CPSC 425 Assignment 4

Jakob Khalil 86176674

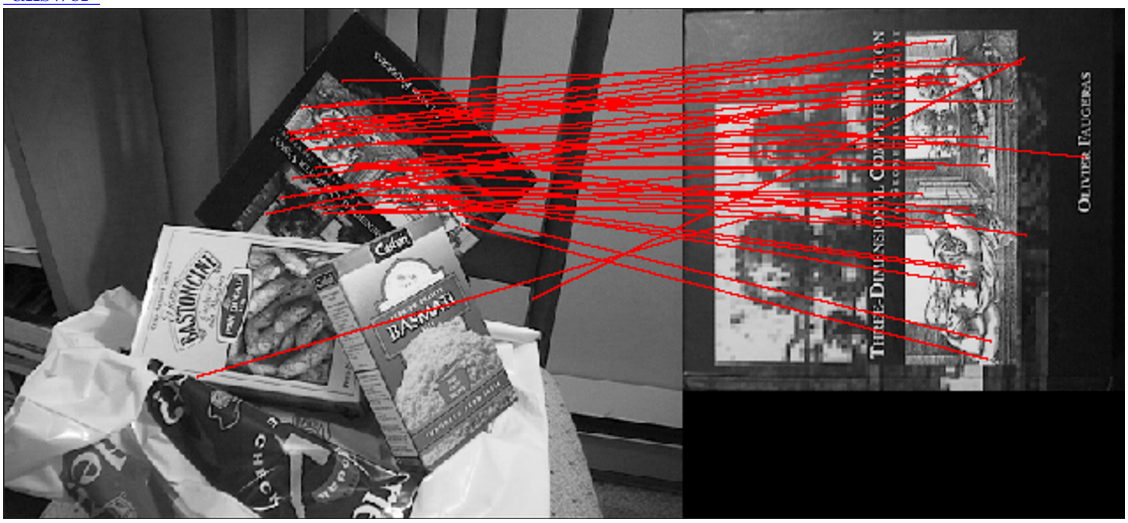
March 20, 2024

Question 1 - SIFT Keypoint Matching

I. (11 points) The sample program, main_match.py, loads two images and their invariant keypoints and then draws 5 lines between randomly selected keypoints to show how matches can be displayed. Your task is to improve this program so that it identifies and displays correct matches by comparing the keypoint descriptor vectors. Note: Your program should find all possible matches, not just 5 as shown in this sample.

Select a threshold that gives mostly good matches for the images scene and book. Try different thresholds so that only a small number of outliers are found (less than about 10). You can judge this by eye, as the outliers will usually produce matching lines at clearly different angles from the others. Print the box image showing the set of matches to the scene image for your suggested threshold value. Write a short description of the particular threshold value used, why you chose it and how important it was to get the value correct.

