Q1: SIFT Keypoint Matching

Question 1

Contents of solution.py

In [2]:

```
import numpy as np
import cv2
import math
import random
```

1.4

In details, for each RANSAC iteration you will select just one match at random, and then check all the other matches for consistency with it.

1. Repeat the random selection 10 times and then select the largest consistent subset that was found. Assume that we are sampling with replacement and that the final consensus set should include the initial match, i.e., the size of consistency set is at minimum 1.
2. keypoints1 and keypoints2 arrays that are provided for each image. Each row provides 4 numbers for a keypoint specifying its location, scale, and orientation in the original image
3. you should check that the change of orientation between the two keypoints of each match agrees within, say, 30 degrees.The difference between the two matches in this case is 180 degrees and would be outside the 30 degree tolerance, hence the second match is not part of the consensus set. Note that orientation is measured in radians, and that orientations are equal modulo 2 pi.
4. Also, check that the change of scale agrees within plus or minus, say, 50%. Assuming two keypoints have scales s1 and s2, the change in scale is defined as s2/s1 (see slides). 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%). For this assignment, we won't check consistency of location, as that is more difficult.

   Test: 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.

In [199]:

```
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
#     Repeat the random selection 10 times
#     and then select the largest consistent subset that was found.
    collect = [] # collect list of subsets
    for i in range(10):
        subset = matched_pairs.copy()
        # keypoints1 and keypoints2 arrays (x,y,scale,orientation)
        # random selection
        randompair = random.choice(matched_pairs)
        randomkp1 = keypoints1[randompair[0]]
        randomkp2 = keypoints2[randompair[1]]

        delta_scale = randomkp2[2]/randomkp1[2]
        delta_angle = randomkp1[3] - randomkp2[3]

        # compare with initial match
#         print(len(matched_pairs))
        for j in range(len(matched_pairs)):
#             print(j)
            initial = matched_pairs[j]
            # repeat find change
            kp1 = keypoints1[initial[0]]
            kp2 = keypoints2[initial[1]]

            delta_iniscale = kp2[2]/kp1[2]
            delta_iniangle = kp1[3] - kp2[3]

            # compare angle with tolerance
            anglediff = (abs(delta_angle-delta_iniangle)) % 360
            if anglediff > orient_agreement:
                subset.remove(initial)
            # compare scale with tolerance (1+- aggrement)
            elif ((1 - scale_agreement) > delta_iniscale/delta_scale or delta_iniscale/delta_s
                subset.remove(initial)

```

```
        collect.append(subset)


    # largest array in collect
    maxlen = len(collect[0])
    i = -1
    index = 0
    for s in collect:
        i+=1
        if len(s) > maxlen:
            maxlen = len(s)
            index = i

    largest_set = collect[index]


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

1.3

The function FindBestMatches in the file solution.py takes the invariant descriptor vectors stored in the numpy arrays descriptors1 for the first image and descriptors2 for the second, and returns a list of index pairs of the matches.

1. Each row corresponds to a descriptor vector. To select the best match for a vector from the first image, you should measure its angle to each vector from the second matrix. As the descriptor vectors are already normalized to have unit length, the angle between them is the inverse cosine (math.acos(x) function in Python) of the dot product of the vectors. The vector with the smallest angle is the nearest neighbor (i.e., the best match).
2. To eliminate false matches, the most effective method is to compare the smallest (best) match angle to the second-best angle. A match should be selected only if this ratio is below a threshold. Hints: The Python function sorted can be used to find the two smallest values in a list. The list method index can then be used to determine the (original) indices of these sorted elements.

Test: 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.

In [ ]:

```
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
    matched_pairs = []
    ## the following is just a placeholder to show you the output format
    for i in range(len(descriptors1)):
        angles = []
        for j in range(len(descriptors2)):
            angle = math.acos(np.dot(descriptors1[i],descriptors2[j]))
            angles.append(angle)

        # The vector with the 1st, 2nd smallest angles
#         first_smallest = min(angles)
#         angles.remove(first_smallest)
#         sec_smallest = min(angles)
        initial_angles = angles
        first_smallest = sorted(angles)[0]
        sec_smallest = sorted(angles)[1]
        # A match should be selected only if this ratio is below a threshold
        ratio = first_smallest/sec_smallest
        if (ratio < threshold):
            index = initial_angles.index(first_smallest) # index = j
            matched_pairs.append([i, index])
    ## END
    return matched_pairs
```

2.3

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`.

1. You first convert the 2d points to homogeneous coordinate, perform the projection by a matrix multiplication,
2. and convert back the projected points in homogeneous coordinate to the regular coordinate by dividing through the extra dimension. If the extra dimension is zero, you should replace it with `1e-10` to avoid dividing by zero.

In [217]:

```
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
    # make xy_points 3D
    row_num = len(xy_points)
    add_col = np.ones((row_num,1))
    xyz_points = np.hstack((xy_points,add_col))
    # matrix projection
    xy_points_out = np.dot(h,xyz_points.transpose()).transpose()
#     print(xy_points_out.shape)
    # cover back
    for i in range(row_num):
        x = xy_points_out[i][0]
        y = xy_points_out[i][1]
        z = xy_points_out[i][2]
        # check z and set it to 1e-10 if it is 0
        if z == 0:
            z = 1e-10
        xy_points_out[i] = [x/z,y/z,1.0]
#     print(xy_points_out.shape)
    xy_points_out = xy_points_out[:,0:2]
    # END
    return xy_points_out
```

2.4

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.

1. Specifically, you run RANSAC for `num_iter` times. At each iteration, you randomly pick 4 matches of points to compute a homography matrix.
2. Then you project all keypoints in the source image to the reference image using the computed homography matrix.
3. You compute the Euclidean distance between each projected point to its correspondance in the reference frame. If, for a match, the projected point gives a distance no more than `tol`, the match is considered an inlier. The consensus set of this iteration is the set of inliers.
4. Across iterations, you will keep track of the largest consensus set, and at the end of the loop, you compute and return the final homography matrix using the largest consensus set.

In [248]:

