle(im,(cnr[0, 1], cnr[0, 0]),4,(0,255,0),1)

cv2.circle(im,(im1.shape[1]+cnr[1, 1],cnr[1, 0]),4,(0,255,0),1)

cv2.line(im,(cnr[0, 1],cnr[0, 0]),(im1.shape[1]+cnr[1, 1],cnr[1, 0]), (0,255,0))

for cnr in outliers:

cv2.circle(im,(cnr[0, 1], cnr[0, 0]),4,(0,0,255),1)

cv2.circle(im,(im1.shape[1]+cnr[1, 1],cnr[1, 0]),4,(0,0,255),1)

cv2.line(im,(cnr[0, 1],cnr[0, 0]),(im1.shape[1]+cnr[1, 1],cnr[1, 0]), (0,0,255))

images = ['Images/1.jpg', 'Images/2.jpg', 'Images/3.jpg', 'Images/4.jpg', 'Images/5.jpg']

im = []; imGray = []; corners = [];

sift = cv2.xfeatures2d.SIFT\_create(nfeatures=5000,nOctaveLayers=4,contrastThreshold=0.03,edgeThreshold=10,sigma=4)# may want to add these arguements

for img in images:

image = cv2.imread(img)

im.append(image)

imageGray = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

imGray.append(imageGray)

corners.append(harrisCorners(imageGray, 1.2))

intPts = []

for i in range(4):

intPts.append(NCC(im[i], im[i+1], corners[i], corners[i + 1]))

cv2.imwrite("Results/12.jpg", showPoints(im[0], im[1], intPts[0]))

cv2.imwrite("Results/23.jpg", showPoints(im[1], im[2], intPts[1]))

cv2.imwrite("Results/34.jpg", showPoints(im[2], im[3], intPts[2]))

cv2.imwrite("Results/45.jpg", showPoints(im[3], im[4], intPts[3]))

H12, inliers12, outliers12 = RANSAC(intPts[0])

cv2.imwrite("Results/inliers12.jpg", showInliers(im[0], im[1], inliers12, outliers12))

H23, inliers23, outliers23 = RANSAC(intPts[1])

cv2.imwrite("Results/inliers23.jpg", showInliers(im[1], im[2], inliers23, outliers23))

H34, inliers34, outliers34 = RANSAC(intPts[2])

cv2.imwrite("Results/inliers34.jpg", showInliers(im[2], im[3], inliers34, outliers34))

H45, inliers45, outliers45 = RANSAC(intPts[3])

cv2.imwrite("Results/inliers45.jpg", showInliers(im[3], im[4], inliers45, outliers45))

stitch1 = stitch(im[0], im[1], H12)

stitch2 = stitch(stitch1, im[2], H23)

stitch3 = stitch(stitch2, im[3], H34)

stitch4 = stitch(stitch3, im[4], H45)

cv2.imwrite("Results/Final.jpg", stitch4)