ECE 661: Computer Vision

Homework 5, Fall 2018

Arindam Bhanja Chowdhury
abhanjac@purdue.edu

1 Overview

This homework is about finding the homography between the images automatically using interest points with the Linear Least Squares algorithm and then to find a robust estimate of the homography using the RANSAC algorithm. This homography is then refined using the Levenberg-Marqaudt algorithm and used to stitch a sequence of images together to form one single image.

The sequences of 5 images used for this homework is shown in Fig 1 to 5. These image have some overlaps among them.

Figure 1: Reference image 1.

Figure 2: Reference image 2.

Figure 3: Reference image 3.

Figure 4: Reference image 4.

Figure 5: Reference image 5.

2 Overview of SIFT

This will be a very brief overview of the Scale Invariant Feature Transform (SIFT) algorithm. This is one of the most popular interest point detectors. It finds the interest points using the following steps.

- **Constructing a scale space:** This is the initial preparation. We create internal representations of the original image to ensure scale invariance. This is done by generating a "scale

space".

- **LOG Approximation:** The Laplacian of Gaussian is great for finding interesting points (or key points) in an image. But it's computationally expensive. So we approximate it using the Difference of Gaussian (DOG).
- **Finding keypoints:** With the super fast approximation, we now try to find key points. These are maxima and minima in the Difference of Gaussian image we calculate in step 2.
- **Get rid of bad key points** Edges and low contrast regions are bad keypoints. Eliminating these makes the algorithm efficient and robust. A technique similar to the Harris Corner Detector is used here.
- **Assigning an orientation to the keypoints** An orientation is calculated for each key point. Any further calculations are done relative to this orientation. This effectively cancels out the effect of orientation, making it rotation invariant.
- **Generate SIFT features** Finally, with scale and rotation invariance in place, one more representation is generated. This helps uniquely identify features. Lets say you have 50,000 features. With this representation, you can easily identify the feature you're looking for.

That was an overview of the entire algorithm. We will be using the SIFT package in opencv for implementing the algorithm. The features or interest points are extracted from each of the images using SIFT and are matched between consecutive images in the sequence. The matched interest points are shown in the images in the Result section.

3 Linear Least Squares Minimization Algorithm

For estimating the 3x3 homography H that relates the points in two different planes using the equation $X' = HX$, the equation is first written in the following format.

$$X' = HX = \begin{bmatrix} h^{1T}X \\ h^{2T}X \\ h^{3T}X \end{bmatrix} \tag{1}$$

where the $h^{1T} = [h_{11}h_{12}h_{13}]$, $h^{2T} = [h_{21}h_{22}h_{23}]$, $h^{3T} = [h_{31}h_{32}h_{33}]$ are the rows of the H matrix.
Now under the assumption that the correspondence between X' and X are error free, it can be written that the cross product between X' and HX is 0.
So this can be expanded as

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} \times \begin{bmatrix} h^{1T}X \\ h^{2T}X \\ h^{3T}X \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{2}$$

Now this equation can be rewritten as

$$\begin{bmatrix} 0^T h^1 - w' X^T h^2 + y' X^T h^3 \\ w' X^T h^1 + 0^T h^2 - x' X^T h^3 \\ -y' X^T h^1 + x' X^T h^2 + 0^T X^T h^3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{3}$$

Now since the above equations are not independent, the 3rd equation is removed from the group to have the following set of 2 equations

$$\begin{bmatrix} 0^T & -w'X^T & y'X^T \\ w'X^T & 0^T & -x'X^T \end{bmatrix} \begin{bmatrix} h^1 \\ h^2 \\ h^3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{4}$$

So, each of the correspondence between X and X' will give two such equation as above.

Now N different correspondence will give 2N equation which creates an overall system of equation which can be written in the form like the following

$$A_{m\times n} h_{n\times 1} = 0_{m\times 1} \tag{5}$$

Where $A_{m\times n}$ is the matrix containing the elements of X and X'. m = 2N here and n = 9 here, as the h vector has 9 elements.

So (5) is a set of homogeneous equation which can have different types of solutions based on the rank of A.

The only condition that if of interest to us is when the rank of A is n-1 and m ¿= n. Then this will become an over-determined system and there will be theoretically no solution, but the solution that will make this system closest to 0 will give the best solution, provided $\| h \| = 1$ i.e. h is not 0 (which is the trivial solution case).

This is referred to as the Linear Least Squares Minimization (LLSM) algorithm for homogeneous equations.

The solution for LLSM algorithm is given by that eigen vector of $A^T A$ which corresponds to the smallest eigen value. If U, D and V be the vectors obtained by the singular value decomposition of A, then the last column vector of V is the solution for h.

4 RANSAC Algorithm

The Ransac algorithm is used for a robust estimation of homography that was obtained from the LLSM algorithm. There are a number of correspondences in LLSM which are too noisy (known as inliers) or are completely wrong (known as outliers). Ransac algorithm states that if all the correspondences are selected randomly and repeatedly, then there will be a few such combinations among them that does represent the correct homography.

So what Ransac does is use a randomly selected least amount of data to construct an estimate of the homography and then ascertain the extent of support that the rest of the data provides to the estimate. The estimate is accepted only if the support exceeds a threshold. When the estimate made in this manner is accepted, then the supporting data is called **inliers** and the unsupporting correspondences are called the **outliers**.

There are three parameters that are important for the Ransac algorithm.

A **decision threshold** to construct the inlier set. When the points are compared with the estimates, if the distance between the two is less than the threshold, the point is placed in the inlier set.

A value of N which is the **number of trials**. It is often computationally infeasible to attempt all possible trials.

A **minimum value** for the size of the inlier set for it to be acceptable.

In most cases these parameters are experimentally determined.

4.1 Distance Threshold (δ)

It will be common to assume that for the inliers the noise-induced displacement of the true location of a pixel would be modelled by the Gaussian

$$g(\Delta x, \Delta y) = \frac{1}{2\pi\sigma^2} exp^{\frac{-(\Delta x^2, \Delta y^2)}{2\sigma^2}} \tag{6}$$

where σ is set to a value between 0.5 and 2.

