ECE 661 Fall 2020 - Homework 5

Brian Helfrecht

bhelfre@purdue.edu

1 Theory

One application of interest point detection and matching in computer vision is panorama stitching, in which a series of images of the same scene are merged into a single image. Correspondences are found between images with overlapping features using an interest point detection algorithm. The RANSAC algorithm is then applied to the correspondences to compute the optimal homography for image alignment. If desired, a nonlinear least squares algorithm such as Levenberg-Marquardt can be applied to refine the homographies. Finally, the images are stitched together using these homographies.

More detailed information about panorama generation is provided in the subsections below, followed by the answer to the theory question posed. Implementation notes are given in section 3. Lastly, results of applying the theory to a set of images, along with a discussion of those results, can be found in section 4.

1.1 Linear least squares for homography estimation

The minimum number of points required to estimate a homography between two images is 4. However, using only the minimum number of points often yields poor results. To remedy this, a homography can be estimated using more than four points using linear least-squares minimization. Two methods are available to do this, with one using homogeneous equations and the other inhomogeneous equations. The latter method is arguably more straightforward, so it was used for this assignment. Its functionality is described below.

Homography estimation using more than 4 corresponding points is quite similar to that using only four points when inhomogeneous equations are used. Homography estimation using 4 corresponding points was discussed at length in homeworks 2 and 3, so some steps will be omitted in this discussion. Recall that for each pair of corresponding physical points in the domain (x, y) and range (x', y'), we can write the equations:

$$x'_{physical} = h_{11}x + h_{12}y + h_{13} - h_{31}xx'_{physical} - h_{32}yx'_{physical}$$

$$y'_{physical} = h_{21}x + h_{22}y + h_{23} - h_{31}xy'_{physical} - h_{32}yy'_{physical}$$

where h_{ij} are the eight unknowns of the homography H to be estimated. The equations can then be written in matrix form:

$$\begin{pmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{pmatrix} = \begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -y_1x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & -y_1y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & -y_2x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y'_2 & -y_2y'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x'_3 & -y_3x'_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y'_3 & -y_3y'_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x'_4 & -y_4x'_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y'_4 & -y_4y'_4 \end{pmatrix} \begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{pmatrix}$$

Note that every two rows contain exactly the coefficients in the two equations above. By induction, we can continue to add rows to these matrices using additional correspondences. However, we saw in the previous assignments that the elements of h are solved via:

$$A^{-1}x' = h$$

which requires A to be a square matrix, so its inverse can be calculated. If we have more than 8 equations, A will not be square. In this case, we can use the *pseudo-inverse* of A, denoted A^+, which will allow us to minimize the error between x' and Ah. Geometrically, this minimization is found by projecting x' onto the subspace spanned by Ah. That is, the least-squares minimization must satisfy:

$$(A^T A)h = A^T b$$

Solving for h yields:

$$h = (A^T A)^{-1} A^T b = A^+ b$$

Here we see the mathematical definition of the pseudo-inverse:

$$A^+ = (A^T A)^{-1} A^T$$

So, ultimately the matrix equations created with more than four corresponding points can be solved using linear least-squares minimization with the following formula:

$$A^+ x' = h$$

All that is left is substituting the solved values of h back into the 3×3 homography matrix H. This will yield the best estimate of the homography using more than 4 correspondences.

1.2 The RANSAC algorithm

As described in the previous section, robust homography estimation can be achieved using more than 4 pairs of points. However, manually picking these correspondences becomes increasingly difficult as more are used. As such, it is desirable to have an automatic method of choosing correspondences. This can be accomplished using the Random Sampling and Consensus (RANSAC) algorithm, which automatically determines valid correspondences between interest points in images of the same scene.

Before RANSAC can be employed, interest points in different images of the same scene are needed. These interest points can be extracted using an interest point extractor, such as a Harris corner detector or the SIFT or SURF algorithms. Additionally, corresponding interest points between the scene views must be marked using a method such as the Sum of Squared Differences or Normalized Cross Correlation. Unsurprisingly, some of the correspondences between interest points are likely to be false, so a method is needed to reject these false pairings.

Identifying valid and invalid correspondences is the focus of the RANSAC algorithm. Valid correspondences are called *inliers*, while invalid correspondences are called *outliers*. Several parameters are needed to apply the algorithm:

- σ: The variance parameter of the estimated Gaussian noise factor between interest points in the target images. Naturally, interest points will not land on exactly the same real-world point in both images, but may instead be shifted by some amount of noise. Often, σ lies between 0.5 and 2.
- δ: The decision threshold (or radius) around each interest point that constitutes a valid correspondence (an inlier). Transformed points that fall within this distance of its true corresponding point are considered inliers, while those that fall outside of the radius are outliers, or invalid correspondences. Typically, $\delta = 3\sigma$.
- n: The number of correspondences used for calculating the initial homography, typically between 4 and 10.
- ϵ: An initial guess as to the percentage of correspondences that are invalid.
- N: The number of trials to conduct. This value should be set such that at least one trial produces a set that contains no outliers. The value of N is largely based off of p, and is calculated by:

$$N = \frac{ln(1-p)}{ln(1-(1-\epsilon)^n)}$$

- p: The probability that at least one trial will contain no outliers. We often set $p = 0.99$.
- M: The minimum size of a valid inlier set, calculated: $M = (1 - \epsilon)n$

With these parameters, The RANSAC algorithm proceeds as follows:

1. Begin with a set of corresponding interest points between two different images of the same scene. This set is likely to contain both valid and invalid correspondences (inliers and outliers).
2. Randomly select n correspondences from the set without replacement.
3. Calculate an initial homography H_{ij} from image i to image j using the n correspondences.
4. Apply the homography to all interest points in image i to obtain the transformed interest points.
5. Calculate the distance d between each transformed interest point and its true corresponding interest point, as initially detected by the interest point detector.
6. Add any interest points where $d \leq \delta$ to a set of inliers. Points where $d > \delta$ constitute the outlier set. Keep note of these sets for each trial.
7. Repeat steps 2-6 N times.
8. Recalculate the homography between images i and j using all correspondences in the largest inlier set. This largest set should have at least M correspondences, and will be the optimal homography constructed using linear least-squares minimization with more than 4 interest points.

The final homography determined using RANSAC can then be applied to transform each image to the domain of another image. This is very helpful when aligning several images, as required for panorama stitching.

1.3 Non-linear least squares using the Levenberg-Marquardt method

The goal of any nonlinear least-squares minimization algorithm is to minimize the error present in a cost function, $C(p)$:

$$C(p) = ||X - f(p)||^2$$

where X is a vector containing theoretical results and $f(p)$ is a vector containing measured results corresponding with each element in X, computed through a nonlinear function with a dependence on some input p. The goal is to choose p such that the error $C(p)$ is minimized. In order for the function to work, however, an initial guess of p must be provided.

In our scenario, p represents the eight unknown coefficients of the homography matrix H. We wish to find the best coefficients to minimize any errors in the homography calculation. To begin p is initially set using the homography matrix coefficients output from the RANSAC algorithm. X is a $2N \times 1$ vector containing alternating (x', y') coordinates in a range image that correspond to (x, y) coordinates in the domain. That is:

$$X = \begin{pmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ \vdots \\ x'_N \\ y'_N \end{pmatrix}$$

These points are output directly from an interest point detection algorithm, and N is the number of interest points detected. The $f(p)$ term is a $2N \times 1$ vector of the same form as X, but with each entry being $\tilde{x}$ or $\tilde{y}$—the resulting coordinates after applying a homography to the domain image (x, y) points. Several methods can then be used to minimize the error between these terms.

