Home Work 3

Name : Sai Teja Chava
Course Name : Advanced Computer Vision(CSCI 677)
USC ID : 5551847788
Instructor Name : Prof. Ram Nevatia

SIFT:

Before going into SIFT, we need to identify the interesting, salient regions of an image that should be described and quantified. These regions are called key points and may correspond to edges, corners, or "blob"-like structures of an image.

After identifying the set of key points in an image, we then need to extract and quantify the region surrounding each key point. The feature vector associated with a key point is called a feature or local feature since only the local neighborhood surrounding the key point is included in the computation of the descriptor.

Many methods exist to use algorithms in extracting local features from an image and one of them is SIFT.

So how Does Shift Work?

After extracting key points, we need to describe and quantify the region of the image surrounding the key point.

The SIFT feature description algorithm requires a set of input key points. Then, for each of the input key points, SIFT takes the 16 x 16 pixel region surrounding the center pixel of the key point region. From there, we divide the 16 x 16 pixel region into sixteen 4 x 4 pixel windows. Then, for each of the 16 windows, we compute the gradient magnitude and orientation. Given the gradient magnitude and orientation, we next construct an 8-bin histogram for each of the 4 x 4 pixel windows. The amount added to each bin is dependent on the magnitude of the gradient. However, we are not going to use the raw magnitude of the gradient. Instead, we are going to utilize Gaussian weighting. The farther the pixel is from the key point center, the less it contributes to the overall histogram:

Finally, we need to collect all 16 of these 8-bin orientation histograms and concatenate them together.

Given that we have 16 of these histograms, our feature vector is of thus: 16 x 8 = 128-dim.

Once we have collected the concatenated histograms, we end up L2-normalizing the entire feature vector. At this point, our SIFT feature vector is finished and ready to be compared to other SIFT feature vectors.

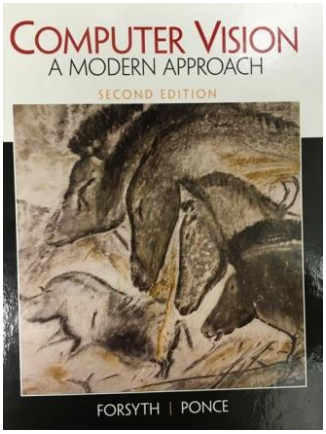
Steps in Code:

1. Read the source and destination images from the paths sent as arguments while the program is run.
2. Convert the images to grayscale.
3. Initiate the SIFT detector
4. Find the key points and descriptors with SIFT for source and destination images.
5. Find the orientation of all the key points
6. Initialize Brute Force Matcher and use knnMatch() function of Brute Force Matcher to get the best 'k' matches. k=2 so that Lowe's ratio test could be applied. Perform Lowe's ratio test to capture the good matches with a threshold of 0.7
7. Set condition that at least 10 matches need to be found
8. Calculate the homography with RANSAN fitting
9. Find the inliers and the outliers
10. Draw the final sift matching between the two images

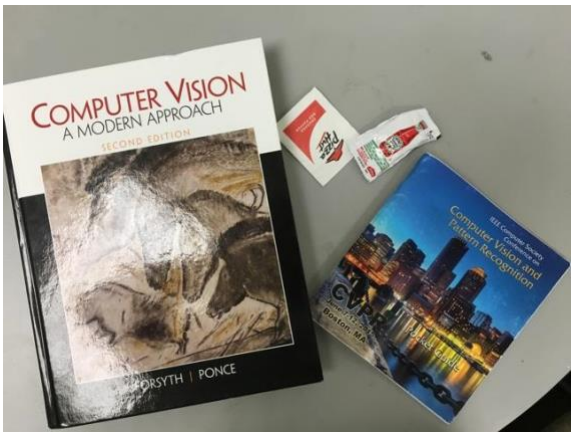
Results

QUERRY / DETECTION IMAGE	**TRAIN / TARGET IMAGE**	**QUERRY IMAGE FEATURES**	**TARGET IMAGE FEATURES**	**GOOD MATCHES: LOWE'S RATIO TEST**	**TOTAL NUMBER OF INLIERS (CONSISTENT WITH HOMOGRAPHY)**
src_1.jpg	dst_1.jpg	3249	1726	417	369
	dst_2.jpg	3249	1626	235	176
	dst_3.jpg	3249	1654	564	527
src_2.jpg	dst_1.jpg	2807	1726	197	168
	dst_2.jpg	2807	1626	210	189
	dst_3.jpg	2807	1654	47	35