answer



I chose a threshold of 0.7 because it gives many good results at the expense of only 2 outliers. I found this threshold by starting at Lowe's recommendation of 0.8, and varying around that point. Larger thresholds did not produce many more inliers, but gave many more outliers. Smaller threhsholds didn't give me as many inliers as I wanted. I balanced the results by maximizing inliers, with a reasonable number of outliers. Choosing a good threshold is important because too many outliers will have a negative impact on our next steps, such as placing the book over the scene in the correct orientation.

```
def FindBestMatches(descriptors1, descriptors2, threshold):
    """
    This function takes in descriptors of image 1 and image 2,
    and find matches between them. See assignment instructions for details.
    Inputs:
        descriptors: a K-by-128 array, where each row gives a descriptor
        for one of the K keypoints.  The descriptor is a 1D array of 128
        values with unit length.
        threshold: the threshold for the ratio test of "the distance to the nearest"
                   divided by "the distance to the second nearest neighbour".
                   pseudocode-wise: dist[best_idx]/dist[second_idx] <= threshold
    Outputs:
        matched_pairs: a list in the form [(i, j)] where i and j means
                       descriptors1[i] is matched with descriptors2[j].
    """
    assert isinstance(descriptors1, np.ndarray)
    assert isinstance(descriptors2, np.ndarray)
    assert isinstance(threshold, float)
    ## START
    ## the following is just a placeholder to show you the output format
    # num = 5
```

```
# matched_pairs = [[i, i] for i in range(num)]

# Assumption: all descriptor vectors are already normalized.
# 'pairwise_cos_angles' is the pairwise cos angle of descriptors1 and descriptors2.
# For notation, let x1 be the first descriptor of descriptors1, and y1 be same
# but for descriptors 2.
# 'pairwise_cos_angles' ends up being the dot-products:
# [[x1y1, x1y2, x1y3, ...],
#  [x2y1, x2y2, x2y3, ...],
#  [...                  ], ...]
pairwise_cos_angles = descriptors1 @ descriptors2.T
pairwise_angles = np.arccos(pairwise_cos_angles)

# Each index of candidate_pairs corresponds to x_i. candidate_pairs[i] = y_j with
# best match.
candidate_pairs = np.argmin(pairwise_angles, axis=1)
candidate_pairs_angle = np.min(pairwise_angles, axis=1)

descriptors_1_index_best_match = np.arange(0, candidate_pairs.shape[0])
descriptors_2_index_best_match = candidate_pairs

# setting the best pairs to have infinite angle so that I can get the second best
# descriptor pair with min and argmin.
pairwise_angles_without_best_pairs = pairwise_angles.copy()
pairwise_angles_without_best_pairs[
    descriptors_1_index_best_match,
    descriptors_2_index_best_match
] = np.inf

second_place_pairs_angle = np.min(pairwise_angles_without_best_pairs, axis=1)
best_vs_second_best_ratios = candidate_pairs_angle / second_place_pairs_angle

# Only take pairs who's ratio beats the threshold.
best_i = descriptors_1_index_best_match[best_vs_second_best_ratios <= threshold, None]
best_j = descriptors_2_index_best_match[best_vs_second_best_ratios <= threshold, None]
matched_pairs = np.concatenate((best_i, best_j), axis=1)

## END
return matched_pairs.tolist()
```

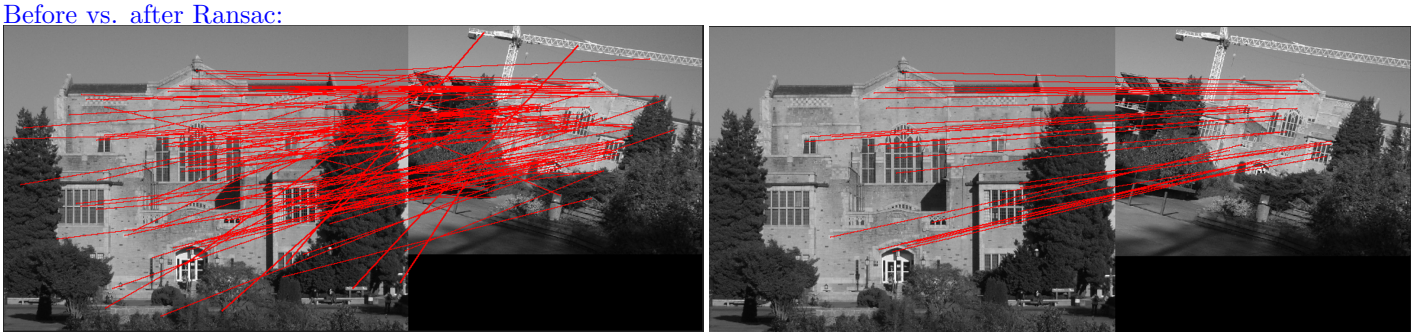
II. (14 points) mplement the RANSAC routine to filter out false matches in the function RANSACFilter. In details, for each RANSAC iteration you will select just one match at random, and then check all the other matches for rotation and scale consistency with it.

Try different values for the orientation and scale agreement (instead of using 30 degrees and 50% as mentioned above), and raise the matching threshold to get as many correct matches as possible while having only a few false matches. Try getting the best possible results on matching the difficult UBC library images.