The LM method is a robust and efficient algorithm for computing nonlinear least-squares minimization. It combines the numerical stability of the Gradient Descent (GD) method and the computational efficiency of the Gauss-Newton (GS) method for computing nonlinear least-squares. This method requires some knowledge about GD and GN that are not presented here, but the results are used. It can be shown that the minimal solution found using the GN method is given by:

$$\delta_p = (J_f^T J_f)^{-1} J_f^T \epsilon(p)$$

which can be re-written as:

$$(J_f^T J_f)\delta_p = J_f^T \epsilon(p)$$

where J_f is the jacobian of the function f with dependence on p. Additionally, if the product $J_f^T J_f$ is a diagonal matrix, then both GD and GN produce the same final solution. We can then modify the GN result to incorporate a weighting of the GD result:

$$(J_f^T J_f + \mu I)\delta_p = J_f^T \epsilon(p)$$

The term μ is known as the *damping coefficient*, and determines which solution (GD or GN) is targeted most closely at each iteration step. If μ is large, the solution will tend towards that produced by the GD method. If μ is small, it will tend toward the result produced by GN. Obviously, being able to set μ properly is critical to the performance of LM.

We also must note that the LM, GD, and GN nonlinear least-squares algorithms are iterative, meaning they take several steps to reach a solution, optimizing themselves at each step. When using LM, the output δ_p at each step k is:

$$\delta_p = (J_f^T J_f + \mu_k I)^{-1} J_f^T \epsilon(p_k)$$

The new "minimized" value of p is then $p_{k+1} = p_k + \delta_p$. This value is "tested" by inputting it to the cost function and comparing it with the previous value of that function, i.e. $C(p) - C(p_{k+1})$. This result is important: it characterizes how "good" the step was (i.e. if the step moved closer to a minimal solution without missing it). More specifically, the cost function difference is used to compute a ratio that defines the next value of μ:

$$\rho_{k+1} = \frac{C(p) - C(p_{k+1})}{\delta_p^T J_f^T \epsilon(p_k) + \delta^T \mu_k I \delta_p}$$

The value of μ for the next iteration is then:

$$\mu_{k+1} = \mu_k \cdot max\{\frac{1}{3}, 1 - (2\rho_{k+1} - 1)^3\}$$

This value then drives the minimization formula closer to the GD or GN solution. Finally, the initial value of μ needs to be calculated. We do this by setting:

$$\mu_0 = \tau \cdot max\{diag(J_f^T J_f)\}$$

for some $0 < \tau \leq 1$. The process described above is repeated until the LM algorithm produces an error that is sufficiently small. The resulting p is the nonlinear least-squares solution.

2 Theory questions

Q: Conceptually speaking, how do we differentiate between the inliers and the outliers when using RANSAC for solving the homography estimation problem using the interest points extracted from two different photos of the same scene?

A: RANSAC differentiates inliers and outliers in the following way: First, the homography between a pair of images (we will call them images 1 and 2) is estimated using a few selected corresponding interest points in each image, say p_1 in image 1 and p_2 in image 2. This homography is then applied to each interest point in image 1, to get a point p_t in the reference frame of image 2. The distance d between p_t and p_2 is then calculated (this can be done because p_t and p_2 are in the same reference frame). If $d \leq \delta$ (with δ set beforehand with some knowledge of the data), we say that p_1 and p_2 constitute a valid correspondence, or an inlier. If $d > \delta$, p_1 and p_2 constitute an outlier correspondence, or an invalid one.

Q: As you will see in lecture 12, Gradient-Descent (GD) is a reliable method for minimizing a cost function, but it can be excruciatingly slow. At the other extreme, we have the much faster Gauss-Newton (GN) method which can be numerically unstable. Explain in your own words how the Levenberg-Marquardt (LM) algorithm combines the best of GD and GN to give us a method that is reasonably fast and numerically stable at the same time.

A: When it comes to nonlinear least-squares minimization, the Gradient Descent (GD) and Gauss-Newton (GN) methods are both good options. However, they each have a downfall: GD can perform extremely slowly because the step size as it approaches the solution becomes smaller and smaller (requiring more and more computations). GN, on the other hand, is efficient, but can fail if the Jacobian of the transformed points is not full rank. Additionally, the initial guess must be somewhat close to the true minimum solution for GN to work. Fortunately, the Levenberg-Marquardt (LM) algorithm combines the best of GD and GN to attempt to overcome these issues.

At a high level, LM behaves like GD if/when the initial/current guess is far from the true minimum, but switches to behave like GN when the guess becomes sufficiently close to that minimum. This is possible because when the product of the jacobian matrix of the transformed points and its transpose are a diagonal matrix, the GD and GN solutions are the same. This gives us an outlet to combine the two, which is accomplished by adding a damping factor to the jacobian product that effectively governs how strong of an influence GD plays in the final solution. When the damping factor is large, the solution is heavily influenced by GD. When the damping factor is small (or 0), the solution is heavily influenced (or totally determined) by GN. This allows the true solution to be obtained with stability and speed.

Several factors determine the value of the damping factor, but the main idea is that the damping factor has an inverse relationship with the distance the current solution point is from the true solution point. That is, the nearer the solution point is to the true solution, the smaller the damping factor will be, and therefore the more heavily GN will weigh on the calculation (this gives the algorithm its convergence speed). The cost function is also used to determine whether a step towards the solution is ideal or not. If the step is not ideal (i.e. it may have jumped "too far" towards the predicted solution), GD will weigh more heavily than GN to prevent any numerical instability. Overall, the LM method is both stable and computationally efficient.

3 Implementation notes

- Optimal parameters for RANSAC were found to be: $\sigma = 2, \delta = 3\sigma = 6, p = 0.99, \epsilon = 0.25, n = 6$.
- The SIFT algorithm for finding correspondences between images was used for interest point detection. This method was used over the Harris corner detector developed in homework 4 and the SURF algorithm for its increased robustness and accuracy.
- Correspondences were made from the strongest 250 SIFT interest points in each image pair by minimizing the SSD between descriptors of all possible pairs of interest points. In order for a match to be valid, the two points must correspond with each other; that is, the points must be each other's best match from the set of all interest points in each image. This ensures a 1:1 correspondence between all interest points.
- The scipy.optimize.least_squares() function was used to implement the non-linear least squares LM method. No custom implementation was created for this assignment.
- The middle image was used as the anchor image for panorama stitching. All homographies were calculated with respect to this image. That is, the homography H_k applied to image $k = 1, 2, 3, 4, 5$ was:

$$H_1 = H_{13} = H_{23} \times H_{12}$$
$$H_2 = H_{23}$$
$$H_3 = I_3$$
$$H_4 = H_{34}^{-1}$$
$$H_5 = H_{53} = H_{34}^{-1} \times H_{45}^{-1}$$

  Where I_3 is the 3×3 identity matrix. This was done to minimize distortion that results from the combination of several homographies, as the small errors in each become amplified.
- A custom algorithm was used for image stitching for efficiency. This method automatically computes the panorama size as images are added and ensures a tight fit (i.e. no entire row/column of the final panorama will be entirely empty). It also avoids the need to compute translation components in some of the homographies. The algorithm works as follows:

  1. Use the anchor image as the initial panorama. The height of the anchor image constrains the height of the panorama.
  2. Begin appending images to the right of the anchor image. For each image i, calculate the new vertex locations by applying the homography H_i. Use the intersection of the line defined by the two right-most transformed vertices and the top and bottom of the panorama to determine how many empty pixels to add to the panorama. Also note the minimum x-coordinate of the left-most vertices to define a region to update with pixels from the image to append.
  3. Apply the homography H_i to the pixels in the region to update as determined above. Note that each pixel to be updated has to be shifted by the minimum x-coordinate vertex to correctly apply the homography.
  4. Append images to the left of the anchor image using the same process as above, but expanding the canvas using the minimum x-coordinate vertices, rather than the maximum. Additionally, we must add back the magnitude of the minimum x-coordinate vertex value (which will be negative) to each point to be updated in the final panorama to ensure pixels are updated properly.
- Two approaches were tested to update pixels within the region to update. One method was to simply update all pixels in the region with pixels from the new image. This was the selected method. The other method was to update only black pixels (i.e. pixels that had not already been taken from one of the input images). The latter method was not used because it caused some minor duplication of small pixel regions near stitch lines.

