Mujtaba Shahid Faizi

BSCS-5A

#131818

Lab 4 of Computer Vision

**All the explanations/results are mentioned in the code as comments.**

**TASK 1:**

**CODE:**

**import** cv2 **as** cv  
img = cv.imread(**'lena.png'**)  
gray= cv.cvtColor(img,cv.COLOR\_BGR2GRAY)  
sift = cv.xfeatures2d.SIFT\_create()  
kp = sift.detect(gray,**None**)  
*#it will draw a circle with size of keypoint and it will even show its orientation*img=cv.drawKeypoints(gray,kp,img,flags=cv.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS)  
cv.imshow(**'dst'**,img)  
**if** cv.waitKey(0) & 0xff == 27:  
 cv.destroyAllWindows()  
sift = cv.xfeatures2d.SIFT\_create()  
  
*#Here kp will be a list of keypoints and des is a numpy array of shape Number\_of\_Keypoints×128.  
# Here kp will be a list of keypoints and des is a numpy array of shape Number\_of\_Keypoints×128.*kp, des = sift.detectAndCompute(gray,**None**)  
print(**"Descriptors representation:"**,des)  
**for** a **in** kp:  
 print (a)

**EXPLANATION:**

The SIFT works as follows:

1. Scale-space Extrema Detection

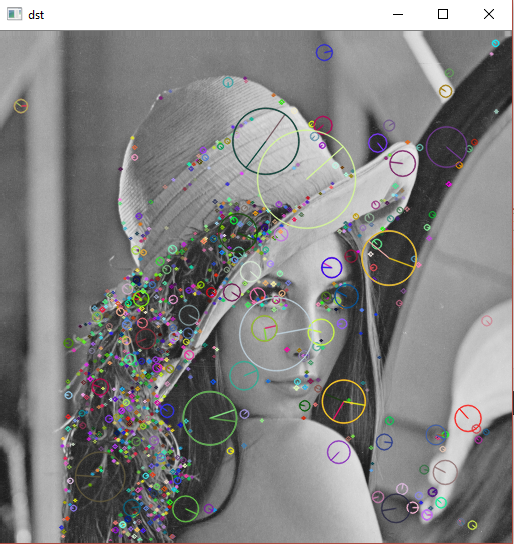
### 2. Keypoint Localization

### 3. Orientation Assignment

### 4. Keypoint Descriptor

### 5. Keypoint Matching

**SCREENSHOT:**

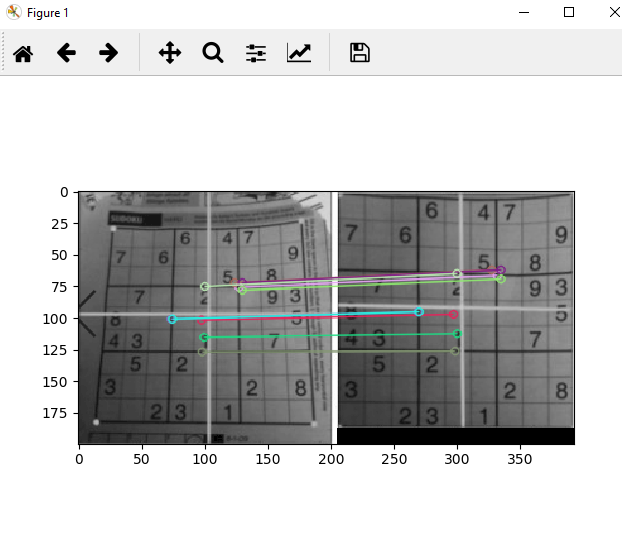


**TASK 2:**  
**CODE:**

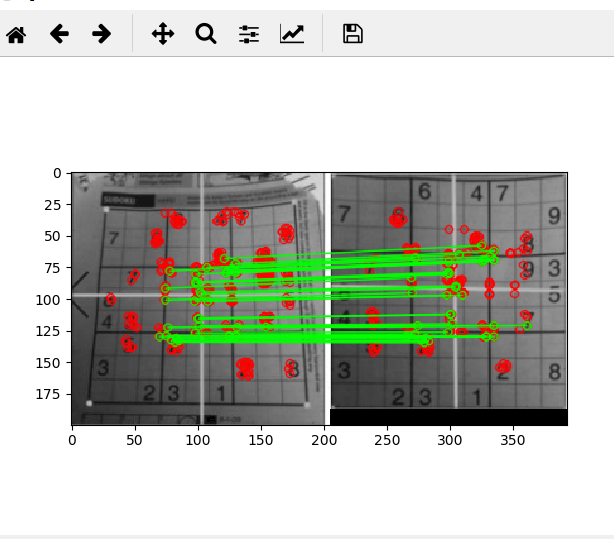
**import** cv2  
**from** matplotlib **import** pyplot **as** plt  
**import** numpy **as** np  
  
