**from** **PIL** **import** Image, ImageDraw

**import** **numpy** **as** **np**

**import** **csv**

**import** **math**

**import** **random**

**def** ReadKeys(image):

#No changes

**def** AppendImages(im1, im2):

#No changes

**def** DisplayMatches(im1, im2, matched\_pairs):

#No changes

**def** match(image1, image2):

*"""Input two images and their associated SIFT keypoints.*

*Display lines connecting the first 5 keypoints from each image.*

*Note: These 5 are not correct matches, just randomly chosen points.*

*The arguments image1 and image2 are file names without file extensions.*

*Returns the number of matches displayed.*

*Example: match('scene','book')*

*"""*

im1, keypoints1, descriptors1 = ReadKeys(image1)

im2, keypoints2, descriptors2 = ReadKeys(image2)

*#*

*# REPLACE THIS CODE WITH YOUR SOLUTION (ASSIGNMENT 5, QUESTION 3)*

*#*

*# Threshold is between 0 and 1*

threshold = 0.91 #0.96 used for library match

**print** "Threshold = ", threshold

*# Number of rounds of RANSAC*

rounds = 100

**print** "Rounds = ", rounds

*# List of pairs of keypoints matched in both images*

matched\_pairs = []

*# Match to keep track of the indexes (values) of a given angle (key)*

match = {}

*# Find values and their respective angles*

**for** rowim1 **in** xrange(len(descriptors1)):

**for** rowim2 **in** xrange(len(descriptors2)):

*# Angle between the descriptors*

*# to measure the similarity between them*

angle = math.acos(np.dot(descriptors1[rowim1],

descriptors2[rowim2]))

match[angle] = (rowim1, rowim2)

*# Sort our matches to find the smalles ones*

sorted\_match = sorted(match)

*# Save the two best matches*

first\_neighbor = sorted\_match[0]

second\_neighbor = sorted\_match[1]

*# If first match is sufficiently close (close is determined by theshold) to a second*

*# minimum match, then we keep it in matched\_pairs*

**if** first\_neighbor/second\_neighbor < threshold:

key1, key2 = match[first\_neighbor]

matched\_pairs.append([keypoints1[key1], keypoints2[key2]])

*# Reset match to be empty for next iteration*

match = {}

*# Call RANSAC over the matched\_pairs*

matched\_pairs = ransac(matched\_pairs, keypoints1, keypoints2, rounds)

*#*

*# END OF SECTION OF CODE TO REPLACE*

*#*

im3 = DisplayMatches(im1, im2, matched\_pairs)

**return** im3

**def** ransac(matched\_pairs, keypoints1, keypoints2, rounds):

*"""*

*Applies the RANSAC algorithm to the matched\_pairs in order to*

*reduce the amount of false positives.*

*Input:*

*matched\_pairs the matched keypoints pairs calculated by match()*

*keypoints1 the keypoints on the first image*

*keypoints2 the keypoints on the second image*

*rounds the amount of subsets to be generated and compared*

*Returns: best\_matched\_pairs the best keypoints subset after the given rounds*

*"""*

*# Keypoints of the best subset*

best\_matched\_pairs = []

*# Temporary subset keypoints pairs used in our loop*

subset\_pairs = []

*# # of times to run the algorithm*

**for** i **in** range(rounds):

*# Matched keypoins 1 and 2 chosen randomly from*

*# the original matched\_pairs*

key\_rand1, key\_rand2 = random.choice(matched\_pairs)

subset\_pairs.append([key\_rand1, key\_rand2])

*# Compare random match with others*

**for** key1, key2 **in** matched\_pairs:

*# Row, Columm, Scale and Orientation for the random pair*

\_, \_, scale\_rand1, o\_rand1 = key\_rand1

\_, \_, scale\_rand2, o\_rand2 = key\_rand2

*# Row, Columm, Scale and Orientation for the others*

\_, \_, scale1, o1 = key1

\_, \_, scale2, o2 = key2

*# Keep keypoints with scale and orientation that are*

*# close to our random match*

**if**(math.fabs(o\_rand1-o\_rand2-math.pi/6) < math.fabs(o1-o2) **and**

math.fabs(o\_rand1-o\_rand2+math.pi/6) > math.fabs(o1-o2) **and**

math.fabs(scale\_rand1-scale\_rand2)\*0.5 < math.fabs(scale1-scale2) **and**

math.fabs(scale\_rand1-scale\_rand2)\*1.5 > math.fabs(scale1-scale2)):

*#Add to our temp list of pairs*

subset\_pairs.append([key1, key2])

*# Keep the larger set of pairs*

**if**(len(subset\_pairs) > len(best\_matched\_pairs)):

best\_matched\_pairs = subset\_pairs

*# Set subset\_pairs back to empty for next*

subset\_pairs = []

**return** best\_matched\_pairs

*#Test run...*

*#match('scene', 'book')*

*#match('library2', 'library')*

match('scene', 'box')

Q3: Used the threshold 0.91 as it gave the right balance of correct matches with a few outliers. The reason it is important to have the right balance is because any less (<0.9) results in a lot of missed matches while going up to ~0.95 results in too many false matches. 0.91 essentially allowed the best angled matches.

Q4: Increasing the threshold causes matches between points less strict and hence results in quite a few outliers. The reason why it works so well in this case of the multiple runs we go through. The “consistency checking” is more forgiving because it knows that overtime (dependent on the # of runs), the eventual best subset will come out. There is a chance that the absolute best points aren’t taken into the algorithm due to us picking a randomly matched pair but it works based on looking at it from the averages point of view.