Include resulting library images showing your best set of matches in a PDF. Aslo include in a PDF a paragraph summarizing the effects of consistency checking and the degree to which it allowed you to raise the matching threshold.

answer

Before vs. after Ransac:



Chosen hyper-parameters:

- ratio threshold = 0.8
- orientation agreement = 25 degrees
- scale agreement = 1%

Consistency checking produced significantly better correspondence results by allowing me to increase the Lowe's Ratio threshold, from 60% to 80%. Although SIFT produced far more outliers with the higher threshold, RANSAC allowed me to prune those outliers from the concensus set. The result was that having a higher threshold produced more significantly more inliers at the expense of outliers. However that was not a problem because those outliers were discarded by our consistency-checking filter.

I tested a variety of ratio thresholds to find decent results without the use of RANSAC. I concluded that 60% was reasonable as the consistency seemed to line-up with my post-RANSAC results. However I will note that there were significantly less correspondences than when I used RANSAC accompanied by an 80% threshold.

```
def RANSACFilter(matched_pairs, keypoints1, keypoints2, orient_agreement, scale_agreement):
    """
    This function takes in `matched_pairs`, a list of matches in indices
    and return a subset of the pairs using RANSAC.
    Inputs:
        matched_pairs: a list of tuples [(i, j)],
            indicating keypoints1[i] is matched
            with keypoints2[j]
        keypoints1, 2: keypoints from image 1 and image 2
            stored in np.array with shape (num_pts, 4)
            each row: row, col, scale, orientation
        *_agreement: thresholds for defining inliers, floats
    Output:
        largest_set: the largest consensus set in [(i, j)] format

    HINTS: the "*_agreement" definitions are well-explained
           in the assignment instructions.
    """
```

```
assert isinstance(matched_pairs, list)
assert isinstance(keypoints1, np.ndarray)
assert isinstance(keypoints2, np.ndarray)
assert isinstance(orient_agreement, float)
assert isinstance(scale_agreement, float)
## START
# Let's use numpy because numpy lists suck lol.
matched_pairs = np.array(matched_pairs)

# These orderd arrays correspond to the indices specified in matched_pairs.
# ex. 'matched_keypoints1_ordered[0]' is keypoint1 of matched_pairs[0],
#     'matched_keypoints1_ordered[1]' is keypoint1 of matched_pairs[1],
#                                                          etc...
# Doing this makes computing faster and easier using numpy.
matched_keypoints1_ordered = keypoints1[matched_pairs[:,0]]
matched_keypoints2_ordered = keypoints2[matched_pairs[:,1]]

#  Each row of of keypoints provides 4 numbers for a keypoint specifying
# [location_y, location_x, scale, orientation] in the original image.
#      0           1        2          3
# Pre-computing these values is faster and easier using numpy.
match_orientation_differences = (
    matched_keypoints2_ordered[:, 3] - matched_keypoints1_ordered[:, 3]
)

# Two keypoints have scales s1 and s2, the change in scale is defined as s2/s1.
# Pre-computing these values is faster and easier using numpy.
match_change_in_scales = (
    matched_keypoints2_ordered[:, 2] / matched_keypoints1_ordered[:, 2]
)

# Repeat the random selection 10 times and then select the largest consistent
# subset that was found.
largest_set = np.array([])
largest_set_size = largest_set.shape[0]
for _ in range(10):
    # randint is inclusive, so -1 to keep index in range.
    sample_match_index = random.randint(0, matched_pairs.shape[0] - 1)
    sample_match_orientation_diff = match_orientation_differences[sample_match_index]
    sample_match_change_in_scale = match_change_in_scales[sample_match_index]

    # "Assume that we are sampling with replacement and that the final consensus set should
    # include the initial match". I don't need to add it here because later, when we check this
    # match against itself, it obviously be within the agreement thresholds and get added to the
    # consensus set.

    # Check that the change of orientation between the two keypoints of each match agrees within,
    # say, 30 degrees.
    # From ReadKeys(...):
    # keypoints: K-by-4 array, in which each row has the 4 values specifying a keypoint
    # (row, column, scale, orientation). The orientation is in the range [-PI, PI] radians.
    # I need to convert orient_agreement into radians lmao.
    orient_radian_tolerance = np.deg2rad(orient_agreement)
    match_orientation_agreements = (
        # sample_orient - tolerance <= match_orient <= sample_orient + tolerance
        np.logical_and(
            sample_match_orientation_diff - orient_radian_tolerance <= match_orientation_differences,
            match_orientation_differences <= sample_match_orientation_diff + orient_radian_tolerance
        )
```

```
    )

    # As an example, if the change of scale for our first match is 3, then to be within
    # a consensus set, the second match must have change in scale within the range of
    # 1.5 to 4.5 (assuming scale agreement of plus or minus 50%)
    sample_match_change_in_scale_tolerance = scale_agreement * sample_match_change_in_scale
    match_scale_agreements = (
        # Check that the change of scale agrees within plus or minus, say, 50%.
        # sample_change_in_scale - tolerance <= match_orient <= sample_change_in_scale + tolerance
        np.logical_and(
            sample_match_change_in_scale - sample_match_change_in_scale_tolerance <= match_change_in_scal
            match_change_in_scales <= sample_match_change_in_scale + sample_match_change_in_scale_toleran
        )
    )

    # "For this assignment, we won't check consistency of location, as that is more difficult."
    # so I'm basing my consensus set purely off orientation and scale.
    is_part_of_consensus = np.logical_and(
        match_orientation_agreements,
        match_scale_agreements
    )

    consensus_set = matched_pairs[is_part_of_consensus]
    consensus_set_size = consensus_set.shape[0]

    if (consensus_set_size > largest_set_size):
        largest_set = consensus_set
        largest_set_size = consensus_set_size

largest_set = largest_set.tolist()

## END
assert isinstance(largest_set, list)
return largest_set
```