The $\delta = 3\sigma$ is considered to be the distance threshold to accept a correspondence to be an inlier.

4.2 Number of Iterations (N)

Let ϵ be the probability that a randomly chosen (X, X') is an outlier. Then $1 - \epsilon$ is the probability of a correspondence to be an inlier.

So the probability of all n correspondences chosen for estimating H in a given trial are all inliers is $(1 - \epsilon)^n$.

So in a given trial the probability that at least one of the correspondence is an outlier is given by $1 - (1 - \epsilon)^n$.

Therefore the probability that every one of the N iterations will involve at least one outlier in the calculation of H is $[1 - (1 - \epsilon)^n]^N$.

So the probability (p) that at least one of the N trials will be free of outliers in the calculation of H is

$$p = 1 - [1 - (1 - \epsilon)^n]^N$$

So the equation of N is given as

$$N = \frac{ln(1 - p)}{ln[1 - (1 - \epsilon)^n]} \tag{7}$$

4.3 Minimum Size of Inlier Set (M)

The M should be roughly the same size as the number of true inliers in the data. So if ϵ is the probability that a correspondence is an outlier, and if n_{total} is the total number of correspondences between the two images, then,

$$M = (1 - \epsilon)n_{total} \tag{8}$$

These (6), (7) and (8) are used for determining the parameters of the Ransac algorithm.

5 Levenberg-Marquardt Algorithm

Once a good estimate of the homography is obtained from Ransac, the Levenberg-Marqardt (LM) algorithm is used to refine this estimate. LM algorithm is a combination of the Gradient Descent (GD) algorithm and the Gauss Newton (GN) method. So first a little introduction of GD and GN is needed before explaining LM algorithm.

5.1 Gradient Descent Algorithm

If $C(p)$ is a surface defined over an n-dimensional space, then our goal is to start from a point p_0 and take small steps from p_0 to p_1 to p_2 and finally reach a point that is the lowest in $C(p)$.

In $C(p)$ given that the current point is p_k, where the value of the surface is $C(p_k)$, the next step is taken towards the point p_{k+1} and this is given as

$$p_{k+1} = p_k - \gamma \nabla C$$

∇C is the gradient of $C(p)$.

The $C(p)$ is usually the error between the true value of a quantity X and the value of the same quantity estimated by some function $f(p)$.

So,

$$C(p) = (X - f(p))^2 = \epsilon(p)\epsilon(p)$$

So, p_{k+1} can be written as

$$p_{k+1} = p_k + 2\gamma J_f(p_k)\epsilon(p)$$

Where $J_f(p_k)$ is the jacobian of the vector function $\epsilon(p)$.

But if GD is used alone, then as it goes closer to the minima, it takes increasingly smaller steps that makes the convergence slower.

5.2 Gauss Newton Method

The GN method on the other hand gives a solution in a single step. But it only works if the approximation is already very close to the true minimum value. So assuming that the current solution is at point p_k, and the solution is at p_{k+1}, the above equation of X can be written as

$$X - f(p_k) = f(p_k) + J_f(p_k)\delta_p$$

So,

$$\epsilon(p) = X - f(p_k) = J_f(p_k)\delta_p$$

And hence using the pseudoinverse, it can be written that

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

But this will only give a good solution if the estimated output is already close to the true solution.

5.3 Levenberg-Marquardt Method

This method combines the good of both the GD and GN methods. The solution to the LM method can be written as

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

It can be seen that the equation is similar to both the equation for GD and GN. The μI term is the damping term.

Starting with an initial guess p_0 at each iteration k consists of first setting a value for the damping coefficient μ_k and then setting δ_p to

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

The new solution will be $p_{k+1} = p_k + \delta_p$.

The real challenge of this algorithm is to find out how to set the value of the damping coefficient at each iteration. The best μ to use, depends on δ_p. At the point p_k first δ_p is calculated and then a quality of the δ_p is checked. If the δ_p does not pass the proper quality check then the algorithm reverts back to the old solution i.e. $p_{k+1} = p_k$. And the μ_{k+1} is set to $2\mu_k$.

The quality of a computed δ_p is tested using the ration of the actual change in the cost function to the change in the cost function predicted by a particular choice for μ_k. The denominator of this ration requires us to express the change in the cost function in terms of μ_k.

The ratio is shown by the following formula

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

The value of ρ_{k+1} thus obtained is used to calculate the damping coefficient for the next iteration through the following formula.

$$\mu_{k+1} = \mu_k . max[\frac{1}{3}, 1 - (\rho_{k+1} - 1)^3]$$

For initializing the sequence of damping coefficients the very first value μ_0 is estimated by setting it to

$$\mu_0 = \tau . max[diag(J_f^T J_f)]$$

where $0 < \tau <= 1$.

This algorithm steers its path towards GD whenever it suspects that the LM is not doing a good job. This is because the GD has a safer convergence even if it takes a long time. But GN may have a faster convergence, but it can behave erratically if the parameters are wronly chosen.

6 Procedure

- First the interest points are extracted using SIFT algorithm from the sequence of images and then they are matched between consecutive images using their euclidean distances.
- Out of all the matches a set of best ones are selected based on a certain minimum threshold for the euclidean distance.
- This set of best correspondences usually contains a group of noisy correct correspondences and a group of wrong correspondences.
- The Ransac algorithm is applied to find the best inlier set. For each trial the number of correspondences used is between 4 and 10..
- After going through N trials, the homography with maximal inlier support is retained as the best estimate of homography.
- After this, this best estimate is refined using the Levenberg-Marquardt algorithm to find a more refined version of the homography.

7 Results

Figure 6: SIFT interest point matches between image 1 and image 2.

Good matches: 135 out of 1351 total matches (with threshold: 10% of total matches (after sorting in ascending order of euclidean distances)).