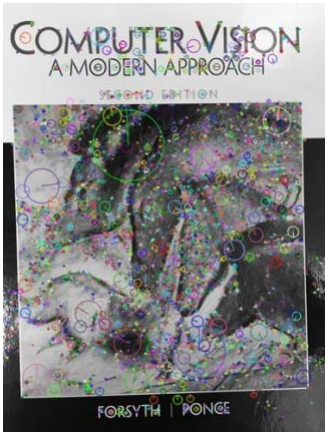
src_1.jpg and dst_1.jpg



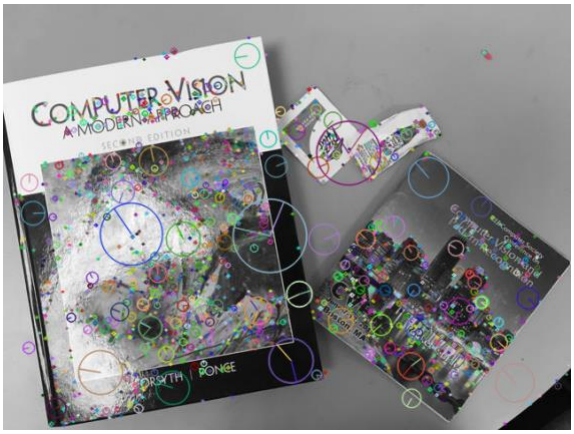
src_1.jpg



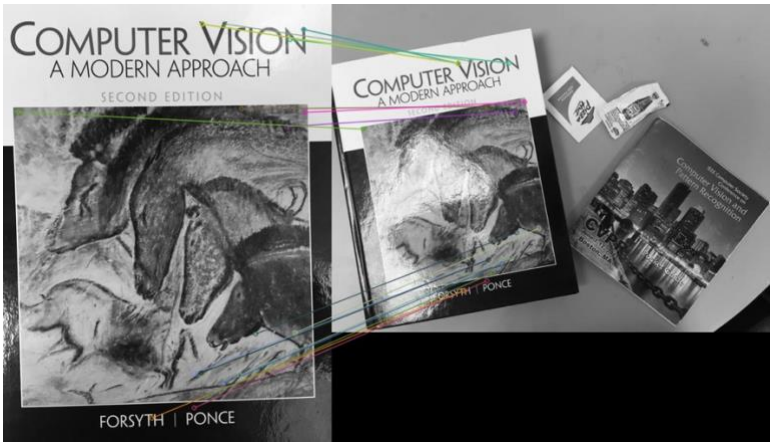
dst_1.jpg



src_1.jpg with Sift key points



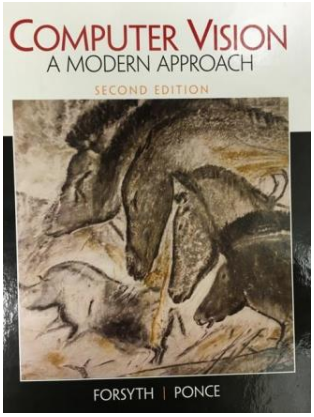
dst_1.jpg with Sift key points



Top 20 matches between src_1.jpg and dst_1.jpg

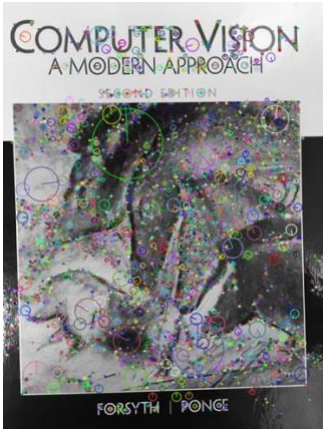
Final SIFT matches between src_1.jpg and dst_1.jpg