4 Results

After completing the tasks, I have drawn the following conclusions:

- The resulting panorama looks quite good. To someone unfamiliar with each individual image, it may be difficult to see any errors. However, some errors are present in the output image, and stem from several causes:
  - Errors in homography calculations (not having a minimal correspondence set, rounding errors).
  - Movement in the scene such as the clouds and water that were not stationary between images.
  - Differing image characteristics such as lighting or white balance between images. This especially causes very visible stitch lines.
- Almost all of the correspondences between images detected with SIFT and my matching algorithm seem perfect to the human eye. However, several of these became outliers in some of the images likely due to slight movements of the camera between shots. Each image was taken with a Nikon D850 DSLR camera at 1/30 sec, f/11, ISO 64 with a 24mm focal length mounted on a tripod. Due to the nature of the scene and the physical length of the lens, some parallax effects were introduced in the images as the camera was rotated, which likely resulted in several correspondences being marked as outliers. To minimize this, I set σ to 2, which is the highest value in its typical range. It is possible that pushing σ even higher to include even more "very close" correspondences would produce even better results.
- Image pars 1-2, 2-3, and 4-5 had 246, 250, and 208 correspondences, respectively. However, image pair 3-4 had only 189–just one higher than the required 188 for satisfactory homography estimation. It is likely that parallax effects from the large foreground element played a major role in this lower value.
- The LM non-linear least squares method does seem to improve each homography slightly. However, the difference these make in the final panorama are hardly noticeable. Likely, the LM algorithm would work much better in images with fewer inliers.

Images depicting interest points and correspondences, inliers and outliers, and the final stitched panorama can be found on the following pages. In images showing inliers and outliers, inliers are marked in green, and outliers in red.

4.1 Input images

Figure 1: Input image 1.

Figure 2: Input image 2.

Figure 3: Input image 3.

Figure 4: Input image 4.

Figure 5: Input image 5.

4.2 Image correspondences

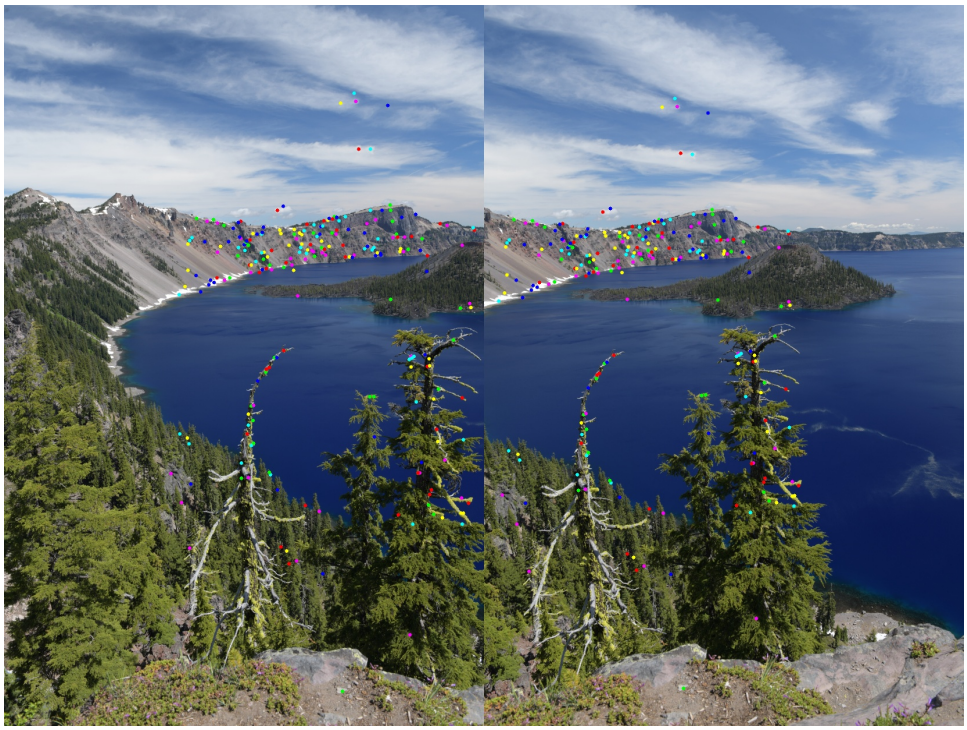

Figure 6: SIFT correspondences between images 1 and 2 (no lines for clarity).

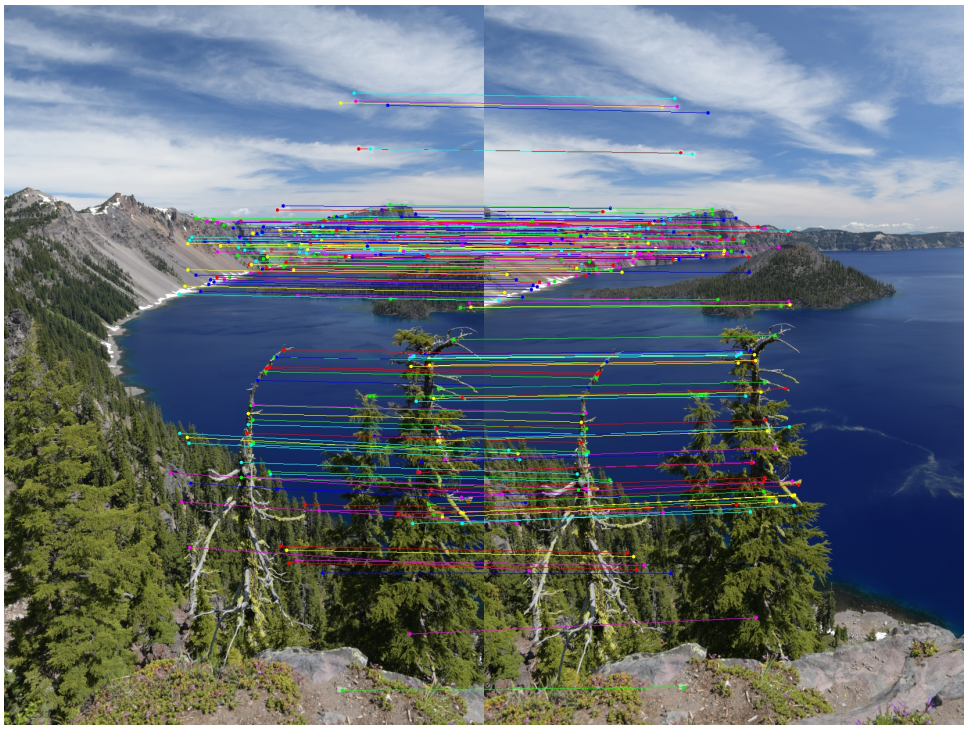

Figure 7: SIFT correspondences between images 1 and 2 (with lines).

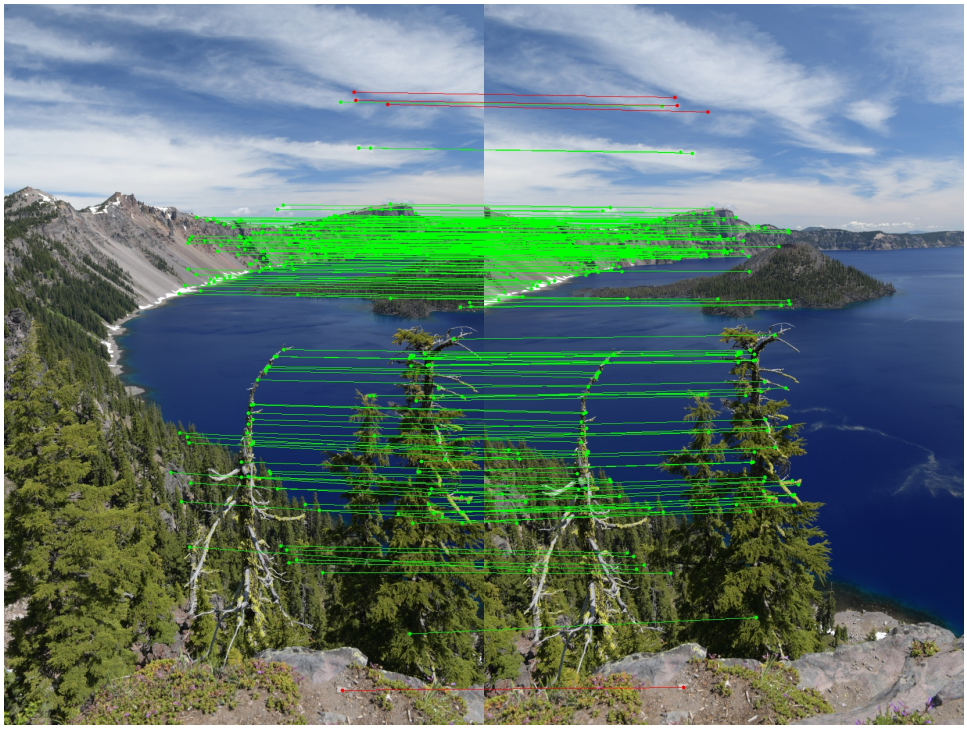

Figure 8: Inliers and outliers between images 1 and 2.