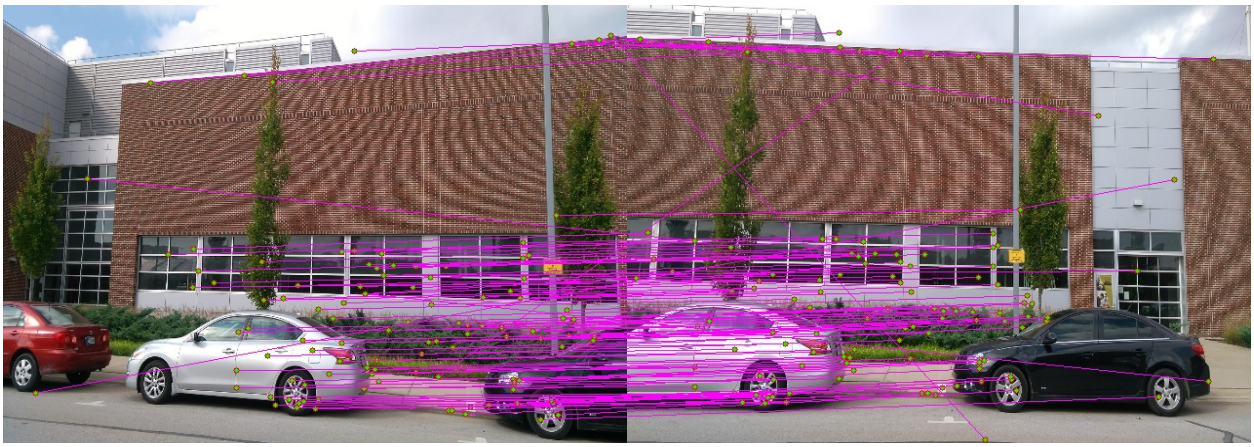

Figure 7: SIFT interest point matches between image 2 and image 3.

Good matches: 133 out of 1334 total matches (with threshold: 10% of total matches (after sorting in ascending order of euclidean distances)).

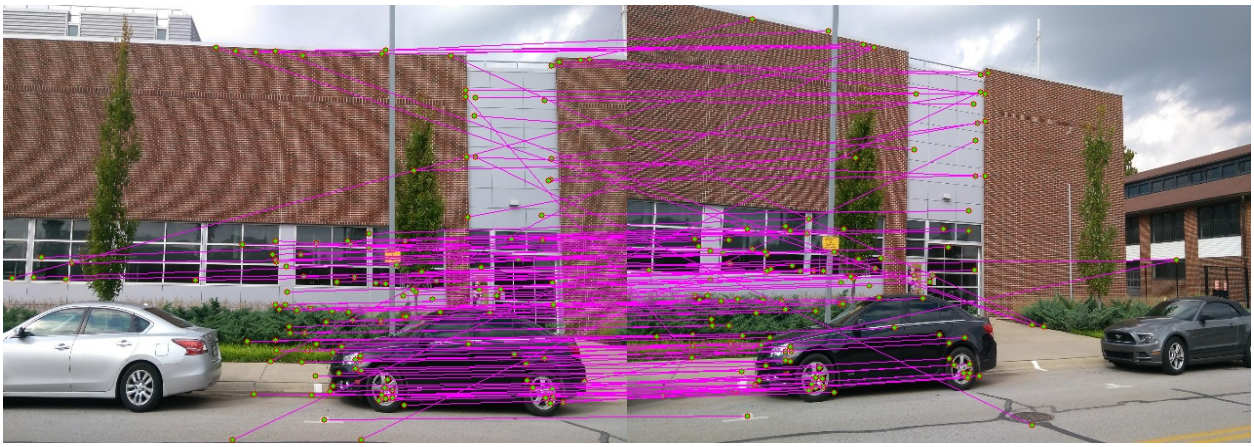

Figure 8: SIFT interest point matches between image 3 and image 4.

Good matches: 133 out of 1334 total matches (with threshold: 10% of total matches (after sorting in ascending order of euclidean distances)).

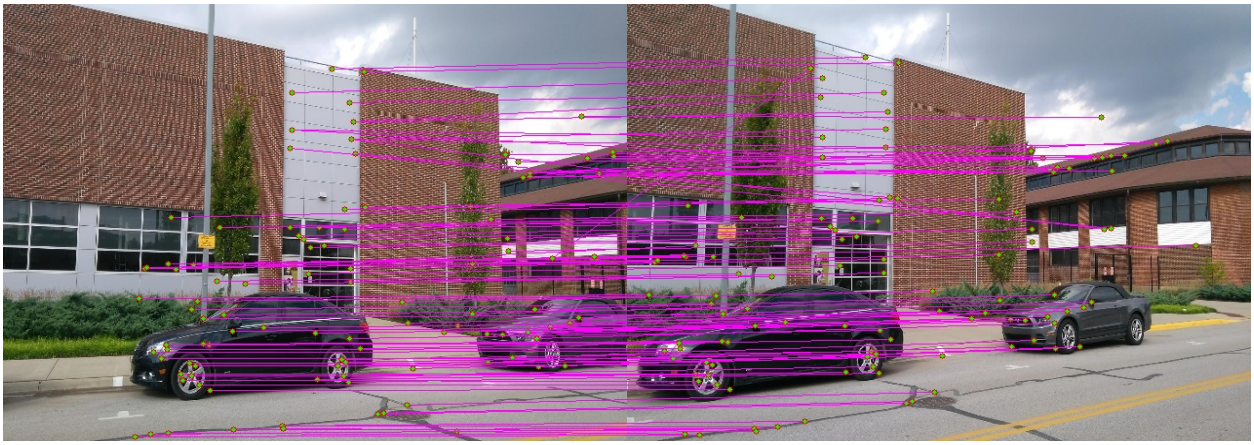

Figure 9: SIFT interest point matches between image 4 and image 5.

Good matches: 123 out of 1239 total matches (with threshold: 10% of total matches (after sorting in ascending order of euclidean distances)).