```
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 anrdomly
    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
    #two array has same size
    num = range(xy_ref.shape[0])
    collect = []
    for i in range(num_iter):
        nnum = list(num)
        subset = nnum
      # randomly choose 4 matches from xy_src and xy_ref
      # find 4 random numbers as index
        random_index = random.sample(nnum,4)
        random_src = np.float32([[xy_src[i][0],xy_src[i][1]] for i in random_index])
        random_ref = np.float32([[xy_ref[i][0],xy_ref[i][1]] for i in random_index])
        h1,_ = cv2.findHomography(random_src,random_ref)
        # projection all keypoints in the source image(src)
        proj = KeypointProjection(xy_src, h1)
        # Euclidean distance between each projected point
        #to its correspondance in the reference frame
        for j in num:
            # np.linalg.norm(np_c1 - np_c2)
            e_dis = np.linalg.norm(proj[j] - xy_ref[j])
            if e_dis > tol:
                subset.remove(j)
        collect.append(subset)

    # largest array in collect
    maxlen = len(collect[0])
    i = -1
    index = i
    for s in collect:
        i+=1
        if len(s) > maxlen:
```

```
            maxlen = len(s)
            index = i

    largest_set = collect[index]

    #print(largest_set)
    # find represent items in array
    largest_src = np.float32([[xy_src[i][0],xy_src[i][1]] for i in largest_set])
    largest_ref = np.float32([[xy_ref[i][0],xy_ref[i][1]] for i in largest_set])
    # find h
    h,_ = cv2.findHomography(largest_src,largest_ref)

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


def FindBestMatchesRANSAC(
        keypoints1, keypoints2,
        descriptors1, descriptors2, threshold,
        orient_agreement, scale_agreement):
    """
    Note: you do not need to change this function.
    However, we recommend you to study this function carefully
    to understand how each component interacts with each other.

    This function find the best matches between two images using RANSAC.
    Inputs:
        keypoints1, 2: keypoints from image 1 and image 2
            stored in np.array with shape (num_pts, 4)
            each row: row, col, scale, orientation
        descriptors1, 2: 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
        orient_agreement: in degrees, say 30 degrees.
        scale_agreement: in floating points, say 0.5
    Outputs:
        matched_pairs_ransac: a list in the form [(i, j)] where i and j means
        descriptors1[i] is matched with descriptors2[j].
    Detailed instructions are on the assignment website
    """
    orient_agreement = float(orient_agreement)
    assert isinstance(keypoints1, np.ndarray)
    assert isinstance(keypoints2, np.ndarray)
    assert isinstance(descriptors1, np.ndarray)
    assert isinstance(descriptors2, np.ndarray)
    assert isinstance(threshold, float)
    assert isinstance(orient_agreement, float)
    assert isinstance(scale_agreement, float)
    matched_pairs = FindBestMatches(
        descriptors1, descriptors2, threshold)
    matched_pairs_ransac = RANSACFilter(
        matched_pairs, keypoints1, keypoints2,
        orient_agreement, scale_agreement)
    return matched_pairs_ransac
```

In [225]:

```
# a = [1,2,0,4,5,6]
# b = min(a)
# a.remove(b)
# print(a)
# min(a)
# matched_pairs = [[i, i] for i in range(5)]
# print(matched_pairs)
# matched_pairs.append((1,2))
# print(matched_pairs)

```

Out[225]:

```
[0, 1, 2, 3, 4]
```

Contents of hw_utils.py

In [96]:

```
import pickle as pkl
import cv2
import numpy as np
import random
from PIL import Image, ImageDraw
import csv
import math
import matplotlib.pyplot as plt
random.seed(1)
```

In [97]:

```
def MatchRANSAC(
        image1, image2, ratio_thres, orient_agreement, scale_agreement):
    """
    Read two images and their associated SIFT keypoints and descriptors.
    Find matches between images based on acos distance.
    Filter a subset of matches using RANSAC
    Display the final matches.
    HINT: See main_match.py on how to use this function.
    """
    im1, keypoints1, descriptors1 = ReadKeys(image1)
    im2, keypoints2, descriptors2 = ReadKeys(image2)

    keypoints1 = np.stack(keypoints1, axis=0)
    keypoints2 = np.stack(keypoints2, axis=0)
    matched_pairs = FindBestMatchesRANSAC(
        keypoints1, keypoints2,
        descriptors1, descriptors2,
        ratio_thres, orient_agreement, scale_agreement)
    matched_pairs = [
        [keypoints1[i], keypoints2[j]] for (i, j) in matched_pairs]
    assert len(matched_pairs) > 0, "No match received"
    im3 = DisplayMatches(im1, im2, matched_pairs)
    return im3


def Match(image1, image2, ratio_thres):
    """
    Read two images and their associated SIFT keypoints and descriptors.
    Find matches between images based on acos distance.
    Display the final matches.
    HINT: See main_match.py on how to use this function.
    """
    im1, keypoints1, descriptors1 = ReadKeys(image1)
    im2, keypoints2, descriptors2 = ReadKeys(image2)

    matched_pairs = FindBestMatches(
        descriptors1, descriptors2, ratio_thres)
    matched_pairs = [
        [keypoints1[i], keypoints2[j]] for (i, j) in matched_pairs]
    assert len(matched_pairs) > 0, "No match received"
    im3 = DisplayMatches(im1, im2, matched_pairs)
    return im3


def ReadKeys(image):
    """Input an image and its associated SIFT keypoints.

    The argument image is the image file name (without an extension).
    The image is read from the PGM format file image.pgm and the
    keypoints are read from the file image.key.

    ReadKeys returns the following 3 arguments:

    image: the image (in PIL 'RGB' format)

    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.

```

```
    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.
    """
    im = Image.open(image+'.pgm').convert('RGB')
    keypoints = []
    descriptors = []
    first = True
    with open(image+'.key','r') as f:
        reader = csv.reader(f, delimiter=' ', quoting=csv.QUOTE_NONNUMERIC,skipinitialspace = 
        descriptor = []
        for row in reader:
            if len(row) == 2:
                assert first, "Invalid keypoint file header."
                assert row[1] == 128, "Invalid keypoint descriptor length in header (should be
                count = row[0]
                first = False
            if len(row) == 4:
                keypoints.append(np.array(row))
            if len(row) == 20:
                descriptor += row
            if len(row) == 8:
                descriptor += row
                assert len(descriptor) == 128, "Keypoint descriptor length invalid (should be 
                #normalize the key to unit length
                descriptor = np.array(descriptor)
                descriptor = descriptor / math.sqrt(np.sum(np.power(descriptor,2)))
                descriptors.append(descriptor)
                descriptor = []
    assert len(keypoints) == count, "Incorrect total number of keypoints read."
    print("Number of keypoints read:", int(count))
    descriptors = np.stack(descriptors, axis=0)
    return [im,keypoints,descriptors]


def AppendImages(im1, im2):
    """Create a new image that appends two images side-by-side.