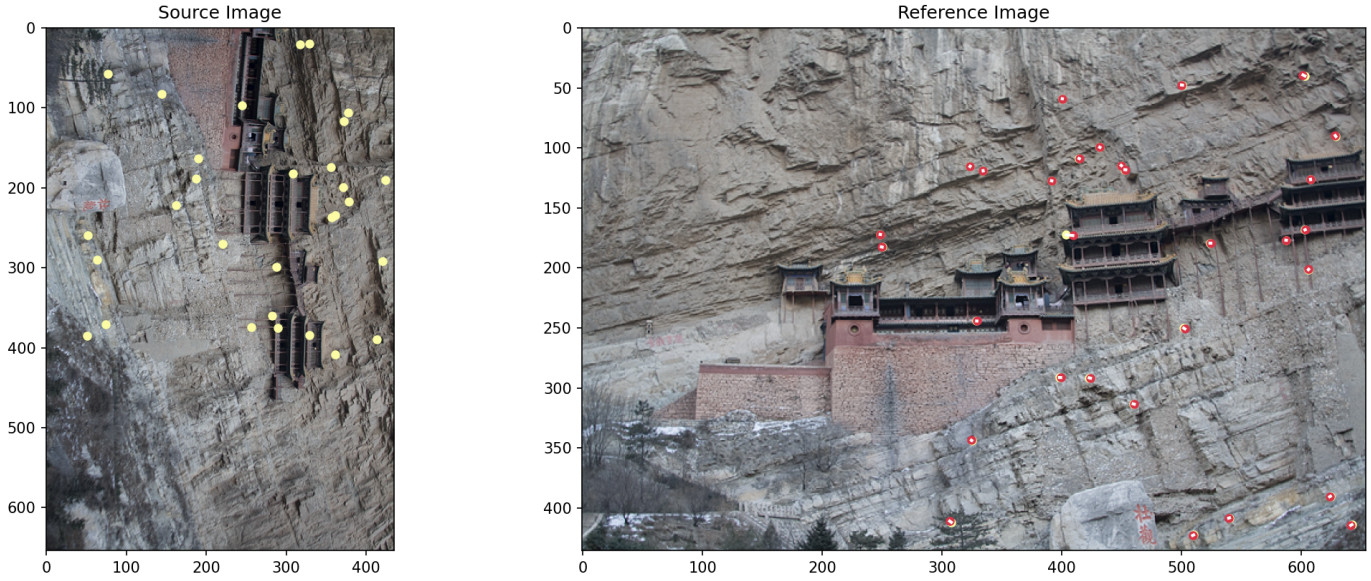
Question 2 - Panorama

I. (9 points) Keypoint Projections: To check if a match is an inlier, we have to know how to project 2d points from source image frame to the reference frame using a homography matrix. Your task is to implement the function KeypointProjection(xy_points, h). This function takes in an array of 2d points xy_points (in xy format) and project the point using the homography matrix h.