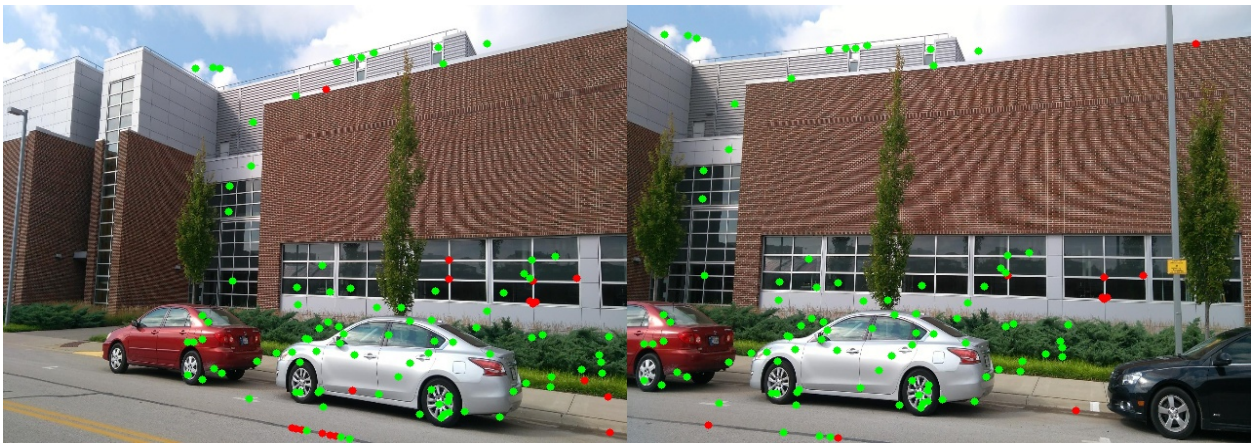

Figure 10: Inliers (GREEN) and Outliers (RED) between image 1 and image 2.

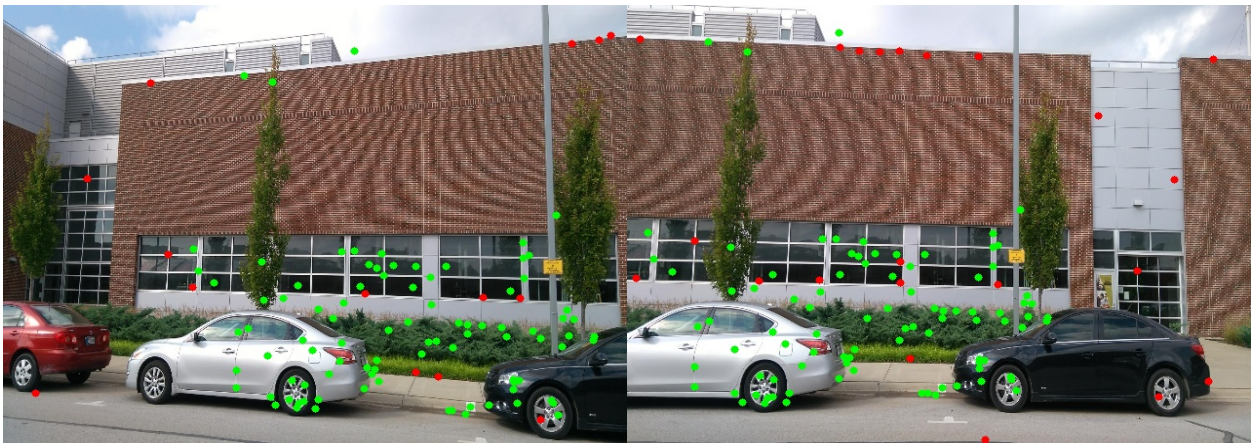

Figure 11: Inliers (GREEN) and Outliers (RED) between image 2 and image 3.

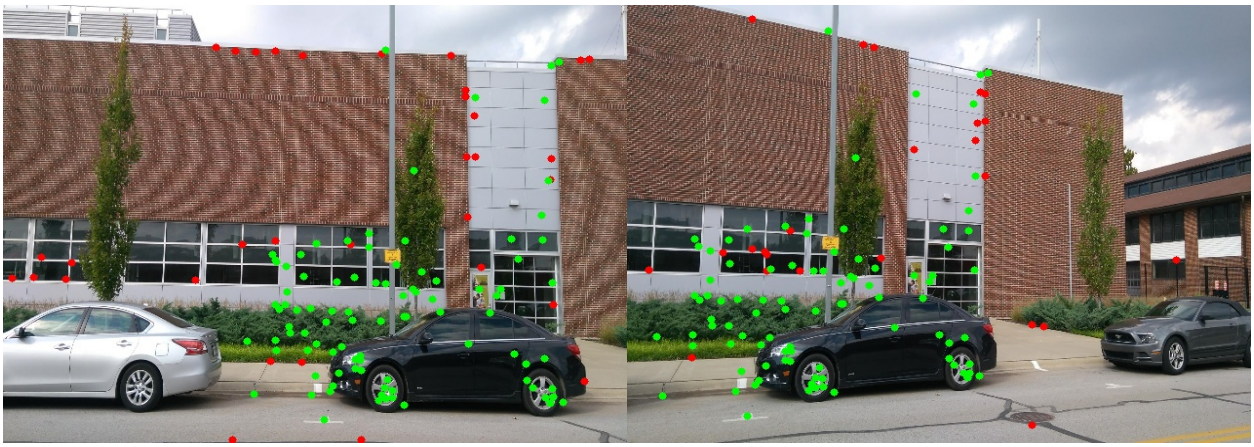

Figure 12: Inliers (GREEN) and Outliers (RED) between image 3 and image 4.