    The arguments, im1 and im2, are PIL images of type RGB
    """
    im1cols, im1rows = im1.size
    im2cols, im2rows = im2.size
    im3 = Image.new('RGB', (im1cols+im2cols, max(im1rows,im2rows)))
    im3.paste(im1,(0,0))
    im3.paste(im2,(im1cols,0))
    return im3

def DisplayMatches(im1, im2, matched_pairs):
    """Display matches on a new image with the two input images placed side by side.

    Arguments:
     im1           1st image (in PIL 'RGB' format)
     im2           2nd image (in PIL 'RGB' format)
     matched_pairs list of matching keypoints, im1 to im2

    Displays and returns a newly created image (in PIL 'RGB' format)
    """
    im3 = AppendImages(im1,im2)
    offset = im1.size[0]
    draw = ImageDraw.Draw(im3)
    for match in matched_pairs:
```

```
        draw.line((match[0][1], match[0][0], offset+match[1][1], match[1][0]),fill="red",width=
    im3.show()
    return im3


def ReadData(fname):
    """
    Given the fname, return the image, keypoints, and descriptors.
    Note: the fname should be a path of the image, but with no extensions.
    For example, '/my/path/ubc.png' should be '/my/path/ubc'
    """
    with open(fname + '.pkl', 'rb') as f:
        data = pkl.load(f)
    im = Image.open(fname + '.png').convert('RGB')
    keypoints = data['keypoints']
    descriptors = data['descriptors']
    return [im, keypoints, descriptors]


def FindBestMatchesXY(im_src_path, im_ref_path, ratio_thres):
    """
    This function takes two image paths, fetch the corresponding keypoints
    of the two image paths, find the best matches between keypoints
    and return the keypoint correspondances in xy coordinates.
    Inputs:
        im_src_path: the path of the source image.
        im_ref_path: the path of the image considered as the reference frame.
        ratio_thres: threshold for ratio test.
    Outputs:
        xy_src: numpy array, (matches, 2), xy coordinates of keypoints in source.
        xy_ref: numpy array, (matches, 2), xy coordinates of keypoints in ref.
    """
    assert isinstance(im_src_path, str)
    assert isinstance(im_ref_path, str)
    assert isinstance(ratio_thres, float)
    _, keypoints1, descriptors1 = ReadData(im_src_path)
    _, keypoints2, descriptors2 = ReadData(im_ref_path)
    matches = list(FindBestMatches(descriptors1, descriptors2, ratio_thres))
    matches = [(keypoints1[i1], keypoints2[i2]) for (i1, i2) in matches]

    # Extract the xy of the matches
    yx_src, yx_ref = zip(*[(match[0][:2], match[1][:2]) for match in matches])
    xy_src = np.array(yx_src)[:, [1, 0]]  # yx to xy
    xy_ref = np.array(yx_ref)[:, [1, 0]]
    return xy_src, xy_ref


def PrepareData(image_list, ratio_thres):
    """
    This function takes in a list of image paths of interests;
    Extracts the keypoints correspondance between the reference image and all other images.
    The first image on the image_list is the reference image.
    Note: there is no RANSAC performed.
    Inputs:
        image_list: a list of paths to the images (with no extensions)
        ratio_thres: the threshold for doing the ratio test of keypoint correspondance.
    Outputs:
        xy_src_list: numpy array, (num_matches, 2)
        xy_ref_list: numpy array, (num_matches, 2)
        im_list: a list of np.array, where each np.array is an image.
    """
```

```
    assert isinstance(image_list, list)
    assert len(image_list) > 1, "Need at leat two images to do stiching"
    assert isinstance(image_list[0], str)
    assert isinstance(ratio_thres, float)
    assert ratio_thres >= 0.0
    assert ratio_thres <= 1.0

    xy_src_list = []
    xy_ref_list = []
    ref_image = image_list[0]
    image_list = image_list[1:]
    for src_image in image_list:
        xy_src, xy_ref = FindBestMatchesXY(
            src_image, ref_image, ratio_thres)
        if xy_src.shape[0] >= 4:
            xy_src_list.append(xy_src)
            xy_ref_list.append(xy_ref)

    im_ref, _, _ = ReadData(ref_image)
    im_list = [np.array(im_ref)] + [
        np.array(ReadData(img)[0]) for img in image_list]
    return xy_src_list, xy_ref_list, im_list


def MergeWarppedImages(canvas_height, canvas_width, warp_list):
    """
    Wrap a list of images in the reference frame into one canvas.
    Note:
        each image is a numpy array of shape (canvas_height, canvas_width, 3)
        The first image in the warp_list is the reference image
    """
    assert isinstance(canvas_height, int)
    assert isinstance(canvas_width, int)
    assert isinstance(warp_list, list)

    canvas = np.zeros((canvas_height, canvas_width, 3), dtype=np.uint8)

    im_ref = warp_list[0]  # reference image in reference frame
    assert im_ref.dtype == np.uint8
    canvas[:im_ref.shape[0], :im_ref.shape[1]] = im_ref
    alpha = 0.5
    for wrap in warp_list[1:]:
        assert isinstance(wrap, np.ndarray)
        assert wrap.shape == canvas.shape
        assert wrap.dtype == np.uint8
        mask_wrap = Image.fromarray(wrap).convert('L')
        mask_wrap = np.array(mask_wrap) > 0

        mask_canvas = Image.fromarray(canvas).convert('L')
        mask_canvas = np.array(mask_canvas) > 0

        mask_intersect = np.logical_and(mask_canvas, mask_wrap)

        # blend in intersected area
        canvas[mask_intersect] = (
                alpha*canvas[mask_intersect] +
                (1-alpha)*wrap[mask_intersect]).astype(np.uint8)
        canvas[mask_intersect] = (
                alpha*canvas[mask_intersect] +
                (1-alpha)*wrap[mask_intersect]).astype(np.uint8)

```