Figure 9: SIFT correspondences between images 2 and 3 (no lines for clarity).

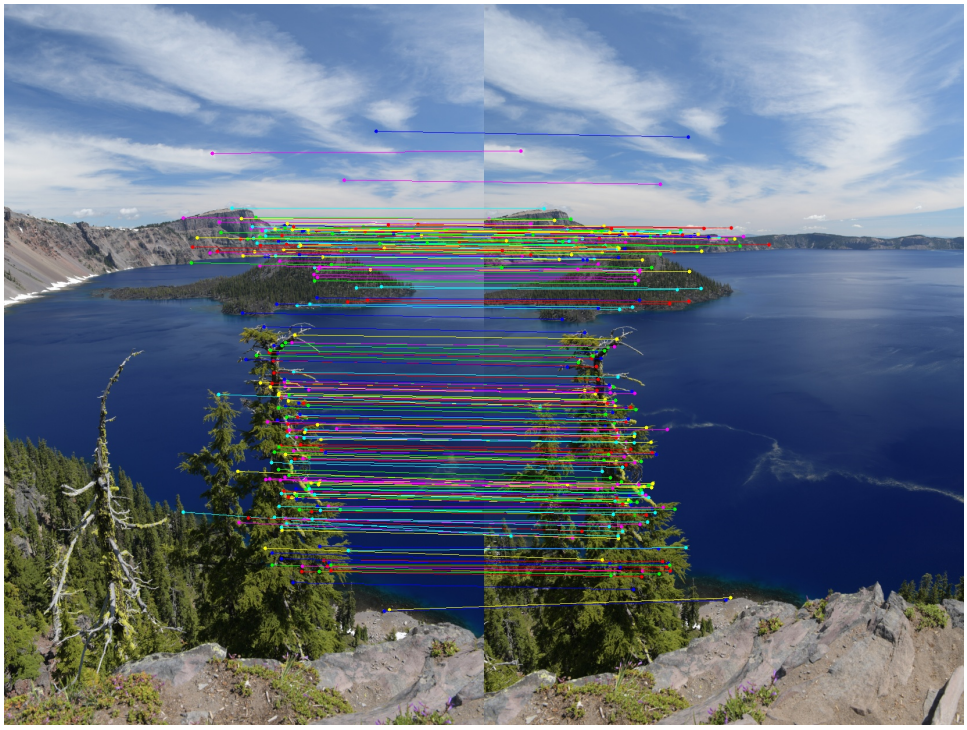

Figure 10: SIFT correspondences between images 2 and 3 (with lines).

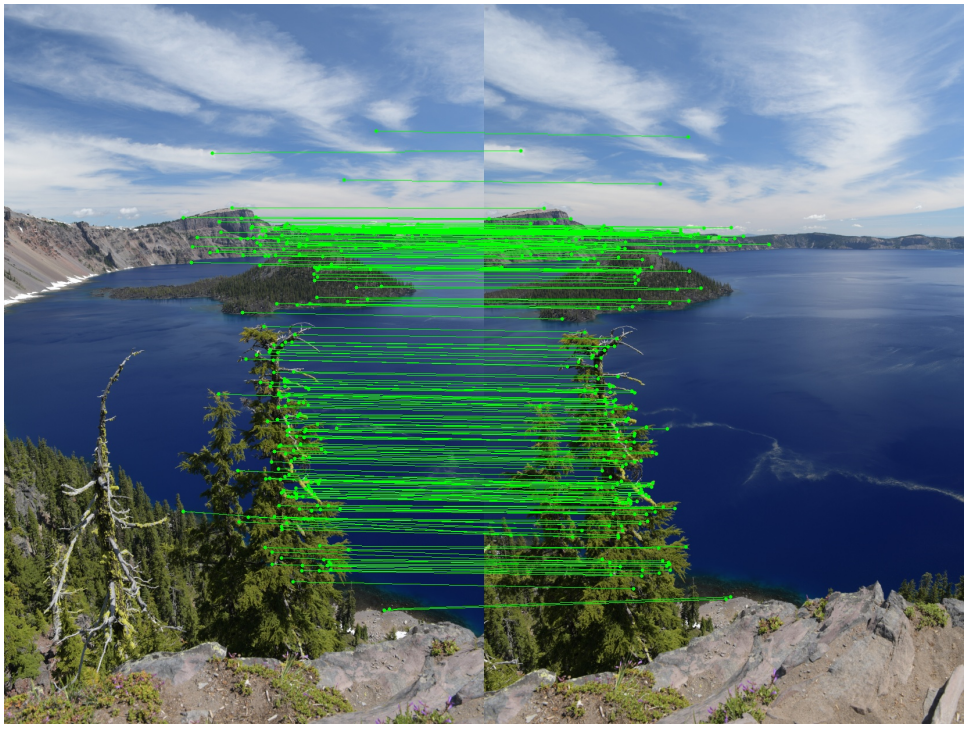

Figure 11: Inliers and outliers between images 2 and 3.

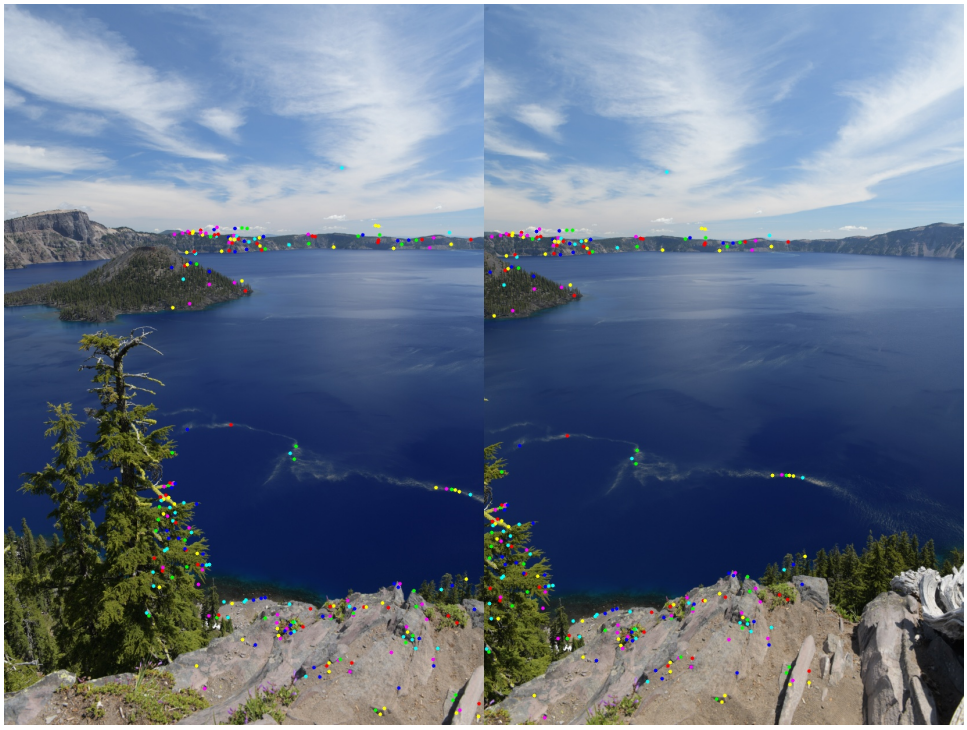

Figure 12: SIFT correspondences between images 3 and 4 (no lines for clarity).