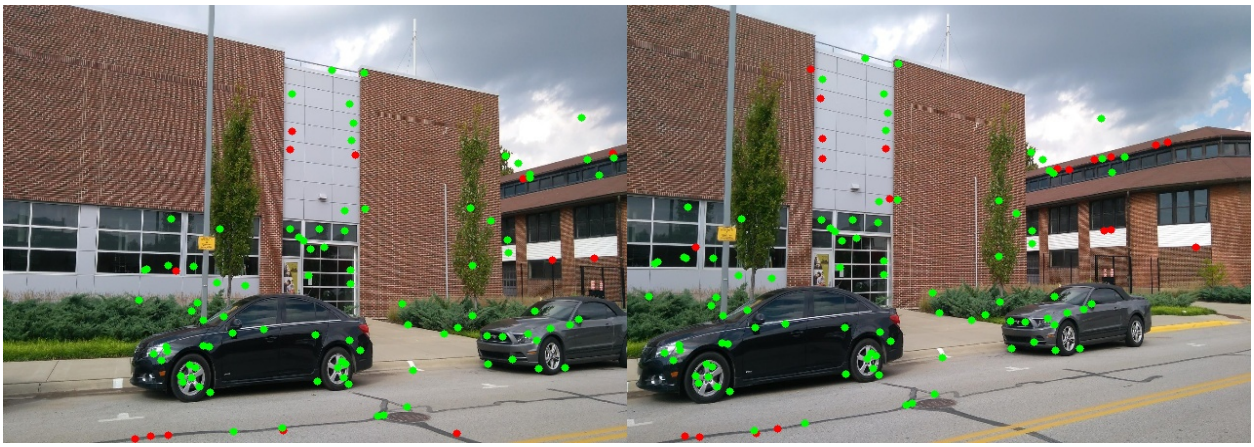

Figure 13: Inliers (GREEN) and Outliers (RED) between image 4 and image 5.

Parameters	**Values**
n	8
σ	2
δ	$3\times\sigma = 6$
ϵ	0.3
p	0.9999
N	155 (from (7))

Table 1: Parametes used for RANSAC

Figure 14: Stitched output using the homography calculated out of RANSAC.

Figure 15: Stitched output using the homography of RANSAC refined by the Levenberg-Marquadt Algorithm.