src_1.jpg and dst_2.jpg

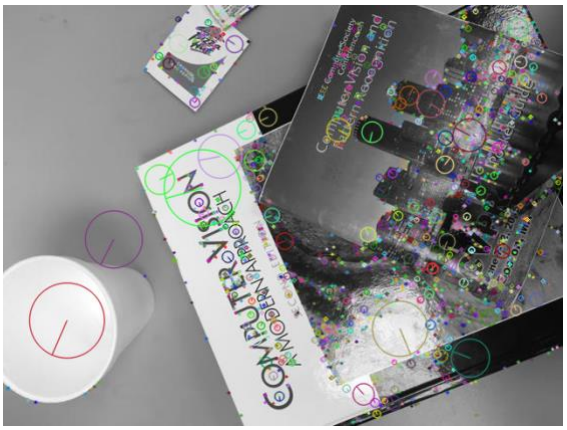


src_1.jpg

dst_2.jpg



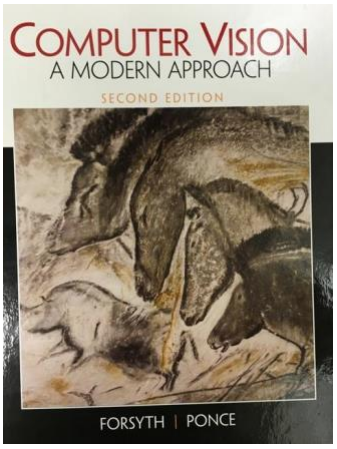
src_1.jpg with Sift key points


dst_1.jpg with Sift key points

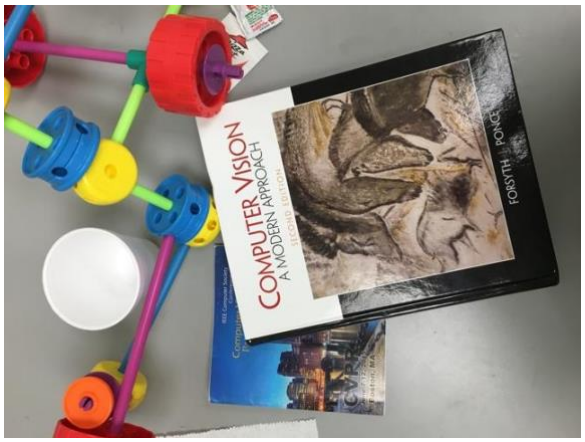
Top 20 matches between src_1.jpg and dst_2.jpg