```
        # copy in non-interected area
        mask_empty = np.logical_not(mask_intersect)
        canvas[mask_empty] += wrap[mask_empty]
    return canvas


def ProjectImages(
        xy_src_list, xy_ref_list, im_list,
        canvas_height, canvas_width, num_iter, tol):
    """
    This function takes in a list of images, and the points correspondance between
    the reference image and other images; computes the homography from every source
    image to the reference image using RANSAC; warp each source image to the reference
    image frame using each homography computed.
    Inputs:
        xy_src_list: a list of np array, each element is keypoint correspondance
                     between a source image to the reference image, in xy coordinates.
        xy_ref_list: a list of np array, each element is keypoint correspondance
                     between a source image to the reference image, in xy coordinates.
        im_list: all images in np.array form, the firs element is the reference image.
        canvas_height, canvas_width: the dimension of the canvas to copy the warps over.
        num_iter: number of RANSAC iterations in RANSACHomography
        tol: the Euclidean tolerance for keypoints matching projection.
    Outputs:
        A list of images in np.array form after they have been projected to
        the reference frame.
    """
    assert isinstance(xy_src_list, list)
    assert isinstance(xy_ref_list, list)
    assert isinstance(im_list, list)
    assert isinstance(canvas_height, int)
    assert isinstance(canvas_width, int)
    assert isinstance(num_iter, int)
    assert isinstance(tol, (int, float))
    assert len(xy_src_list) == len(xy_ref_list)
    assert len(xy_src_list) + 1 == len(im_list), \
        "Num of source images + 1 == num of all images"

    homo_list = []
    for xy_src, xy_ref in zip(xy_src_list, xy_ref_list):
        h = RANSACHomography(xy_src, xy_ref, num_iter, tol)
        homo_list.append(h)
    warp_list = [im_list[0]]
    im_list = im_list[1:]
    assert len(im_list) == len(homo_list)
    for im, h in zip(im_list, homo_list):
        result = cv2.warpPerspective(im, h, (canvas_width, canvas_height))
        warp_list.append(result)
    return warp_list


def VisualizePointProj(xy_src, xy_ref, xy_proj, im_src, im_ref):
    assert isinstance(xy_src, np.ndarray)
    assert isinstance(xy_ref, np.ndarray)
    assert isinstance(xy_proj, np.ndarray)
    assert isinstance(im_src, np.ndarray)
    assert isinstance(im_ref, np.ndarray)
    assert xy_src.shape == xy_ref.shape
    assert xy_src.shape == xy_proj.shape

    fig, axes = plt.subplots(
```

```
        1, 2, figsize=(30, 30), gridspec_kw={'width_ratios': [1, 2]})
    for xy_a, xy_b in zip(xy_proj, xy_ref):
        x1, y1 = xy_a
        x2, y2 = xy_b
        axes[1].plot([x1, x2],[y1, y2], 'w-', linewidth=2)

    axes[0].imshow(im_src)
    axes[0].scatter(xy_src[:, 0], xy_src[:, 1], c='#fafba4', s=100, marker='.')
    axes[0].title.set_text('Source Image')

    axes[1].imshow(im_ref)
    axes[1].scatter(xy_proj[:, 0], xy_proj[:, 1], c='#fafba4', s=100, marker='.')
    axes[1].scatter(xy_ref[:, 0], xy_ref[:, 1], c='#d63447', s=100, marker='.')
    axes[1].title.set_text('Reference Image')
    fig.show()
    input('Press any key to exit the program')

```

This sample program 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.

In [76]:

```
import matplotlib.pyplot as plt
```

Contents of main_match.py

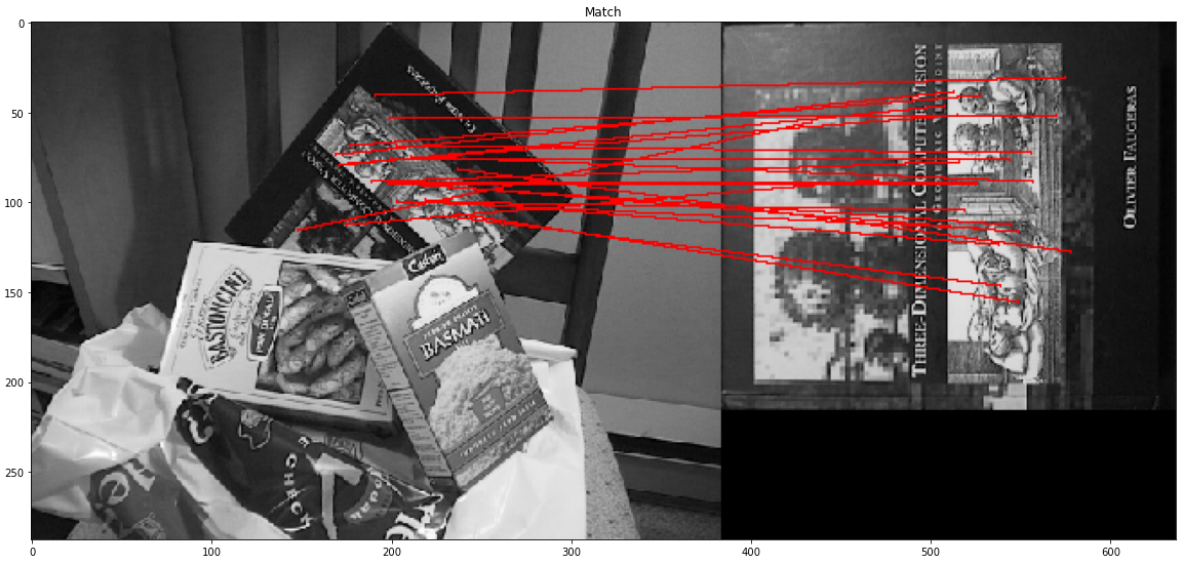
In [83]:

```
# Test run matching with no ransac 1.3
plt.figure(figsize=(20, 20))
im = Match('./data/scene', './data/book', ratio_thres=0.5)
plt.title('Match')
plt.imshow(im)

# # Test run matching with ransac 1.3
# plt.figure(figsize=(20, 20))
# im = MatchRANSAC(
#     './data/scene', './data/book',
#     ratio_thres=0.6, orient_agreement=30, scale_agreement=0.5)
# plt.title('MatchRANSAC')
# plt.imshow(im)
```

```
Number of keypoints read: 694
Number of keypoints read: 490
```

Out[83]:

```
<matplotlib.image.AxesImage at 0x1eadc102130>
```



In [ ]:

```
# library
```

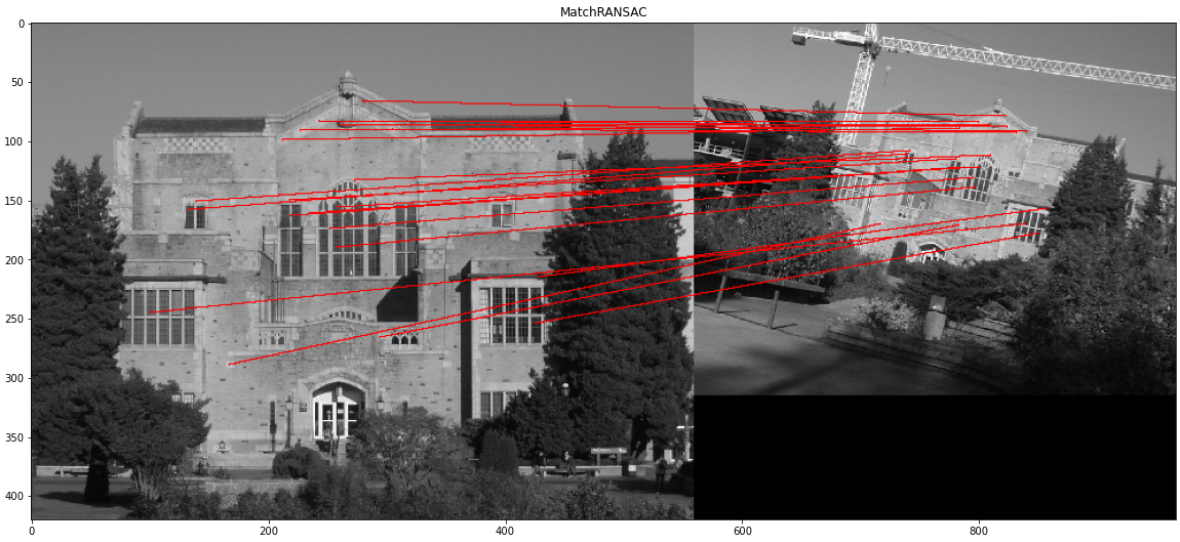
In [151]:

```
# # Test run matching with no ransac 1.4
# plt.figure(figsize=(20, 20))
# im = Match('./data/library', './data/library2', ratio_thres=0.6)
# plt.title('Match')
# plt.imshow(im)

# Test run matching with ransac 1.4
plt.figure(figsize=(20, 20))
im = MatchRANSAC(
    './data/library', './data/library2',
    ratio_thres=0.6, orient_agreement=30, scale_agreement=0.5)
plt.title('MatchRANSAC')
plt.imshow(im)
```

```
Number of keypoints read: 1766
Number of keypoints read: 849
```

Out[151]:

```
<matplotlib.image.AxesImage at 0x1eae13725e0>
```



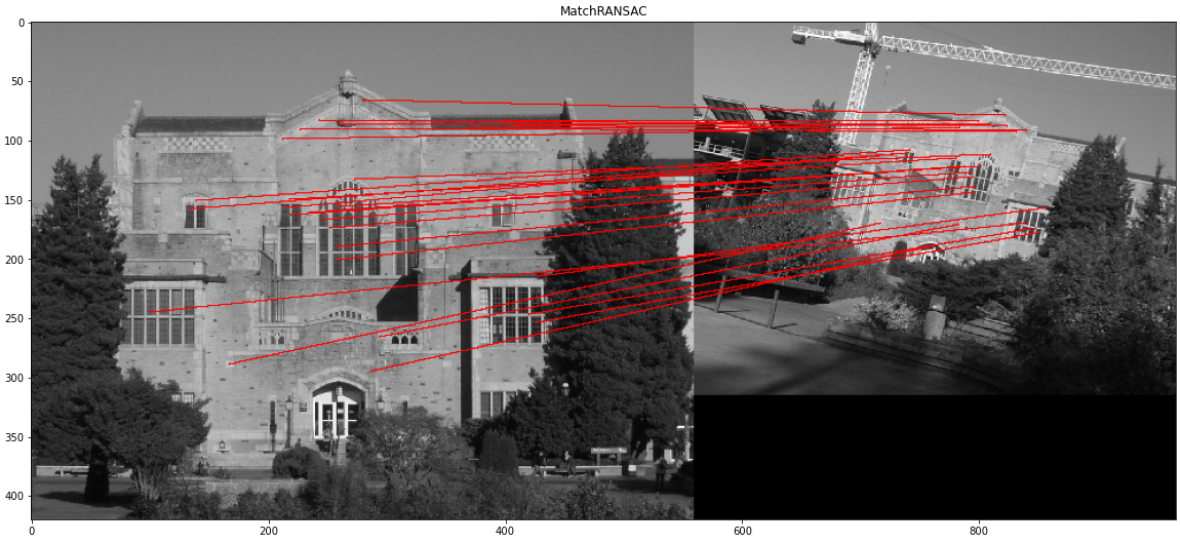
In [170]:

```
# Test run matching with ransac 1.4
plt.figure(figsize=(20, 20))
im = MatchRANSAC(
    './data/library', './data/library2',
    ratio_thres=0.62, orient_agreement=30, scale_agreement=0.6)
plt.title('MatchRANSAC')
plt.imshow(im)
```

```
Number of keypoints read: 1766
Number of keypoints read: 849
```

Out[170]:

```
<matplotlib.image.AxesImage at 0x1eae19177c0>
```



Q2: Panorama

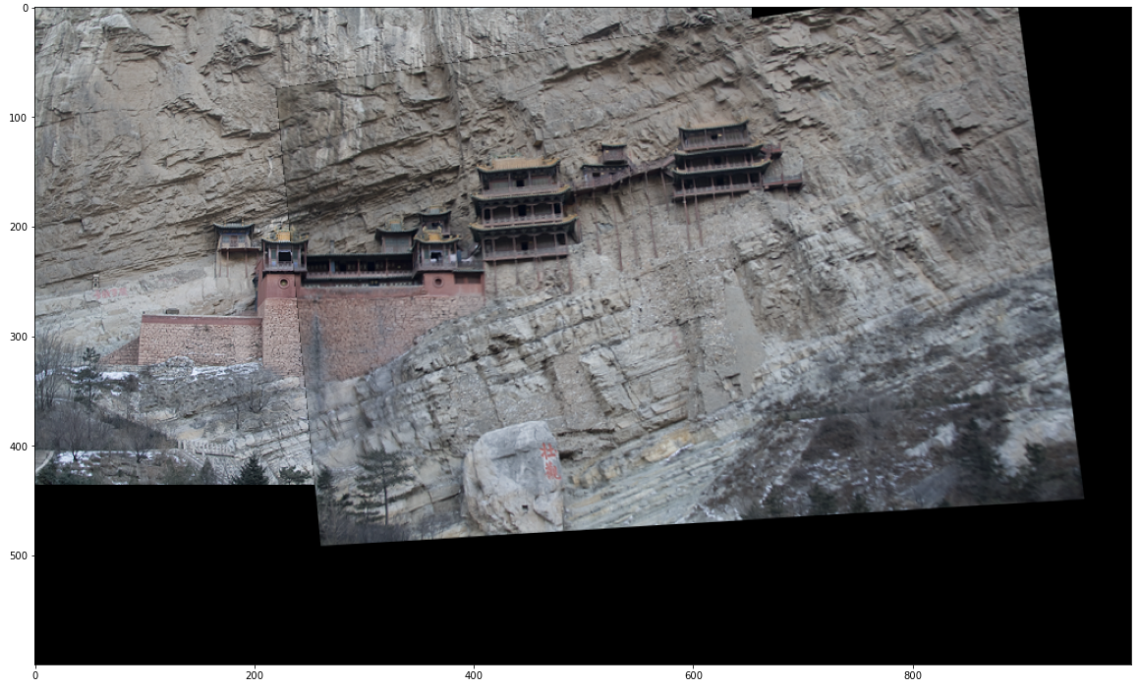
In [8]:

```
import os.path as op
```

Contents of test_pano.py

In [197]:

```
path = './data/'
image_list = ['Hanging1', 'Hanging2']
image_list = [op.join(path, im) for im in image_list]
# the dimension of the canvas (numpy array)
# to which we are copying images.
canvas_width = 1000
canvas_height = 600

# some precomputed data for sanity check
with open('./data/test.pkl', 'rb') as f:
    test_dict = pkl.load(f)
h_gt = test_dict['h']  # the homograph matrix we computed

# matches between the source and the refence image
xy_src = test_dict['xy_src']  # (match, 2)
xy_ref = test_dict['xy_ref']  # (match, 2)

# image_list should store both the reference and the source images
ref_image = image_list[0]  # first element is the reference image
source_image = image_list[1]

# compute the homography matrix to transform the source to the reference
h, _ = cv2.findHomography(xy_src, xy_ref)

# The current computed value should equal to our precomputed one
norm_diff = ((h-h_gt)**2).sum()
assert norm_diff < 1e-7, 'The computed homography matrix should equal to the given one.'

# read the two images as numpy arrays
im_src = np.array(ReadData(source_image)[0])
im_ref = np.array(ReadData(ref_image)[0])

# project source image to the reference image using the homography matrix
# the size of canvas is specified to store all images after projections.
im_src_warp = cv2.warpPerspective(im_src, h, (canvas_width, canvas_height))

# warp_list should contain all images, where the first
# element is the reference image
warp_list = [im_ref, im_src_warp]
result = MergeWarppedImages(canvas_height, canvas_width, warp_list)

# plot the result of the warping
plt.figure(figsize=(20, 20))
plt.imshow(result)
plt.show()
```



In [183]:

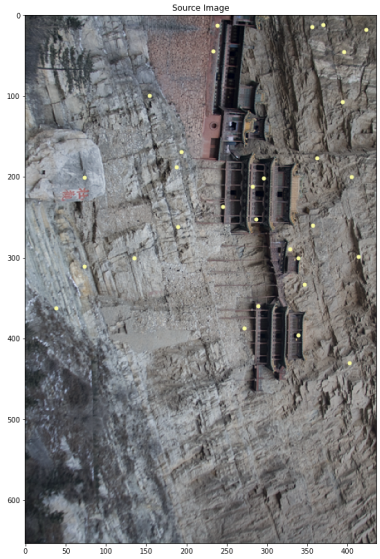
```
1 import sys
```

Contents of main_proj.py

In [198]:

```
with open('./data/test.pkl', 'rb') as f:
    test_dict = pkl.load(f)

# visualize 30 random matches
num_pts = 30
idx = np.random.permutation(test_dict['xy_src'].shape[0])[:num_pts]
xy_src = test_dict['xy_src'][idx]
xy_ref = test_dict['xy_ref'][idx]
h = test_dict['h']

# project the src keypoints to the reference frame using homography
xy_proj = KeypointProjection(xy_src, h)

# visualize the results
im_ref = np.array(Image.open('./data/Hanging1.png'))
im_src = np.array(Image.open('./data/Hanging2.png'))
VisualizePointProj(xy_src, xy_ref, xy_proj, im_src, im_ref)
```

```
<ipython-input-97-527ad4660e3d>:318: UserWarning: Matplotlib is currently using modu
le://ipykernel.pylab.backend_inline, which is a non-GUI backend, so cannot show the
figure.
  fig.show()

(30, 3)
(30, 3)
Press any key to exit the programa
```



Contents of main_pano.py

In [230]:

```
def create_pano(
        image_list, ratio_thres,
        canvas_height, canvas_width,
        num_iter, tol, figsize=(20, 20)):
    """
    This function creates a panorama using a list of images.
    Inputs:
        image_list: a list of str, the path to each image (without file extensions).
        ratio_thres: the ratio test threshold in `FindBestMatches`
        canvas_height, canvas_width: The dimension of the canvas
        num_iter: num of iterations of performing RANSAC to find the homography matrix.
        tol: tolerance for keypoint projection
    """
    # Get the matches from `FindBestMatches`
    # xy_src_list: np.array, (matches, 2) in xy format
    # xy_ref_list: np.array, (matches, 2) in xy format
    # im_list: a list of images in np.array
    xy_src_list, xy_ref_list, im_list = PrepareData(
        image_list, ratio_thres)

    # Use the matches to estimate a homography matrix to the ref image frame
    # for each source image. Then project each source image to the reference
    # frame using the homography matrix.
    wrap_list = ProjectImages(
        xy_src_list, xy_ref_list, im_list,
        canvas_height, canvas_width, num_iter, tol)