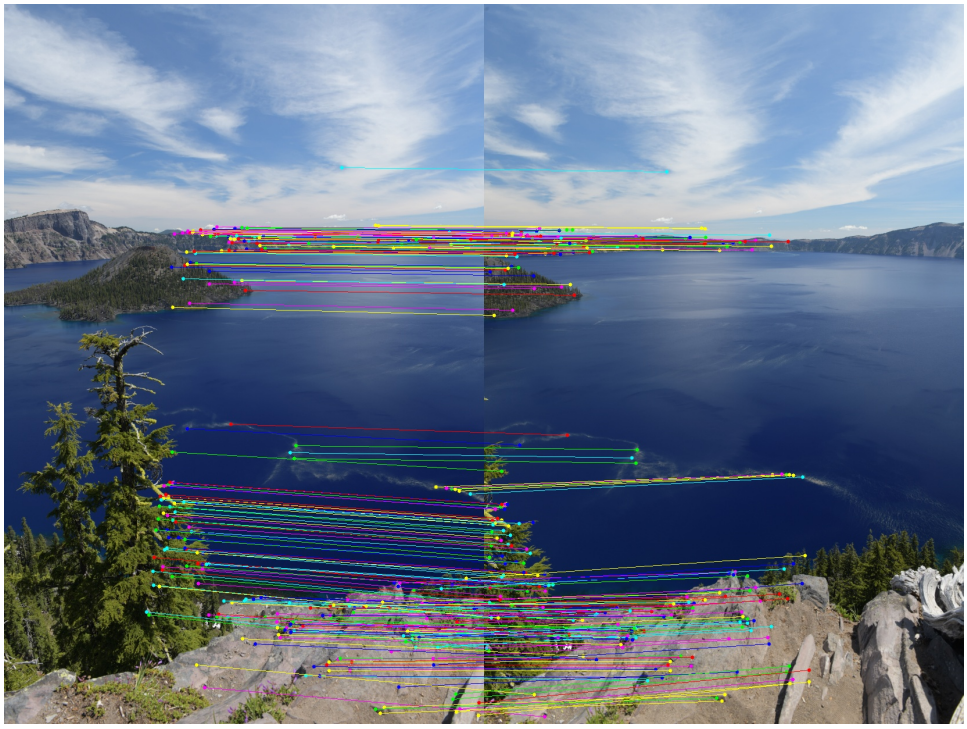

Figure 13: SIFT correspondences between images 3 and 4 (with lines).

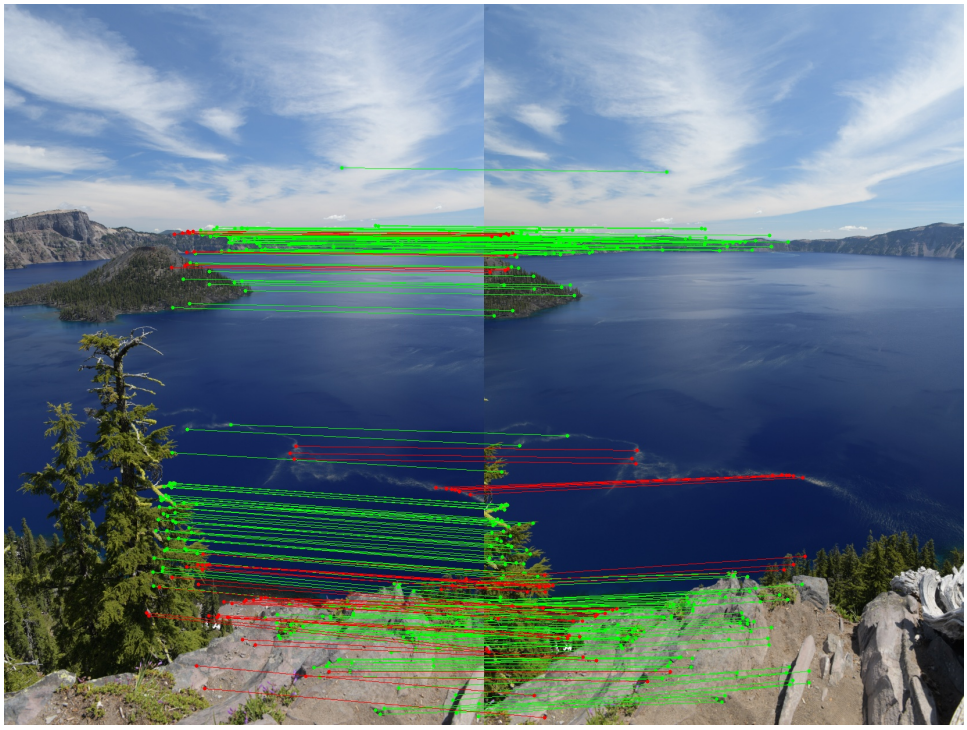

Figure 14: Inliers and outliers between images 3 and 4.

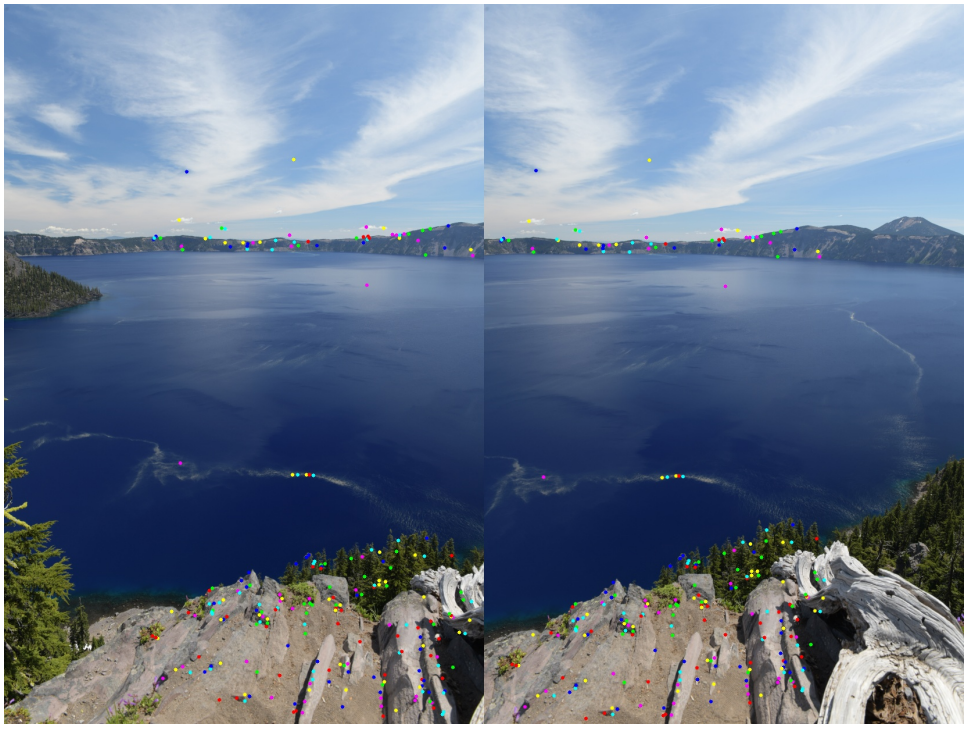

Figure 15: SIFT correspondences between images 4 and 5 (no lines for clarity).