The bright-yellow dots are the points in the source figure. Their projections are plotted in the reference image with the same color. The red dots are the corresponding reference points (remember, we have matches). If your implementation is reasonable, you should obtain a figure similar to this one, where the red and yellow dots overlap a lot.

answer

Resulting test figure:




```python
def KeypointProjection(xy_points, h):
    """
    This function projects a list of points in the source image to the
    reference image using a homography matrix `h`.
    Inputs:
        xy_points: numpy array, (num_points, 2)
        h: numpy array, (3, 3), the homography matrix
    Output:
        xy_points_out: numpy array, (num_points, 2), input points in
        the reference frame.
    """
    assert isinstance(xy_points, np.ndarray)
    assert isinstance(h, np.ndarray)
    assert xy_points.shape[1] == 2
    assert h.shape == (3, 3)

    # START
    # First convert the 2d points to homogeneous coordinate.
    number_of_points = xy_points.shape[0]
    homogenous_row = np.ones((1, number_of_points))
    xy_points_homogenous = np.concatenate((xy_points.T, homogenous_row), axis=0)

    # Perform the projection by a matrix multiplication
    homogenous_projected_points = h @ xy_points_homogenous
```

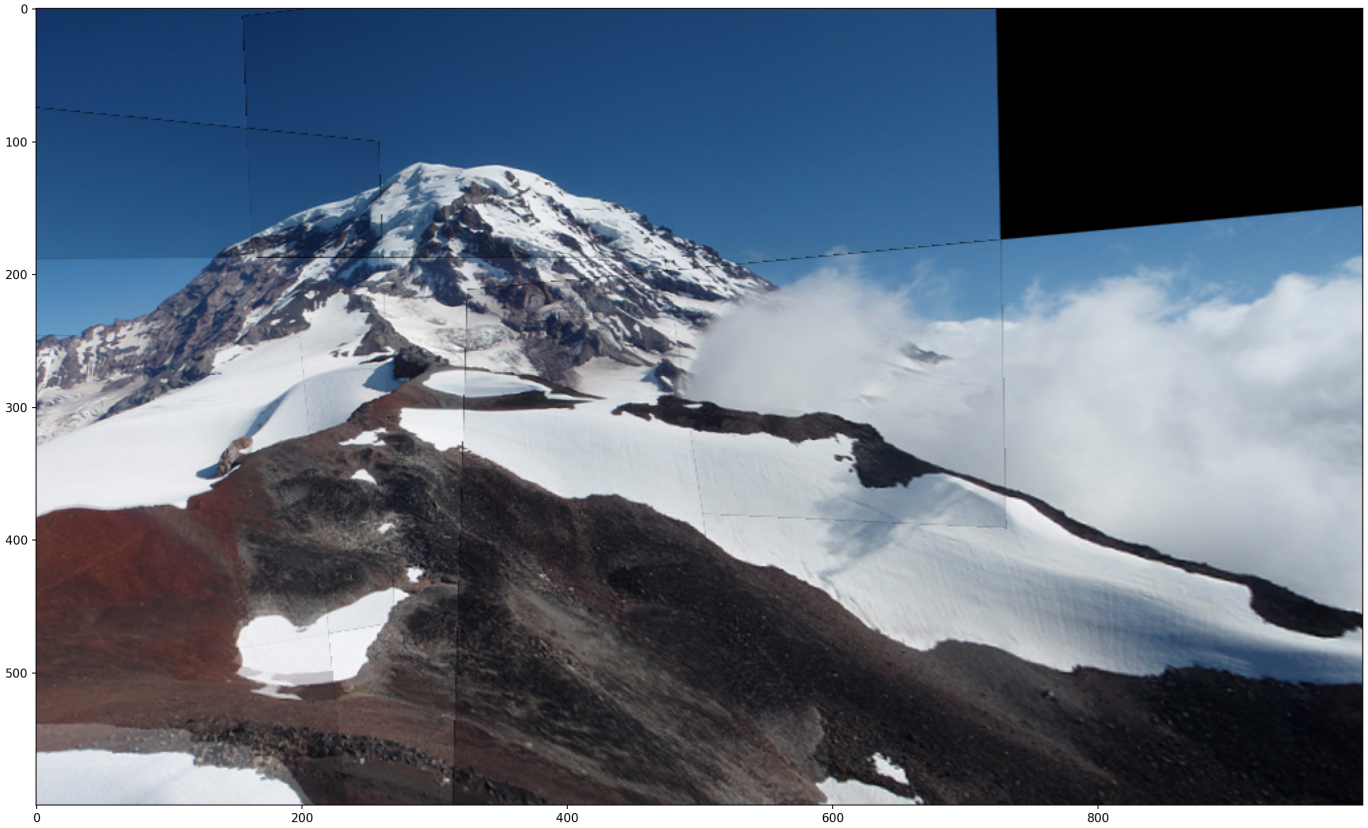
```
# If the extra dimension is zero, you should replace it with
# 1e-10 to avoid dividing by zero.
extra_dim_is_zero = homogenous_projected_points[2] == 0
homogenous_projected_points[2, extra_dim_is_zero] = 1e-10

# Convert back the projected points in homogeneous coordinate to
# the regular coordinate by dividing through the extra dimension.
projected_points = homogenous_projected_points[:2] / homogenous_projected_points[2]
xy_points_out = projected_points.T

# END
return xy_points_out
```