Final SIFT matches between src_1.jpg and dst_2.jpg

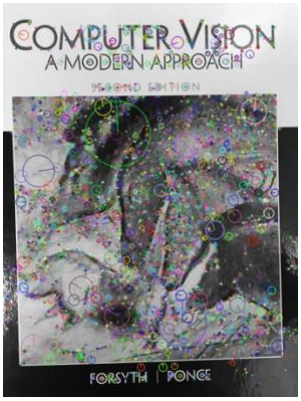
src_1.jpg and dst_3.jpg



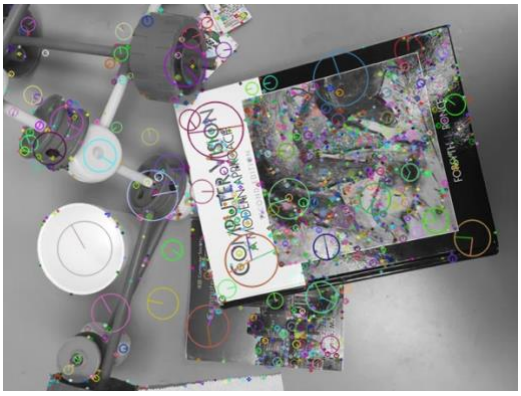
src_1.jpg



dst_3.jpg

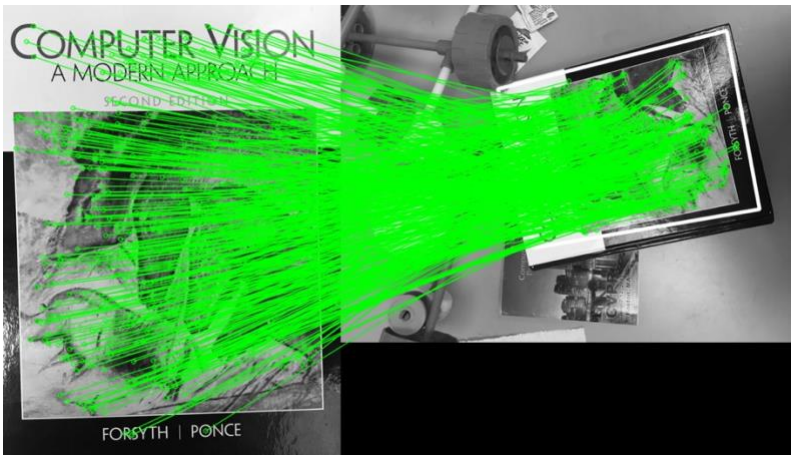


src_1.jpg with Sift key points



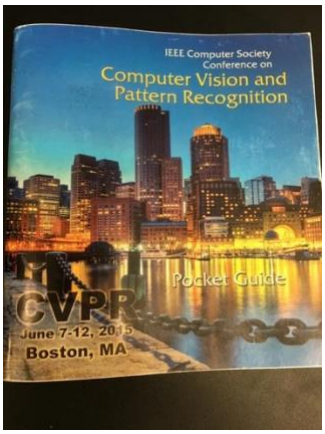
dst_1.jpg with Sift key points

Top 20 matches between src_1.jpg and dst_3.jpg

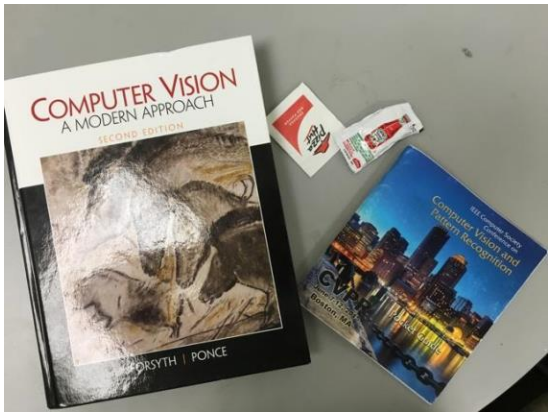


Final SIFT matches between src_1.jpg and dst_3.jpg

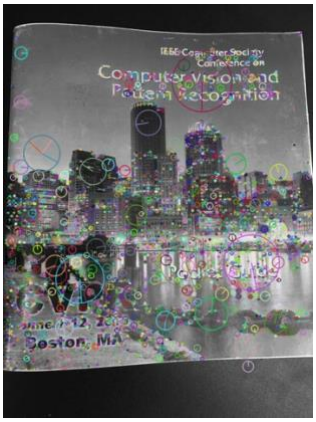
src_2.jpg and dst_1.jpg

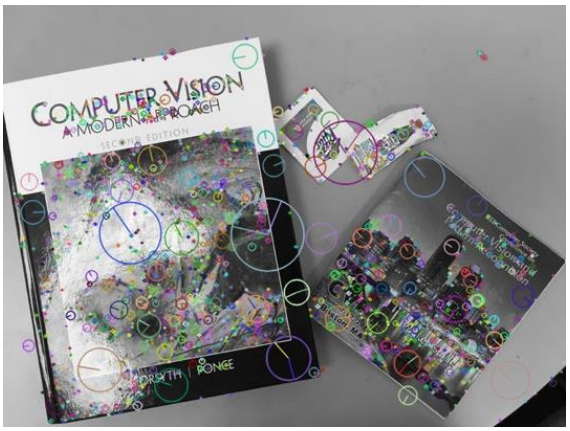