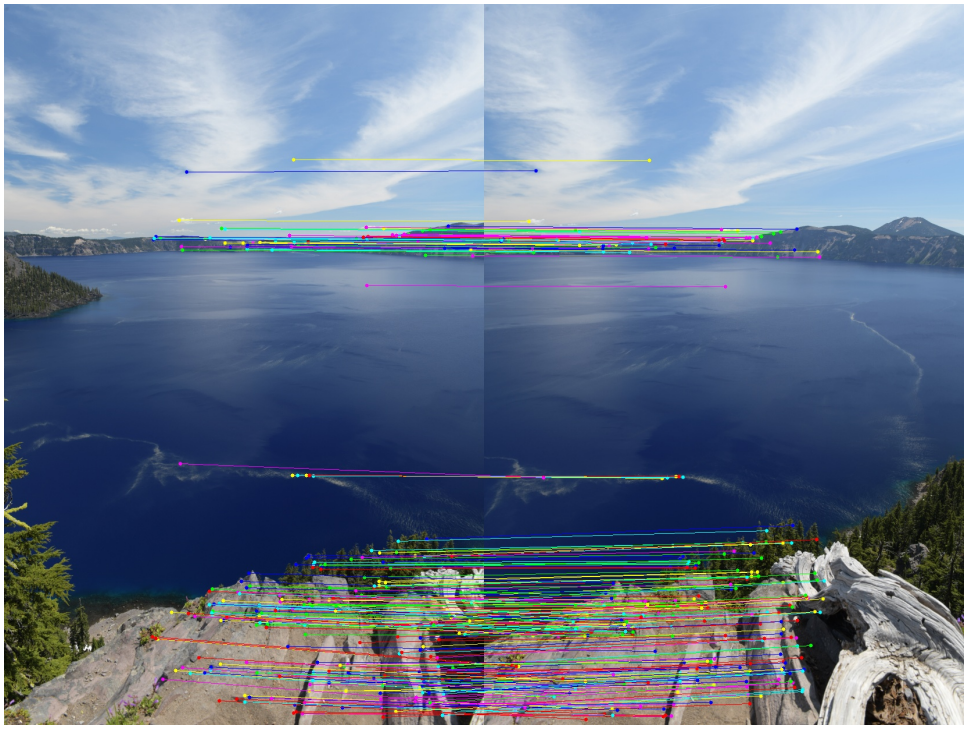

Figure 16: SIFT correspondences between images 4 and 5 (with lines).

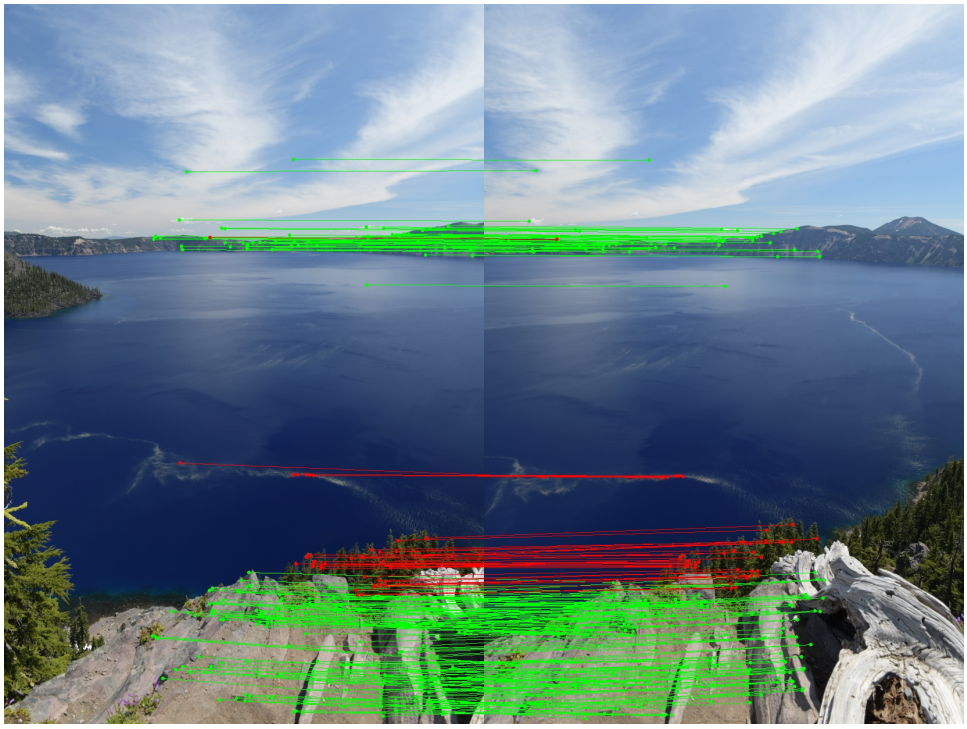

Figure 17: Inliers and outliers between images 4 and 5.

4.3 Final panoramas

Figure 18: Final panorama without LM refinement.

Figure 19: Final panorama with LM refinement.