II. (13 points) RANSAC Homography: Now that you can project 2d points from the source images to the reference image using a homography matrix. The next step is to use RANSAC to find the biggest consensus set of matches to compute the final homography matrix. In the function RANSACHomography(xy_src, xy_ref, num_iter, tol), the parameters xy_src and xy_ref store the xy coordinates of matches between a source image and a reference image. The matches are from FindBestMatches.

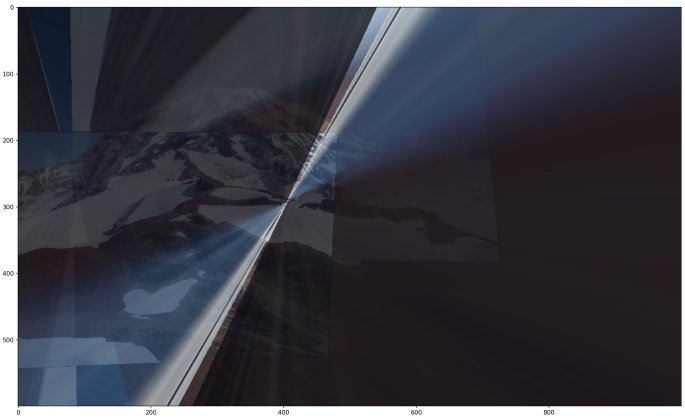
You can change various parameters such as num_iter, tol, ratio_thres in the main_pano.py. Experiment with different combination of num_iter and tol. How does each of the two parameters contributes to the final panorama result? Include the discussion and the created panoramas in your report.

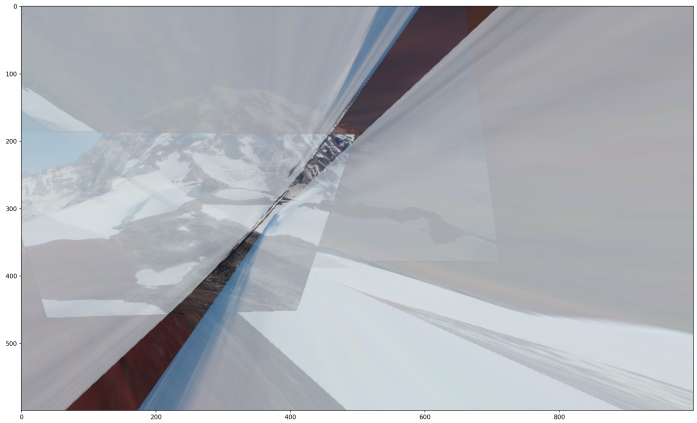
answer

Panorama result (num_iter=50, tol=10):



Some other results with (num_iter=50, tol=100) and (num_iter=500, tol=100) respectively:





Tolerance contributes to the accuracy of computed projection matrix. If our tolerance is too high, then our results start to become warped/distorted as we allow more outliers to negatively influence our results. Having a smaller tolerance assures that a majority of our consensus points do in-fact get projected to the reference coordinates by our hypothesis transformation. However if the tolerance is too low, then we may not have enough correspondences to generate a projec-

tion matrix. That outcome also depends on factors the scope of this hyper-parameter though.