src_2.jpg

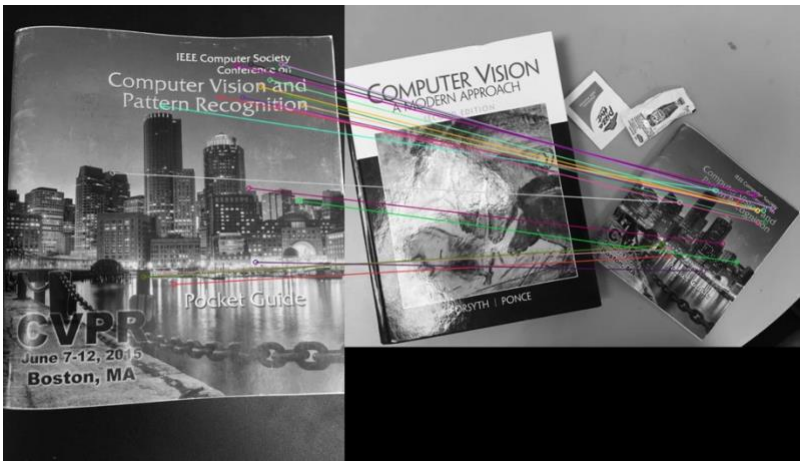


dst_1.jpg


src_1.jpg with Sift key points



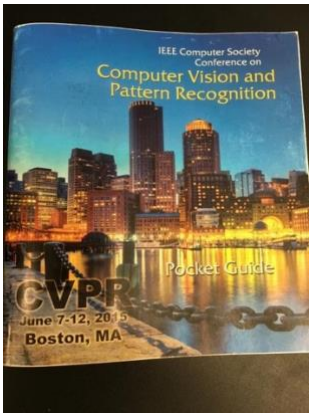
dst_1.jpg with Sift key points



Top 20 matches between src_2.jpg and dst_1.jpg

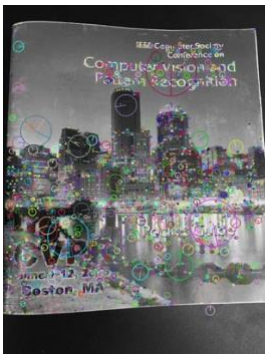
Final SIFT matches between src_2.jpg and dst_1.jpg

src_2.jpg and dst_2.jpg



src_2.jpg

dst_2.jpg



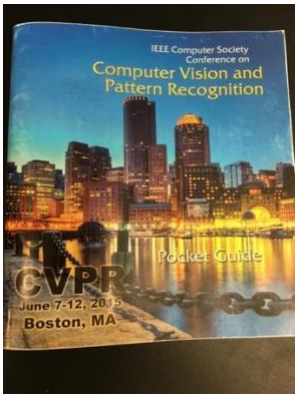
src_1.jpg with Sift key points

dst_1.jpg with Sift key points

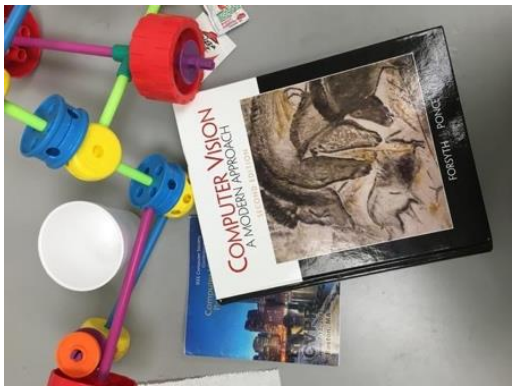
Top 20 matches between src_2.jpg and dst_2.jpg