5 Source Code

```
## ============= FILE INFORMATION ============= ##
#
# Name:         Brian Helfrecht
# Email:        bhelfre@purdue.edu
# Course:       ECE 661
# Assignment:   Homework 5, Task 1
# Due date:     October 5, 2020
#
## ============= PACKAGE/FILE IMPORTS ============= ##
import numpy as np
import matplotlib
import cv2 as cv
import scipy
from scipy import optimize
import math
import time
import random
## ============= FUNCTION DEFINITIONS ============= ##
def calcIntersection(ln1, ln2):
    #Calculate a HC point given 2 lines
    intersect = np.cross(ln1, ln2)
    return intersect / intersect[2]

def calcLine(pt1, pt2):
    #Calculate a HC line given two points
    if len(pt1) < 3:
        pt1.append(1.0)
    if len(pt2) < 3:
        pt2.append(1.0)
    line = np.cross(pt1, pt2)
    return line / line[2]

def findSIFTCorrespondences(pts1, img1Des, pts2, img2Des, maxMatches):
    #Find correspondences using SSD between
    #the descriptors of image 1 and image 2.

    img12Corr = np.zeros(len(img1Des), np.uint16) * -1
    img21Corr = np.zeros(len(img2Des), np.uint16) * -1
    img12Vals = np.zeros(len(img1Des), np.uint16) * -1
    img21Vals = np.zeros(len(img2Des), np.uint16) * -1

    #Similar to the process for the other algorithms, compare each descriptor
    #in image 1 to each in image 2, and calculate the SSD.
    for i in range(len(img1Des)):
        des = img1Des[i]
        sqDiffs = (des - img2Des) ** 2
        sumSqDiffs = np.sum(sqDiffs, 1)
        img12Corr[i] = np.argmin(sumSqDiffs)
        img12Vals[i] = np.min(sumSqDiffs)
    for i in range(len(img2Des)):
        des = img2Des[i]
        sqDiffs = (des - img1Des) ** 2
        sumSqDiffs = np.sum(sqDiffs, 1)
        img21Corr[i] = np.argmin(sumSqDiffs)
        img21Vals[i] = np.min(sumSqDiffs)

    #Extract the correspondences of only the points that exactly correspond
    #to each other
    correspondences = []
    vals = []
    for i in range(len(img12Corr)):
        if (img21Corr[img12Corr[i]] == i):
            pts1Idx = i
            pts2Idx = img12Corr[i]
            correspondences.append(np.around([pts1[pts1Idx].pt, pts2[pts2Idx].pt]).astype(int))
            vals.append(img12Vals[pts1Idx] + img21Vals[pts2Idx])

    #Sort the correspondences based on the smallest SSD values
    sortedCorr = [pt for _, pt in sorted(zip(vals, correspondences), key=lambda pair: pair[0])]
    correspondences = sortedCorr[0:maxMatches] #Take the maxMatches most matches
    return correspondences

def showCorrespondences(img1, img2, correspondences):
    #Find the image with the shortest height
    img1Height = img1.shape[0]
    img2Height = img2.shape[0]
    if (img1Height < img2Height): #Image 1 shorter
        img1 = np.concatenate((img1, np.zeros((img2Height-img1Height, img1.shape[1], 3))), 0)
    elif (img2Height < img1Height): #Image 2 shorter
        img2 = np.concatenate((img2, np.zeros((img1Height-img2Height, img2.shape[1], 3))), 0)

    noLines = np.concatenate((img1, img2), 1) #Stack images horizontally
    lines = noLines.copy() #Stack images horizontally
    img2OffsetX = img1.shape[1]

    #Establish possible line colors
    # RBGCMY (randomly assigned to differentiate)
    colors = ((255,   0,   0), #R
              (  0, 255,   0), #G
              (  0,   0, 255), #B
              (  0, 255, 255), #C
              (255,   0, 255), #M
              (255, 255,   0)) #Y

    #Draw correspondence points and lines between them
    for i in range(len(correspondences)):
        ptSet = correspondences[i] #Form: ((x1, y1), (x2, y2))
        colorIdx = i % len(colors)
        pt1 = tuple(ptSet[0])
        pt2 = tuple(np.array(ptSet[1]) + [img2OffsetX, 0])

        #Draw lines first to ensure points are visible on top
        cv.line(lines, pt1, pt2, colors[colorIdx], 1)
        cv.circle(lines, pt1, 3, colors[colorIdx], -1)
```

```
            cv.circle(lines, pt2, 3, colors[colorIdx], -1)
            cv.circle(noLines, pt1, 3, colors[colorIdx], -1)
            cv.circle(noLines, pt2, 3, colors[colorIdx], -1)

        return [noLines, lines]

    def showInliersOutliers(img1, img2, inliers, outliers):
        #Find the image with the shortest height
        img1Height = img1.shape[0]
        img2Height = img2.shape[0]
        if (img1Height < img2Height): #Image 1 shorter
            img1 = np.concatenate((img1, np.zeros((img2Height-img1Height, img1.shape[1], 3))), 0)
        elif (img2Height < img1Height): #Image 2 shorter
            img2 = np.concatenate((img2, np.zeros((img1Height-img2Height, img2.shape[1], 3))), 0)

        lines = np.concatenate((img1, img2), 1) #Stack images horizontally
        img2OffsetX = img1.shape[1]

        #Draw inliers
        for i in range(len(inliers)):
            ptSet = inliers[i] #Form: ((x1, y1), (x2, y2))
            pt1 = tuple(ptSet[0])
            pt2 = tuple(np.array(ptSet[1]) + [img2OffsetX, 0])

            #Draw lines first to ensure points are visible on top
            cv.line(lines, pt1, pt2, (0, 255, 0), 1)
            cv.circle(lines, pt1, 3, (0, 255, 0), -1)
            cv.circle(lines, pt2, 3, (0, 255, 0), -1)

        #Draw outliers
        for i in range(len(outliers)):
            ptSet = outliers[i] #Form: ((x1, y1), (x2, y2))
            pt1 = tuple(ptSet[0])
            pt2 = tuple(np.array(ptSet[1]) + [img2OffsetX, 0])

            #Draw lines first to ensure points are visible on top
            cv.line(lines, pt1, pt2, (0, 0, 255), 1)
            cv.circle(lines, pt1, 3, (0, 0, 255), -1)
            cv.circle(lines, pt2, 3, (0, 0, 255), -1)

        return lines

    def calcHomography(corrs):
        #Calculates the 3x3 homography matrix given corresponding domain and range points

        #Make sure at least 4 points are specified.
        numCorrs = len(corrs)
        if (numCorrs < 4):
            print('WARNING! At least 4 points are needed to calculate a homography!')
            return

        A = np.zeros((2 * numCorrs, 8))
        b = np.zeros((2 * numCorrs, 1))

        #Populate the A matrix and x vector using the correspondences
        for i in range(numCorrs):
            corr = corrs[i]
            domPt = corr[0]
            rngPt = corr[1]

            #Append x and y range image coordinates
            b[2*i]   = rngPt[0]
            b[2*i+1] = rngPt[1]

            #Append to A matrix to populate with equations
            A[2*i]   = np.array([domPt[0], domPt[1], 1, 0, 0, 0, \
                       -domPt[0]*rngPt[0], -domPt[1]*rngPt[0]])
            A[2*i+1] = np.array([0, 0, 0, domPt[0], domPt[1], 1, \
                       -domPt[0]*rngPt[1], -domPt[1]*rngPt[1]])

        pinvA = np.linalg.pinv(A) #Compute pseudo-inverse
        hVec  = np.dot(pinvA, b)  #Compute h
        hMat  = np.reshape(np.append(hVec, 1), (3, 3)) #Reshape
        return hMat

    def getInliers(H, corrs, delta):
        #First create an Nx3 matrix of domain points
        #We have to do this a bit slowly due to the way the correspondences are returned
        #in sorted form.
        img1Pts = np.zeros((len(corrs), 3), np.uint16)
        img2Pts = np.zeros((len(corrs), 3), np.uint16)

        #Create HC vectors for the correspondences in images 1 and 2
        for i in range(len(corrs)):
            img1Pts[i] = [*corrs[i][0], 1]
            img2Pts[i] = [*corrs[i][1], 1]

        #Calculate tranformed coordinates
        rngPts = np.transpose(np.dot(H, np.transpose(img1Pts)))
        rngPts[:, 0] = rngPts[:, 0] / rngPts[:, 2]
        rngPts[:, 1] = rngPts[:, 1] / rngPts[:, 2]
        #rngPts = np.ndarray.astype(np.round(rngPts), int)

        #Get physical coordinates (remove 'z' element)
        img2Pts = img2Pts[:, 0:2]
        rngPts = rngPts[:, 0:2]

        #Compute the distance between the transformed and inital range points
        diffs = (img2Pts - rngPts) ** 2
        dists = np.sqrt(np.sum(diffs, 1))

        #Get inliers and outliers
        inliers = np.array(corrs)[dists <= delta]
        outliers = np.array(corrs)[dists > delta]

        #Return the correspondences within the required distance
        return [inliers, outliers, np.mean(dists)]
```

```
def ransac(sigma, p, epsil, n, corrs):
    delta = 3 * sigma #Typical
    nTot  = len(corrs) #Total number of correspondences available
    N     = math.ceil(math.log(1 - p) / math.log(1 - ((1 - epsil) ** n))) #Trials to conduct
    M     = math.ceil((1 - epsil) * nTot) #Minimum size of acceptable inlier set
    print('nTot:', nTot)
    print('N:', N)
    print('M:', M)

    outlierMat = []
    inlierMat  = []
    inlierLen  = []

    for i in range(N): #Perform N trials
        selCorr = random.sample(corrs, n) #Randomly select n correspondences w/o replacement