8 Comments

- We have used the Linear Least Squares Minimzation algorithm for homogeneous equations, not the Linear Least Squares for inhonogeneous equations with pseudoinverse. This is because the later assumes that the last element of the matrix $h_{33} = 1$, which may be incompatible with the case when the homography may map the origin in one plane to a point on l_∞ in the other plane.
- For this code we have used the open source python package of LM algorithm implemented in scipy.
- Increasing p will increase N, as more iterations will be needed to have better accuracy to remove outliers. Also if ϵ increases, the that means probability of outliers is more, hence N has to increase to better filter out the outliers.
- There was not a very appreciable difference in how the final stitched image looks like in case sof RANSAC and in the case of RANSAC and LM algorithm. This is because the homography obtained from RANSAC was very close to that obtained after refining it by the LM algorithm.

```python
#!/usr/bin/env python

import numpy as np, cv2, os, time, math, copy
from scipy import signal, optimize

#================================================================================
# ARINDAM BHANJA CHOWDHURY
# abhanjac@purdue.edu
# ECE 661 FALL 2018, HW 5
#================================================================================

#================================================================================
# FUNCTIONS CREATED IN HW2.
#================================================================================

#================================================================================

# Converts an input point (x, y) into homogeneous format (x, y, 1).
homogeneousFormat = lambda pt: [ pt[0], pt[1], 1 ]

#================================================================================

# Converts an input point in homogeneous coordinate (x, y, z) into planar form.
planarFormat = lambda pt: [ int(pt[0] / pt[2]), int(pt[1] / pt[2]) ]

#================================================================================

# Converts an input point in homogeneous coordinate (x, y, z) into planar form,
# But does not round them into integers, keeps them as floats.
planarFormatFloat = lambda pt: [ pt[0] / pt[2], pt[1] / pt[2] ]

#================================================================================

# The input is a 2 element list of the homography matrix H and the point pt.
# [H, pt]. Applies H to the point pt. pt is in homogeneous coordinate form.
applyHomography = lambda HandPt: np.matmul( HandPt[0], \
                                            np.reshape( HandPt[1], (3,1) ) )

#================================================================================

#================================================================================
# FUNCTIONS CREATED IN HW3.
#================================================================================

def dimsOfModifiedImg( img=None, H=None ):
    '''
    This function takes in an entire image and a homography matrix and returns
    what the dimensions of the output image (after applying this homography)
    should be, to accomodate all the modified point locations along with the
    min and max x and y coordinates of the modified image.
    '''

    imgH, imgW, _ = img.shape

    # The output image will be of a different dimension than the input image.
    # But the corner points will always stay contained inside the final image.
    # In order to find the dimensions of the final image we find out where the
    # corners of the input image will be mapped.

    tlc = [ 0, 0, 1 ]   # Top left corner of the image. (x, y, 1 format)
    trc = [ imgW, 0, 1 ]   # Top right corner of the image. (x, y, 1 format)
    brc = [ imgW, imgH, 1 ]   # Bottom right corner of the image. (x, y, 1 format)
    blc = [ 0, imgH, 1 ]   # Bottom left corner of the image. (x, y, 1 format)

    # Applying homography.
```

```
    tlcNew = applyHomography( [H, tlc] )    # Top left corner in new image.
    tlcNew = planarFormat( tlcNew )
    trcNew = applyHomography( [H, trc] )    # Top right corner in new image.
    trcNew = planarFormat( trcNew )
    brcNew = applyHomography( [H, brc] )    # Bottom right corner in new image.
    brcNew = planarFormat( brcNew )
    blcNew = applyHomography( [H, blc] )    # Bottom left corner in new image.
    blcNew = planarFormat( blcNew )

    # Making a list of the x and y coordinates of the corner locations in new image.
    listOfX = [ tlcNew[0], trcNew[0], brcNew[0], blcNew[0] ]
    listOfY = [ tlcNew[1], trcNew[1], brcNew[1], blcNew[1] ]

    maxX, maxY = max( listOfX ), max( listOfY )
    minX, minY = min( listOfX ), min( listOfY )

    # Now so that the minX and minY map to 0, 0 in the modified image, so those
    # locations should be subtracted from the max locations.
    # If maxX is 5 and minX is -2, then image should be between 0 and 5 - (-2) = 7.
    # If maxX is 5 and minX is 1, then image should be between 0 and 5 - (1) = 4.
    # The +1 is because if the max col, row index is 639, 479, then image is 640 x 480.
    dimX, dimY = maxX - minX + 1, maxY - minY + 1

    return [ dimX, dimY, maxX, maxY, minX, minY ]

#================================================================================

#================================================================================
# FUNCTIONS CREATED IN HW4.
#================================================================================

def distanceSift( kp1=None, des1=None, kp2=None, des2=None, matchThresh=None ):
    '''
    This calculates the euclidean distance between two keypoint descriptors for sift.
    keypoints are lists of keypoint object in opencv and descriptors are numpy arrays
    with 128 columns (for sift), where each of the rows represent the 128 element
    vector for the corresponding keypoint.
    '''

    # Taking the smaller set among the two listOfCorners. Such that the
    # no. of corners in A image is always less than that of B image.
    if len( kp1 ) < len( kp2 ):
        kpA, desA, kpB, desB = kp1, des1, kp2, des2
        Ais1 = True     # Shows if desA is des1 or not.
        # Used later to return matched pair of points.
    else:
        kpA, desA, kpB, desB = kp2, des2, kp1, des1
        Ais1 = False     # Shows if desA is des1 or not.
        # Used later to return matched pair of points.

    # Creating lists of same lenth as kpA
    matchedKpB = np.ones( len(kpA) ).tolist()
    distValue = np.ones( len(kpA) ).tolist()

#--------------------------------------------------------------------------------

    # Scanning all the descriptors of A and finding the euclidean distance with
    # each of B to get the match (i.e. the descriptor in B with which it has the
    # minimum ssd value).
    for idxa, a in enumerate( desA ):
        minDist = 100000
        for idxb, b in enumerate( desB ):
            dist = (a - b) * (a - b)
            dist = np.sqrt( np.sum( dist ) )
```

```
            if minDist > dist:
                minDist = dist    # Calculating the minimum euclidean distance value.
                pt = [ int(kpB[ idxb ].pt[0]), int(kpB[ idxb ].pt[1]) ]
                matchedKpB[ idxa ] = pt    # Storing best match.
                distValue[ idxa ] = minDist      # Storing the distance values.

#-----------------------------------------------------------------------------

    # The matchedPairs1To2 list stores the tuple of matched points of image 1 to 2.
    # And not the other way round. So the first element of each of the tuples is
    # a keypoint of image 1 and the 2nd element is a keypoint of image 2.
    # This is done using the flag Ais1.
    if Ais1:
        matchedPairs1To2 = [ ( [ int(kpA[i].pt[0]), int(kpA[i].pt[1]) ], matchedKpB[i] ) \
                                for i in range( len( kpA ) ) ]
    else:
        matchedPairs1To2 = [ ( matchedKpB[i], [ int(kpA[i].pt[0]), int(kpA[i].pt[1]) ] ) \
                                for i in range( len( kpA ) ) ]