    # Merge the projected images above
    # Note: the first element is the reference image in warp_list
    result = MergeWarppedImages(
        canvas_height, canvas_width, wrap_list)

    # show the final panorama
    plt.figure(figsize=figsize)
    plt.imshow(result)
    plt.show()
```

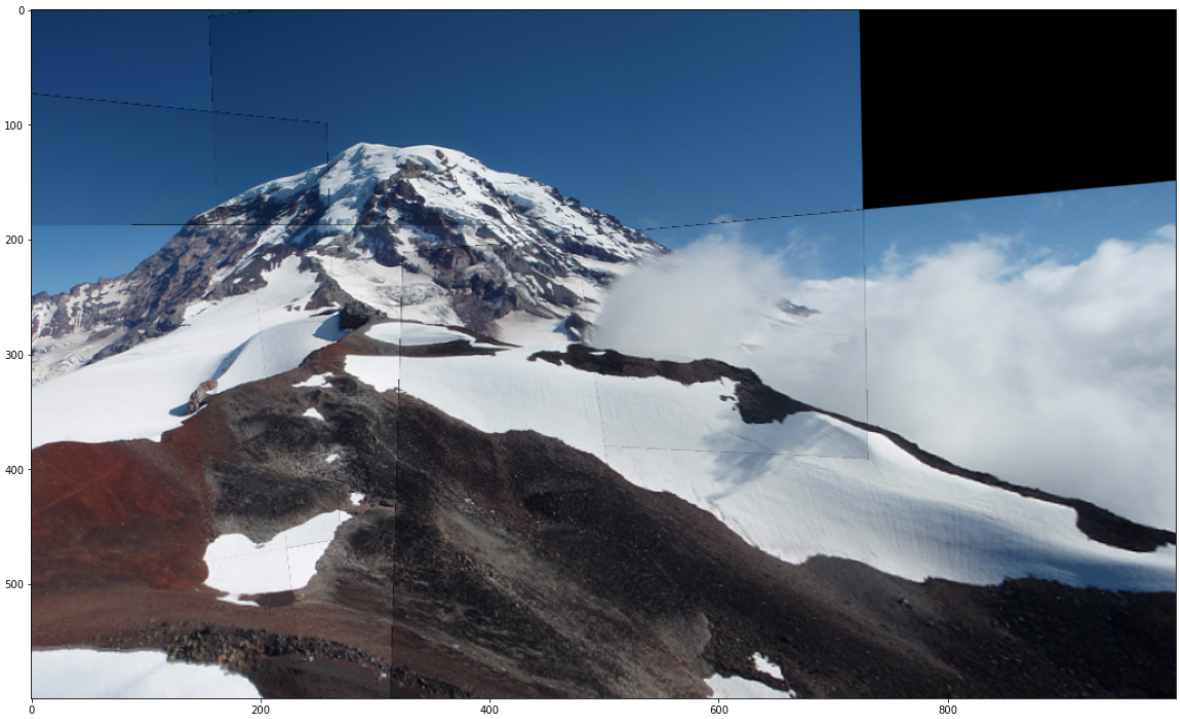
In [262]:

```
path = './data/'

canvas_height = 600
canvas_width = 1000
image_list = ['Rainier1', 'Rainier2', 'Rainier3','Rainier4','Rainier5','Rainier6']

num_iter = 50
tol = 10
ratio_thres = 0.1
image_list = [op.join(path, im) for im in image_list]
create_pano(image_list, ratio_thres, canvas_height, canvas_width,
            num_iter, tol, figsize=(20, 20))
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-262-702cd67d1893> in <module>
      9 ratio_thres = 0.1
     10 image_list = [op.join(path, im) for im in image_list]
---> 11 create_pano(image_list, ratio_thres, canvas_height, canvas_width,
     12             num_iter, tol, figsize=(20, 20))

<ipython-input-230-61dbf5d951d9> in create_pano(image_list, ratio_thres, ca
nvas_height, canvas_width, num_iter, tol, figsize)
     22     # for each source image. Then project each source image to the
 reference
     23     # frame using the homography matrix.
---> 24     wrap_list = ProjectImages(
     25         xy_src_list, xy_ref_list, im_list,
     26         canvas_height, canvas_width, num_iter, tol)

<ipython-input-97-527ad4660e3d> in ProjectImages(xy_src_list, xy_ref_list, i
m_list, canvas_height, canvas_width, num_iter, tol)
    276     assert isinstance(tol, (int, float))
    277     assert len(xy_src_list) == len(xy_ref_list)
--> 278     assert len(xy_src_list) + 1 == len(im_list), \
    279         "Num of source images + 1 == num of all images"
    280

AssertionError: Num of source images + 1 == num of all images
```

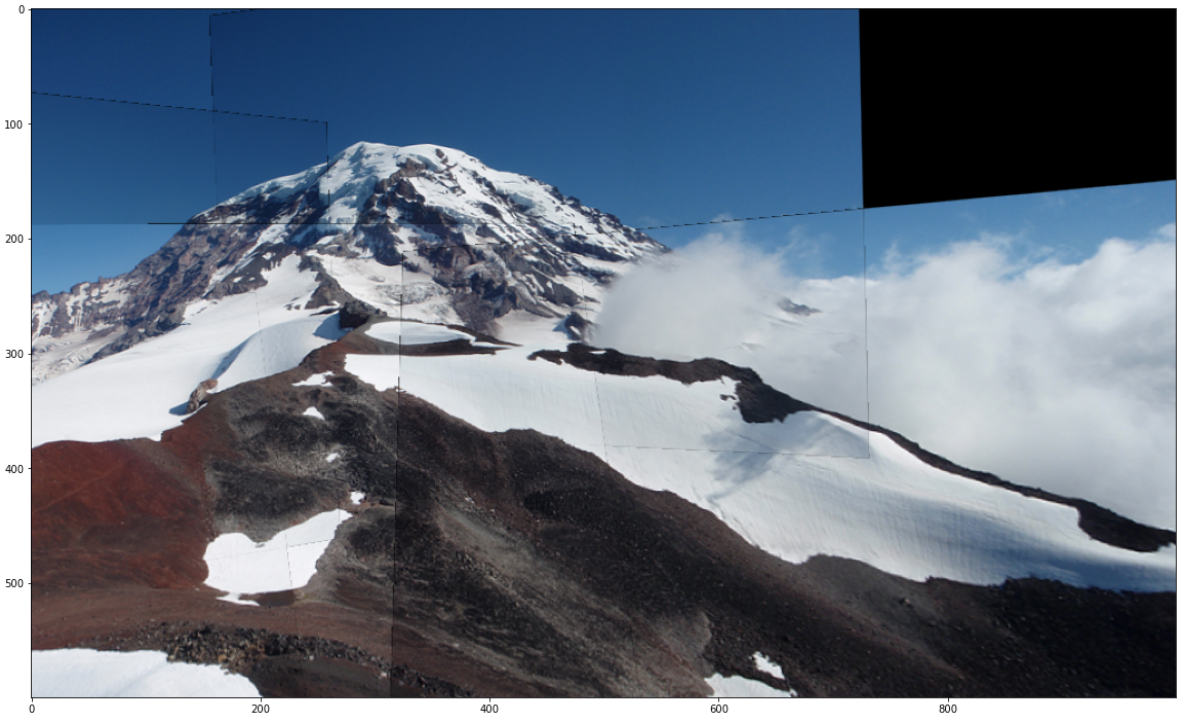
In [257]:

```
path = './data/'

canvas_height = 600
canvas_width = 1000
image_list = ['Rainier1', 'Rainier2', 'Rainier3','Rainier4','Rainier5','Rainier6']

num_iter = 50
tol = 5
ratio_thres = 0.9
image_list = [op.join(path, im) for im in image_list]
create_pano(image_list, ratio_thres, canvas_height, canvas_width,
            num_iter, tol, figsize=(20, 20))
```



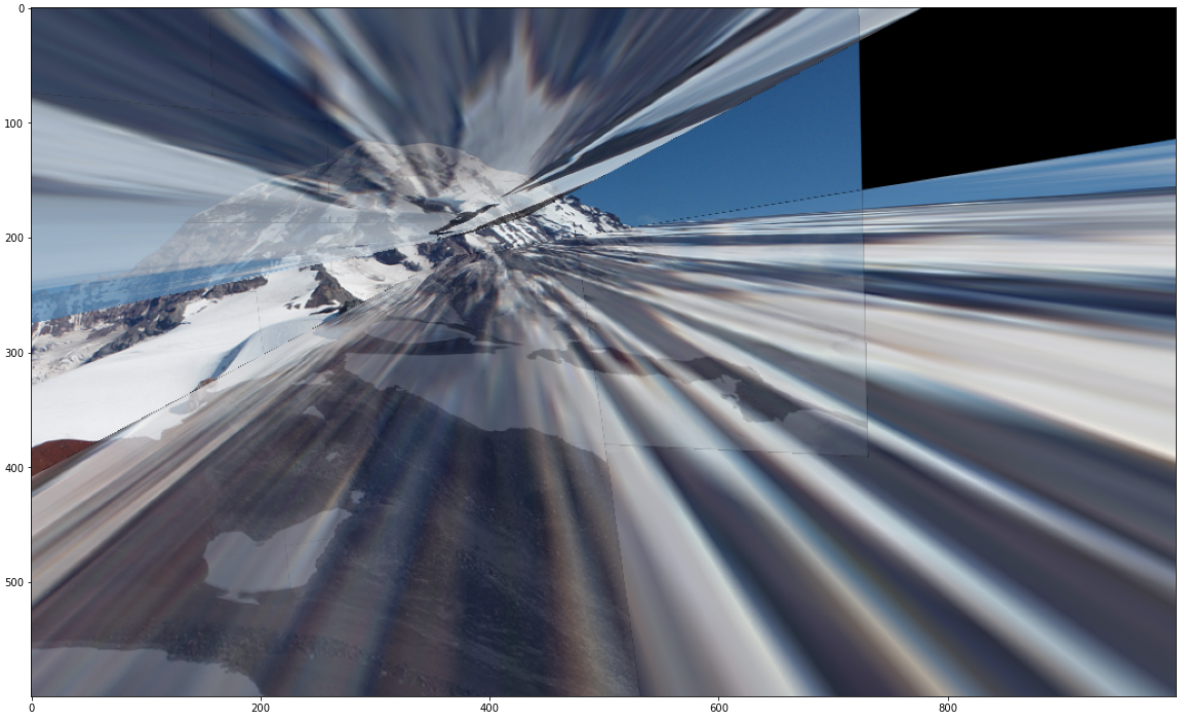
In [258]:

```
path = './data/'

canvas_height = 600
canvas_width = 1000
image_list = ['Rainier1', 'Rainier2', 'Rainier3','Rainier4','Rainier5','Rainier6']

num_iter = 50
tol = 10
ratio_thres = 0.5
image_list = [op.join(path, im) for im in image_list]
create_pano(image_list, ratio_thres, canvas_height, canvas_width,
            num_iter, tol, figsize=(20, 20))
```



In [259]:

```
path = './data/'

canvas_height = 600
canvas_width = 1000
image_list = ['Rainier1', 'Rainier2', 'Rainier3','Rainier4','Rainier5','Rainier6']

num_iter = 50
tol = 10
ratio_thres = 1.0
image_list = [op.join(path, im) for im in image_list]
create_pano(image_list, ratio_thres, canvas_height, canvas_width,
            num_iter, tol, figsize=(20, 20))
```



In [268]:

```
path = './data/'

canvas_height = 1125
canvas_width = 1500
image_list = ['fountain4', 'fountain0']

num_iter = 100
tol = 10
ratio_thres = 0.8
image_list = [op.join(path, im) for im in image_list]
create_pano(image_list, ratio_thres, canvas_height, canvas_width,
            num_iter, tol, figsize=(20, 20))
```

In [276]:

```
path = './data/'

canvas_height = 1125
canvas_width = 1500
image_list = ['garden0', 'garden3','garden4']

num_iter = 200
tol = 15
ratio_thres = 0.8
image_list = [op.join(path, im) for im in image_list]
create_pano(image_list, ratio_thres, canvas_height, canvas_width,
            num_iter, tol, figsize=(20, 20))
```

In [273]:

```
path = './data/'

canvas_height = 1125
canvas_width = 1500
image_list = ['irving_out3', 'irving_out6','irving_out5']

num_iter = 150
tol = 10
ratio_thres = 0.9
image_list = [op.join(path, im) for im in image_list]
create_pano(image_list, ratio_thres, canvas_height, canvas_width,
            num_iter, tol, figsize=(20, 20))
```

In [ ]:

```

```