Final SIFT matches between src_2.jpg and dst_2.jpg

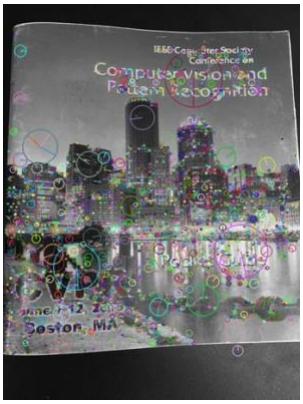
src_2.jpg and dst_3.jpg

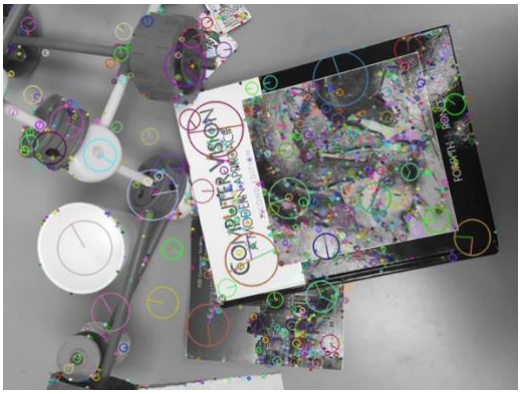


src_2jpg



dst_3.jpg


src_1.jpg with Sift key points


dst_1.jpg with Sift key points

Top 20 matches between src_2.jpg and dst_3.jpg

Final SIFT matches between src_2.jpg and dst_3.jpg

Analysis:

From the above images, the following conclusions can be drawn:

1. The features calculated for all the images are shown along with the orientation. These features were calculated based on different variations of object in different parts of the image like slant in text content, size of image parts, etc. and certain salient features.
2. In some images, like in dst_3.jpg, the object is being occluded by other objects. However, features were found in the part of the object that is not occluded and mapping happened.
3. Though the object was small (scale was less) in the target images, still SIFT was able to do very good job in finding the features and mapped to query image giving us proof that it is scale invariant.
4. In src_1.jpg and dst_1.jpg, even though there was a change in brightness/intensity i.e object was brighter in dst_1.jpg when compared to src_1.jpg, SIFT performed well.
5. Before applying homography, in the images where top 20 points are shown, some feature matching points were lost in the target image clutter and lead to some error. However, the error was less due to the use of knnMatch() present in brute force matcher followed by applying the Lowe's ratio test which reduced the matches to the best possible estimates based on the distance.
6. After applying homography using RANSAC, the errors found before applying the homography mentioned above got suppressed as RANSAC helped us in giving the best fit. Total number of inliers were found and on plotting those the correct feature match was achieved. So, from the total good matches approximately 90% of the matches fell under the inliers and 10% were outliers.