    # Sorting the list with ascending order of distance value, such that the best
    # lowest distance value is at the beginning.
    matchedPairs1To2 = sorted( matchedPairs1To2, \
                               key=lambda x: distValue[ matchedPairs1To2.index(x) ] )
    distValue = sorted( distValue )

    # If no matchThresh is specified then all points are returned.
    # If a percentage in the form of '10%' is specified, then 10% of the total
    # number of matched points will be returned.
    if matchThresh == None:     matchThresh = 100000
    elif type( matchThresh ) == str:
        threshDistIdx = int( float( matchThresh[:-1] ) * len( matchedPairs1To2 ) / 100 )
        matchThresh = distValue[ threshDistIdx ]
    else:
        print( '\nERROR. Invalid matchThresh value. Aborting.' )
        return

    #print(matchThresh)

    goodMatches1to2 = [ matchedPairs1To2[i] for i in range( len(matchedPairs1To2) ) \
                                    if distValue[i] < matchThresh ]

    return matchedPairs1To2, goodMatches1to2, distValue

#=============================================================================

#=============================================================================
# FUNCTIONS CREATED IN HW5.
#=============================================================================

def findHomographyByLLSM( srcPtList=None, dstPtList=None ):
    '''
    This function finds the homography by Linear Least Square Minimization method.
    The points in the above lists should be in homogeneous coordinates form.
    '''
    nSrcPts, nDstPts = len( srcPtList ), len( dstPtList )

    if nSrcPts < 4 or nDstPts < 4:
        print( '\nERROR. Number of points should be at least 4. Aborting.' )
        return

    # Creating the A matrix.
    A = []
    for i in range( nSrcPts ):
        xs, ys, ws = srcPtList[i][0], srcPtList[i][1], srcPtList[i][2]
        xd, yd, wd = dstPtList[i][0], dstPtList[i][1], dstPtList[i][2]
```

```
        A.append( [ 0, 0, 0, -wd * xs, -wd * ys, -wd * ws, yd * xs, yd * ys, yd * ws ] )
        A.append( [ wd * xs, wd * ys, wd * ws, 0, 0, 0, -xd * xs, -xd * ys, -xd * ws ] )

    h, A, y = np.zeros( (9, 1) ), np.array( A ), np.zeros( (9, 1) )

    if np.linalg.matrix_rank( A ) < 8:
        print( '\nRank of A less than 8.' )
        return np.reshape( h, (3,3) )
        # If A is less than rank 8 then return h as all zeros.
        # Since h will have 8 elements (h33 = 1), so rank of A should be equal to 8.

    # Calculate h.
    U, D, Vt = np.linalg.svd( A )

    VtInv = np.linalg.inv( Vt )     # The output if svd is V transpose itself. So
    # we only need to do the inverse to get the V inverse transpose.
    y[8, 0] = 1

    h = np.matmul( VtInv, y )

    return h

#================================================================================

def findHomographyByRANSAC( totalListOfCorresp=None, n=None, \
                            delta=None, N=None, M=None ):
    '''
    This function finds the homography by RANSAC. This function takes in the
    list of correspondences where the point in the source image (domain) and
    destination image (range) are arranged as a pair in a tuple. So one element
    of the totalListOfCorresp list looks like ( [srcX, srcY], [dstX, dstY] )
    '''
    if totalListOfCorresp is None or n is None or delta is None or N is None or M is None:
        print( '\nERROR. totalListOfCorresp or n or delta or N or M not provided. ' \
               'Aborting.\n' )
        return

    if n > 10:
        n = 10
        print( f'n too large. using n = {n} for the RANSAC.' )

    print( f'RANSAC parameters: delta = {delta}, N = {N}, M = {M}' )

    nTotalCorresp = len( totalListOfCorresp )

    # Separating the source and destination points from the totalListOfCorresp list
    # and also converting the points into homogeneous coordinates.
    srcPts = [ [ pair[0][0], pair[0][1], 1 ] for pair in totalListOfCorresp ]
    dstPts = [ [ pair[1][0], pair[1][1], 1 ] for pair in totalListOfCorresp ]
    srcPts, dstPts = np.array( srcPts ), np.array( dstPts )

#--------------------------------------------------------------------------------

    # Starting RANSAC trials.

    nInliers = 0
    bestH = np.zeros( (3,3) )
    hFound = False      # Flag to indicate that a best H is found.
    nMaxInliersFound = 0    # Counts the max no. of inliers found yet.

    # It may happen that for a too noisy image or the random combination of source
    # points used to calculate the homography is not giving a good result, and so
    # some more trials are needed.
    # So, if a best H is not found in N trials, then N is increased to some extent
```

```
    # to do some more trials to find H. But in this way as well, N is not
    # increased indefinitely. There is an upper limit to this increment as well.
    upLimitN = 100*N

    for i in range( N ):
        # Select n points randomly.
        idx = np.zeros( n )
        while len( list( set( idx.tolist() ) ) ) < n:
            idx = np.random.randint( 0, nTotalCorresp, n )
            # i.e. if not all the points in the randomly chosen set are unique,
            # then choose again.

        domainPts, rangePts = srcPts[ idx, : ], dstPts[ idx, : ]

        # Find homography.
        h = findHomographyByLLSM( domainPts, rangePts )

        H = np.reshape( h, (3,3) )      # Reshaping and making the last element 1.
        H = H / H[2, 2]

        #print(H)

        # Map all the domain points to the range with this homography.
        mappedPts = np.transpose( np.matmul( H, srcPts.T ) )

        # Convert mapped pts to planar form.
        for j in range( mappedPts.shape[0] ):     mappedPts[j] /= mappedPts[j][2]

        # Calculate distance between the true range points and these mapped points.
        distBetwRangeAndMappPts = np.linalg.norm( dstPts - mappedPts, axis=1 )

        # Find the number of inliers by checking how many of the points fall
        # within the distance threshold.
        inlierIndex = np.where( distBetwRangeAndMappPts < delta )

        inlierSrcPts, inlierDstPts = srcPts[ inlierIndex ], dstPts[ inlierIndex ]

        # Counting the number of inliers to find the best homography matrix.
        if len( inlierSrcPts ) >= M and len( inlierSrcPts ) >= nMaxInliersFound:
            hFound = True
            nInliers = len( inlierSrcPts )
            bestH = H
            inlierSrcSet = inlierSrcPts     # Set of all inliers in source group.
            inlierDstSet = inlierDstPts     # Set of all inliers in destination group.
            nMaxInliersFound = len( inlierSrcPts )
            #break

        if i == N-1 and not hFound and N < upLimitN:
            N += N
            print( f'\nRepeating. (i: {i}, N: {N})\n' )

        #print( f'{i+1}/{N}, nInliers: {len( inlierSrcPts )}, M: {M}' )

#-------------------------------------------------------------------------------

    if nInliers > M:  # This is the case when the h from LLSM is all 0s.
        # Refine the bestH with all the inlier set.
        h = findHomographyByLLSM( inlierSrcSet, inlierDstSet )
    else:
        h, inlierSrcSet, inlierDstSet = np.zeros( (3,3) ), None, None

    return h, inlierSrcSet, inlierDstSet

#===============================================================================
```

```
def refinedHomographyByLM( initialGuess=None, inlierSrcSet=None, inlierDstSet=None ):
    '''
    This function takes in the initial guess for the homography, the set of source
    and destination inliers and calculate a refined version of the homography using
    the Levenberg Marquardt (LM) algorithm. The h should be in the form of a 9x1 vector
    and not a 3x3 vector.
    '''
    h = initialGuess
    h = np.reshape( h, (9) )
    h /= h[8]       # Dividing h by the last element of h (h33).

    nPts = inlierSrcSet.shape[0]

#-------------------------------------------------------------------------------