        H = calcHomography(selCorr)
        [inliers, outliers, mean] = getInliers(H, corrs, delta)
        inlierLen.append(len(inliers))
        inlierMat.append(inliers)
        outlierMat.append(outliers)
        print('Inlier set %2d: Length = %d' % (i, len(inliers)))

    bestInlierIdx = np.argmax(inlierLen) #Most matches
    bestInlierLen = inlierLen[bestInlierIdx]
    print('Best inlier length:', bestInlierLen)
    if (bestInlierLen < M):
        print('WARNING! Best inlier set of length %d is less than M!' % (bestInlierLen))
    bestInlierSet = inlierMat[bestInlierIdx]
    bestOutlierSet = outlierMat[bestInlierIdx]
    return [bestInlierSet, bestOutlierSet]

def addFrame(pano, frame, H, appendRight, prevMinX = 0):
    #Calculate the transformed vertex positions of the new frame to determine how
    #to expand the panorama.
    [width, height] = [frame.shape[1], frame.shape[0]]
    vertices = ((0, 0), (0, height-1), (width-1, height-1), (width-1, 0))
    xVerts = []
    yVerts = []

    #Calculate transformed vertex coordinates
    for i in range(4):
        x = vertices[i][0]
        y = vertices[i][1]
        [newX, newY, newZ] = np.dot(H, [x, y, 1.0]) #World to Image
        xVerts.append(round(newX / newZ))
        yVerts.append(round(newY / newZ))

    #Determine the minimum coordinates to expand the frame
    frameTop = calcLine([0, 1], [10, 1])
    frameBot = calcLine([0, pano.shape[0]], [10, pano.shape[0]])

    #Expand the panorama
    if (appendRight):
        panoEdge = calcLine([xVerts[2], yVerts[2]], [xVerts[3], yVerts[3]])
        topIntersect = calcIntersection(frameTop, panoEdge)
        botIntersect = calcIntersection(frameBot, panoEdge)
        maxX = round(max(topIntersect[0], botIntersect[0]))
        minX = min(xVerts)
        panoExpansion = np.zeros((pano.shape[0], maxX - pano.shape[1], 3), np.uint8)
        newFrameRegion = np.zeros((pano.shape[0], maxX - minX, 3), np.uint8)
        pano = np.concatenate((pano, panoExpansion), 1) #Add the blank region

        #Now, update the added pixels with pixels in the frame
        newFrRegW = newFrameRegion.shape[1]
        newFrRegH = newFrameRegion.shape[0]
        xyIdxs = np.indices((newFrRegW, newFrRegH))
        xIdxs  = xyIdxs[0].reshape(newFrRegW*newFrRegH, 1) + minX
    else:
        panoEdge = calcLine([xVerts[0], yVerts[0]], [xVerts[1], yVerts[1]])
        topIntersect = calcIntersection(frameTop, panoEdge)
        botIntersect = calcIntersection(frameBot, panoEdge)
        minX = round(min(topIntersect[0], botIntersect[0]))
        maxX = max(xVerts)
        panoExpansion = np.zeros((pano.shape[0], abs(minX - prevMinX), 3), np.int16)
        newFrameRegion = np.zeros((pano.shape[0], abs(minX) + maxX, 3), np.int16)
        pano = np.concatenate((panoExpansion, pano), 1)

        #Now, update the added pixels with pixels in the frame
        newFrRegW = newFrameRegion.shape[1]
        newFrRegH = newFrameRegion.shape[0]
        xyIdxs = np.indices((newFrRegW, newFrRegH))
        xIdxs  = xyIdxs[0].reshape(newFrRegW*newFrRegH, 1) + minX


    yIdxs  = xyIdxs[1].reshape(newFrRegW*newFrRegH, 1)
    zIdxs  = np.ones((newFrRegW*newFrRegH, 1), np.uint8)
    finalIdxs = np.ndarray.astype(np.concatenate((xIdxs, yIdxs, zIdxs), 1), np.int16)
    invH = np.ndarray.astype(np.linalg.pinv(H), np.float32)
    newIdxs = np.transpose(np.dot(invH, np.transpose(finalIdxs)))
    newIdxs[:, 0] = newIdxs[:, 0] / newIdxs[:, 2]
    newIdxs[:, 1] = newIdxs[:, 1] / newIdxs[:, 2]
    newIdxs = np.ndarray.astype(np.round(newIdxs), int)

    #Trim rows that correspond to points outside the domain image
    finalIdxs = finalIdxs[newIdxs[:, 0] >= 0] #Trim any x < 0
    newIdxs = newIdxs[newIdxs[:, 0] >= 0] #Trim any x < 0
    finalIdxs = finalIdxs[newIdxs[:, 1] >= 0] #Trim any y < 0
    newIdxs = newIdxs[newIdxs[:, 1] >= 0] #Trim any y < 0
    finalIdxs = finalIdxs[newIdxs[:, 0] < width] #Trim any x > width
    newIdxs = newIdxs[newIdxs[:, 0] < width] #Trim any x > width
    finalIdxs = finalIdxs[newIdxs[:, 1] < height] #Trim any y > height
    newIdxs = newIdxs[newIdxs[:, 1] < height] #Trim any y > height

    if (not appendRight):
        finalIdxs[:, 0] = finalIdxs[:, 0] - minX
```

```
    #Iterate over the remaining pixels. I could not find a better method for this (yet)
    for row in range(np.size(newIdxs, 0)):
        pano[finalIdxs[row, 1]][finalIdxs[row, 0]] = frame[newIdxs[row, 1]][newIdxs[row, 0]]

    return [pano, minX]

def costFunc(h, inliers):
    #Minimize E = |X - F|^2
    X = []
    F = []

    #Create HC vectors for the correspondences in images 1 and 2
    for i in range(len(inliers)):
        xd = inliers[i][0][0] #x points
        yd = inliers[i][0][1]
        xr = inliers[i][1][0] #x' points
        yr = inliers[i][1][1]

        #Append range points
        X.append(xr)
        X.append(yr)

        #Append mappings (points transformed through the homography) to F
        F.append((h[0]*xd + h[1]*yd + h[2]) / (h[6]*xd + h[7]*yd + h[8]))
        F.append((h[3]*xd + h[4]*yd + h[5]) / (h[6]*xd + h[7]*yd + h[8]))

    #We only need to return the residual vector, not its norm
    return np.array(X) - np.array(F)


## ============= MAIN CODE BEGINS BELOW ============= ##
#Print version information to verify library loads
print('OpenCV Version:', cv.__version__)

''' ============= TASK 1.1 ============= '''

hMats = np.zeros((5, 3, 3))
useLM = True

for imgNum in range(1, 6): #1-6
    print('=======================================')
    print('Image pair %d-%d:' % (imgNum, imgNum+1))
    #Read in the input images for the image pair
    img1 = cv.imread('inputs/%d.jpg' % (imgNum))
    img2 = cv.imread('inputs/%d.jpg' % (imgNum+1))

    #Use grayscale images as inputs to SIFT
    gray1 = cv.cvtColor(img1, cv.COLOR_BGR2GRAY)
    gray2 = cv.cvtColor(img2, cv.COLOR_BGR2GRAY)

    #Calculate interest points and descriptors using SIFT
    cvSift = cv.SIFT.create()
    print('\tFinding SIFT points in image 1... ')
    [pts1, des1] = cvSift.detectAndCompute(gray1, None)
    print('\tFinding SIFT points in image 2... ')
    [pts2, des2] = cvSift.detectAndCompute(gray2, None)

    #Find correspondences in SIFT using SSD
    print('\tFinding correspondences... ')
    siftCorr = findSIFTCorrespondences(pts1, des1, pts2, des2, 250)
    #[noLines, lines] = showCorrespondences(img1, img2, siftCorr)
    #cv.imwrite('outputs/%d-%d_sift_nolines.jpg' % (imgNum, imgNum+1), noLines)
    #cv.imwrite('outputs/%d-%d_sift_lines.jpg' % (imgNum, imgNum+1), lines)

    #Apply RANSAC
    print('Applying RANSAC...')
    [inliers, outliers] = ransac(2, 0.99, 0.25, 6, siftCorr)
    inOutImg = showInliersOutliers(img1, img2, inliers, outliers) #Show the inliers/outliers
    cv.imwrite('outputs/%d-%d_inout.jpg' % (imgNum, imgNum+1), inOutImg)

    #Calculate the optimal homography between the image pair
    optH = calcHomography(inliers)

    if (useLM):
        optH = optH.reshape((1, 9))[0]
        optH = optimize.least_squares(costFunc, optH, args = [inliers], method = 'lm').x
        optH = optH.reshape(3, 3)

    hMats[imgNum-1] = optH

#Now we need to create the final panorama using the homographies
#Using the center image as the anchor helps to eliminate very distorted images
print(hMats)

#Read in the fresh images and store in a list
images = []
for i in range(1, 7):
    img = cv.imread('inputs/%d.jpg' % (i))
    images.append(img)

anchorImg = round(len(images) / 2.0) #Determine the anchor image
print('Anchor image: %d' % anchorImg)

#Transform each pairwise homography to put all images in the frame of the anchor
#Our homographies were constructed from img1 -> img2, so use the inverse to go backward
for i in range(anchorImg - 2, -1, -1): #Images up to anchor
    hMats[i-1] = np.dot(hMats[i], hMats[i-1])
hMats = np.insert(hMats, anchorImg-1, np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]]), 0)
for i in range(anchorImg, len(images)): #Images after anchor
    hMats[i] = np.dot(hMats[i-1], np.linalg.pinv(hMats[i]))

print(hMats)
#Now our hMats will transform each image to the domain of the anchor

if (useLM):
    outFilename = 'outputs/LM_pano'
else:
```

```
    outFilename = 'outputs/pano'

#Now, using the left-most image as the anchor, stitch the panorama
pano = images[anchorImg-1]
for i in range(anchorImg, 5): #1-6
    currFrame = images[i]
    [pano, _] = addFrame(pano, currFrame, hMats[i], True)
    print('Writing pano pair %d...'% (i))
    cv.imwrite(outFilename + '%d.jpg' % (i), pano)

prevMinX = 0
for i in range(anchorImg-2, -1, -1): #1-6
    currFrame = images[i]
    [pano, prevMinX] = addFrame(pano, currFrame, hMats[i], False, prevMinX)
    print('Writing pano pair %d...'% (i))
    cv.imwrite(outFilename + '%d.jpg' % (i), pano)
```