Inliers – Good matches
Outliers – Wrong matches

Code:

```
import numpy as np
import cv2
from matplotlib import pyplot as plt
import argparse

ap = argparse.ArgumentParser()
ap.add_argument("-d","--detect", required = True, type=str, help = "path to detect image")
ap.add_argument("-t","--target", required = True, type=str, help = "path to target image")
args = vars(ap.parse_args())

img_detect = args["detect"]
img_target = args["target"]

# Read the images.
img1   = cv2.imread(img_detect)
img2   = cv2.imread(img_target)
image1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY) # queryImage
image2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY) # trainImage

#Display the detect image
cv2.imshow('Detect',image1)
cv2.waitKey(0)

#Display the target image
cv2.imshow('Target',image2)
cv2.waitKey(0)

# Initiate SIFT detector
sift =  cv2.xfeatures2d.SIFT_create()
#sift =  cv2.SIFT()

# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(image1,None)
kp2, des2 = sift.detectAndCompute(image2,None)

print('The features found in querry image ' + img_detect + ': ' + str(des1.shape[0]))
print('The features found in train image ' + img_target + ': ' + str(des2.shape[0]))

# features overlaid on the images with circles based on size of keypoints and its orientation
img4 =
cv2.drawKeypoints(image1,kp1,image1,flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINT
S)
img5 =
cv2.drawKeypoints(image2,kp2,image2,flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINT
S)


#Show the query image with SIFT keypoints
cv2.imshow('Query image with SIFT keypoints',img4)
cv2.waitKey(0)
s1 = img_detect.split(".")
```

```
cv2.imwrite(s1[0] +' with SIFT keypoints'+'.png',img4)

#Display the train image with SIFT keypoints
cv2.imshow('Train image with SIFT keypoints',img5)
cv2.waitKey(0)
s2 = img_target.split(".")
cv2.imwrite(s2[0] +' with SIFT keypoints'+'.png',img5)

# BFMatcher with default params
bf = cv2.BFMatcher()
matches = bf.knnMatch(des1,des2, k=2)

# Apply ratio test
# store all the good matches as per Lowe's ratio test.
good_matches = []
for m,n in matches:
    if m.distance < 0.7*n.distance:
        good_matches.append(m)

# Sort the good matches in the order of their distance and plot the top 20 matches
good_matches = sorted(good_matches, key = lambda x:x.distance)
img6 = cv2.drawMatches(image1, kp1, image2, kp2, good_matches[:20], None, flags=2)
cv2.imshow('Top 20 matches between ' + img_detect + ' and ' + img_target, img6)
cv2.waitKey(0)
s3 = img_detect.split('/')
s4 = img_target.split('/')
cv2.imwrite('HW3_Data/'+'Top 20 matches between ' + s3[1] + ' and ' + s4[1] +'.png', img6)

# set that atleast 10 matches are to be there to find the objects
MIN_MATCH_COUNT = 10

# condition set that atleast 10 matches are to be there to find the objects
if len(good_matches)>MIN_MATCH_COUNT:
    src_pts = np.float32([kp1[m.queryIdx].pt for m in good_matches ]).reshape(-1,1,2)
    dst_pts = np.float32([kp2[m.trainIdx].pt for m in good_matches ]).reshape(-1,1,2)

    # compute the homography
    M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC,5.0)
    matchesMask = mask.ravel().tolist()

    print('The total number of good matches are: ' + str(mask.size))
    print('The total number of inliers (good matches providing correct estimation and total numbers consistent
with the computed homography)) are: ' + str(np.sum(matchesMask)))
    print('The homography matrix when ' + img_detect + ' is matched with ' + img_target + ':')
    print(M)

    # capture the height and width of the image
    h, w = image1.shape

    # If enough matches are found, we extract the locations of matched keypoints in both the images
    pts = np.float32([ [0,0],[0,h-1],[w-1,h-1],[w-1,0] ]).reshape(-1,1,2)

    # The locations of matched keypoints  are passed to find the perpective transformation
```

```
    dst = cv2.perspectiveTransform(pts,M)

    # Once we get this 3x3 transformation matrix, we use it to transform the corners of queryImage to
corresponding points in trainImage.
    # Then we draw it.
    image2 = cv2.polylines(image2,[np.int32(dst)],True,255,3, cv2.LINE_AA)

else:
    print ('Not enough matches are found - %d/%d' % (len(good),MIN_MATCH_COUNT))
    matchesMask = None

# Finally we draw our inliers (if successfully found the object) or matching keypoints (if failed)
draw_params = dict(matchColor=(0, 255, 0),  # draw matches in green color
                   singlePointColor=None,
                   matchesMask=matchesMask,  # draw only inliers
                   flags=2)

# draw the final SIFT matching
img3 = cv2.drawMatches(image1, kp1, image2, kp2, good_matches, None, **draw_params)
cv2.imshow('Final SIFT Match between ' + img_detect + ' and ' + img_target, img3)
cv2.imwrite('HW3_Data/'+'Final SIFT Match between '+ s3[1] + ' and ' + s4[1] +'.png',img3)

# wait for ESC key to exit
k = cv2.waitKey(0)
if k == 27:
    cv2.destroyAllWindows()
    plt.close('all')
```