img1 = cv2.imread(**'1.PNG'**, 0) *# queryImage*img2 = cv2.imread(**'2.PNG'**, 0) *# trainImage  
  
# Initiate ORB detector*orb = cv2.ORB\_create()  
  
*# find the keypoints and descriptors with ORB*kp1, des1 = orb.detectAndCompute(img1, **None**)  
kp2, des2 = orb.detectAndCompute(img2, **None**)  
  
*#Brute-Force Matching  
#It takes the descriptor of one feature in first set and is matched with all other features in second set using some distance calculation. And the closest one is returned.  
#we create a BFMatcher object with distance measurement cv2.NORM\_HAMMING and crossCheck is switched on for better results*bf = cv2.BFMatcher(cv2.NORM\_HAMMING, crossCheck=**True**)  
  
*# Match descriptors.*matches = bf.match(des1, des2)  
  
*# Sort them in the order of their distance.*matches = sorted(matches, key=**lambda** x: x.distance)  
  
*# Draw first 10 matches.*img3 = cv2.drawMatches(img1, kp1, img2, kp2, matches[:10], **None**, flags=2)  
  
plt.imshow(img3), plt.show()  
  
*# FLANN based Matcher  
#FLANN stands for Fast Library for Approximate Nearest Neighbors. It contains a collection of algorithms optimized for fast nearest neighbor search in large datasets and for high dimensional features*FLANN\_INDEX\_KDTREE = 0  
index\_params = dict(algorithm = FLANN\_INDEX\_KDTREE, trees = 5)  
search\_params = dict(checks=50) *# or pass empty dictionary*flann = cv2.FlannBasedMatcher(index\_params,search\_params)  
  
matches=flann.knnMatch(np.asarray(des1,np.float32),np.asarray(des2,np.float32), 2)  
  
*# Need to draw only good matches, so create a mask*matchesMask = [[0,0] **for** i **in** range(len(matches))]  
  
*# ratio test as per Lowe's paper***for** i,(m,n) **in** enumerate(matches):  
 **if** m.distance < 0.7\*n.distance:  
 matchesMask[i]=[1,0]  
  
draw\_params = dict(matchColor = (0,255,0),  
 singlePointColor = (255,0,0),  
 matchesMask = matchesMask,  
 flags = 0)  
  
img3 = cv2.drawMatchesKnn(img1,kp1,img2,kp2,matches,**None**,\*\*draw\_params)  
  
plt.imshow(img3,),plt.show()

**SCREENSHOT:**

### Brute-Force Matching with ORB Descriptors with distance measurement cv2.NORM\_HAMMING



## **FLANN (Fast Library for Approximate Nearest Neighbors) based Matcher**



**TASK 3:**  
**CODE:**

**import** cv2  
**import** numpy **as** np  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 *# Read source image.* im\_src = cv2.imread(**'1.PNG'**)  
 *# Four corners of the book in source image* pts\_src = np.array([[141, 131], [480, 159], [493, 630], [64, 601]])  
  
 *# Read destination image.* im\_dst = cv2.imread(**'2.PNG'**)  
 *# Four corners of the book in destination image.* pts\_dst = np.array([[318, 256], [534, 372], [316, 670], [73, 473]])  
  
 *# Calculate Homography* h, status = cv2.findHomography(pts\_src, pts\_dst)  
  
 *# Warp source image to destination based on homography* im\_out = cv2.warpPerspective(im\_src, h, (im\_dst.shape[1], im\_dst.shape[0]))  
  
 *# Display images* cv2.imshow(**"Source Image"**, im\_src)  
 cv2.imshow(**"Destination Image"**, im\_dst)  
  
 *# Print estimated homography* print(**"Estimated homography : \n"**, h)  
 cv2.waitKey(0)

**SCREENSHOT:**