Number of iterations controls our number of RANSAC trials. The stochastic nature of using randomized anchor points means that there is always a possibility that we end up with artifacts/errors in our final results. Increasing the number of trials reduces the probability that we have random errors in our final result. The number of desired trials can probably be calculated as a function of the error rate that we may find acceptable.

```
def RANSACHomography(xy_src, xy_ref, num_iter, tol):
    """
    Given matches of keyponit xy coordinates, perform RANSAC to obtain
    the homography matrix. At each iteration, this function randomly
    choose 4 matches from xy_src and xy_ref.  Compute the homography matrix
    using the 4 matches.  Project all source "xy_src" keypoints to the
    reference image.  Check how many projected keyponits are within a `tol`
    radius to the coresponding xy_ref points (a.k.a. inliers).  During the
    iterations, you should keep track of the iteration that yields the largest
    inlier set. After the iterations, you should use the biggest inlier set to
    compute the final homography matrix.
    Inputs:
        xy_src: a numpy array of xy coordinates, (num_matches, 2)
        xy_ref: a numpy array of xy coordinates, (num_matches, 2)
        num_iter: number of RANSAC iterations.
        tol: float
    Outputs:
        h: The final homography matrix.
    """
    assert isinstance(xy_src, np.ndarray)
    assert isinstance(xy_ref, np.ndarray)
    assert xy_src.shape == xy_ref.shape
    assert xy_src.shape[1] == 2
    assert isinstance(num_iter, int)
    assert isinstance(tol, (int, float))
    tol = tol*1.0

    # START
    number_of_matches = xy_src.shape[0]
    largest_set_size = 0
    largest_src = np.array([])
    largest_ref = np.array([])
    h = None
    for _ in range(num_iter):
        # Randomly pick 4 matches of points. Make sure sample_matches
        # are unique.
        sample_match_indices = []
        while len(sample_match_indices) < 4:
            sample_index = random.randint(0, number_of_matches - 1)
            if sample_index not in sample_match_indices:
                sample_match_indices.append(sample_index)

        sample_src = xy_src[sample_match_indices]
        sample_ref = xy_ref[sample_match_indices]

        # Compute a homography matrix.
        h, _ = cv2.findHomography(sample_src, sample_ref)

        # Project all keypoints in the source image to the reference
        # image using the computed homography matrix.
        projected_points = KeypointProjection(xy_src, h)
```

```
    # Compute the Euclidean distance between each projected point
    # to its correspondance in the reference frame.
    euclidean_dists = np.linalg.norm(projected_points - xy_ref, ord=2, axis=1)

    # Summing a boolean vector counts how many satisfy the
    # tolerance condition.
    is_in_consensus = euclidean_dists <= tol
    consensus_src = xy_src[is_in_consensus]
    consensus_ref = xy_ref[is_in_consensus]
    consensus_set_size = consensus_src.shape[0]

    # If, for a match, the projected point gives a distance
    # no more than tol, the match is considered an inlier.
    if (consensus_set_size > largest_set_size):
        largest_set_size = consensus_set_size
        largest_src = consensus_src
        largest_ref = consensus_ref

# At the end of the loop, you compute and return the final
# homography matrix using the largest consensus set.
h, _ = cv2.findHomography(largest_src, largest_ref)

# END
assert isinstance(h, np.ndarray)
assert h.shape == (3, 3)
return h
```

III. (3 points) Asumming that you've implemented the panorama program correctly, lets apply it to some UBC images! We've collected photos of some landmarks of UBC taken from the fountain, the rose garden (both lower level and upper level), and our Irving library. You can find the images and features under the data folder.

Please produce the three panorama named fountain40.png, garden034.png, and irving_out365.png. The order of digits on each photo indicates the order in your panorama program. For example, irving_out_365.png denotes that your reference image is irving_out3.png, and you project irving_out6.png and irving_out5.png into the reference frame. Include these results into your report.

answer
thresholds: 0.9, 0.7, 0.9
iterations: 500, 500, 500
tolerances: 10, 5, 5
respectively