    # Defining a function to find the h. Because to optimize using LM method, the
    # operation has to be defined as a function. This function calculates the
    # destination points with the h guessed by the LM optimizer and checks the
    # error with the actual given destination points.
    # The input and output of this function however has to be only a vector of
    # one dimension. Cannot be a matrix or array.
    # This is a requirement for the optimizer itself. And so the function defined
    # below takes in the h in a vector form, reshapes it internally, calculates
    # the error and finally reshapes the error back to a vector form to return.
    def function( hVec ):
        H = np.reshape( hVec, (3,3) )
        H = H / H[2, 2]
        mappedSet = np.transpose( np.matmul( H, inlierSrcSet.T ) )

        # This mappedSet does not have the 3rd coordinate of the points as 1.
        # So it has to be made 1 by dividing the points by the 3rd elements.
        # Since the inlierDstSet has the points which has the 3rd element as 1,
        # so if the mappedSet are also not in that form, then the error between
        # the inlierDstSet and the mappedSet will be too high. Hence the LM will
        # not give proper results.
        for j in range( mappedSet.shape[0] ):     mappedSet[j] /= mappedSet[j][2]
        error = inlierDstSet - mappedSet
        error = np.linalg.norm( error, axis=1 )
        error = np.reshape( error, (nPts) )
        return error

#-------------------------------------------------------------------------------

    # Optimized output.
    hNew = optimize.least_squares( function, h, method='lm' )

    # The optimized vector is obtained as hNex.x.
    return np.reshape( hNew.x, (9, 1) )   # Reshape to 9x1 and return.

#===============================================================================

def modifyImg5( sourceImg=None, H=None, canvas=None, canvasMinX=None, canvasMinY=None ):
    '''
    Basic operation of this function is the following:
    This function takes in a source image, and a homography of the target to source.
    Then it applies the inverse of this homography to the source (which is a
    deformed image) and finds the location where the corner points will be mapped
    when the H is applied to the source image. Then creates a target image with
    these dimensions. Now it takes all the locations of this blank target image
    one by one and applies the homography to them to find the location of the
    corresponding points in the source image. Then takes the pixel values of these
    points from the source image and replace the points in the target image.
    It also shifts the locations of the points mapped to the negative coordinates.

    But a modification to this is that, this can take in a canvas image as well.
```

```
    This image may finally contain a stitched version of several modified images.
    So, since the points needs to be mapped to the same canvas, it shifts the
    images by the canvasMinX and canvasMinY values instead of their own tgtMinX
    and tgtMinY values.
    '''

    # Drawing a black polygon to make all the pixels in the target region = 0,
    # before mapping.

    tgtW, tgtH, tgtMaxX, tgtMaxY, tgtMinX, tgtMinY = \
            dimsOfModifiedImg( sourceImg, H=np.linalg.inv( H ) )

    #print( tgtW, tgtH, tgtMaxX, tgtMaxY, tgtMinX, tgtMinY )

    sourceH, sourceW, _ = sourceImg.shape

    canvasH, canvasW, _ = canvas.shape

    processingTime = time.time()

    # Mapping the points.
    for r in range( tgtMinY, tgtMaxY + 1 ):
        for c in range( tgtMinX, tgtMaxX + 1 ):

            targetPt = homogeneousFormat( [ c, r ] )
            sourcePt = planarFormatFloat( applyHomography( [H, targetPt] ) )

            # Coordinates of the canvas where the source pixels will be copied.
            canvasR, canvasC = r - canvasMinY, c - canvasMinX

            if sourcePt[0] < sourceW and sourcePt[1] < sourceH and \
               sourcePt[0] > 0 and sourcePt[1] > 0 and \
               canvasR < canvasH and canvasC < canvasW and canvasR > 0 and canvasC > 0:

                # Mapping the source point pixel to the target point pixel.
                # If some pixel is already having some values due to some previous
                # mapping, then the new value is averaged with the old one.
                # To find these pixels, it is checked if sum of all the channels
                # at that pixel location is 0 or not (if it is not a 0,0,0
                # color pixel, then it means something is mapped to it earlier).

                if np.sum( canvas[ canvasR ][ canvasC ] ) > 0:
                    # Before the averaging the type of these pixel values should be
                    # made float, othewise the default uint8 value will overflow
                    # and distort the colors of these pixels.
                    val1 = np.asarray( canvas[ canvasR ][ canvasC ], dtype=np.float )
                    val2 = np.asarray( sourceImg[ int( sourcePt[1] ) ][ \
                                                  int( sourcePt[0] ) ], dtype=np.float )
                    # Average and convert to int and store to canvas.
                    val = ( val1 + val2 ) / 2
                    canvas[ canvasR ][ canvasC ] = np.asarray( val, dtype=np.uint8 )

                else:       # All zero pixel.
                    canvas[ canvasR ][ canvasC ] = sourceImg[ \
                                    int( sourcePt[1] ) ][ int( sourcePt[0] ) ]  # Sum.


    print( 'Time taken: {}'.format( time.time() - processingTime ) )

    return canvas    # Returning target image after mapping target into it.

#================================================================================

if __name__ == '__main__':
```

```
    # TASK 1.1 and 1.2

    # Loading the images.

    filepath = './PicsSelf'
    subfolder = 'group1'
    listOfImgs = os.listdir( os.path.join( filepath, subfolder ) )
    #print(listOfImgs)
    filename1, filename2, filename3, filename4, \
                        filename5 = 'a.jpg', 'b.jpg', 'c.jpg', 'd.jpg', 'e.jpg'

    imgA = cv2.imread( os.path.join( filepath, subfolder, filename1 ) )
    imgB = cv2.imread( os.path.join( filepath, subfolder, filename2 ) )
    imgC = cv2.imread( os.path.join( filepath, subfolder, filename3 ) )
    imgD = cv2.imread( os.path.join( filepath, subfolder, filename4 ) )
    imgE = cv2.imread( os.path.join( filepath, subfolder, filename5 ) )

    imgH, imgW, _ = imgA.shape

    listOfH = []

#-------------------------------------------------------------------------------

    # Find SIFT interest points and matches between the every pair of sequential
    # images. (like (a,b), (b,c), (c,d) etc.)
    i = 0
    nImgs = len( listOfImgs )

    lim = nImgs - 1      # -1 is because two image are considered in every iteration.
    #lim = 1

    while i < lim:
        img1 = cv2.imread( os.path.join( filepath, subfolder, listOfImgs[i] ) )
        img2 = cv2.imread( os.path.join( filepath, subfolder, listOfImgs[i+1] ) )

        img1Name, img2Name = listOfImgs[i].split('.')[0], listOfImgs[i+1].split('.')[0]

        img1gray = cv2.cvtColor( img1, cv2.COLOR_BGR2GRAY )
        img2gray = cv2.cvtColor( img2, cv2.COLOR_BGR2GRAY )

        # Initiate sift detector and find keypoints.
        sift = cv2.xfeatures2d.SIFT_create()
        kp1, des1 = sift.detectAndCompute( img1gray, None )
        kp2, des2 = sift.detectAndCompute( img2gray, None )

        matchThresh = '10%'

        matchedPairs1To2, goodMatches1to2, distValue = distanceSift( kp1, des1, kp2, des2, \
                                                                matchThresh=matchThresh )

        print( f'\nBetween image {listOfImgs[i]} and image {listOfImgs[i+1]}:' )
        print( f'Good matches: {len(goodMatches1to2)} out of {len(matchedPairs1To2)} ' \
                f'total matches (with threshold: {matchThresh}).' )

#-------------------------------------------------------------------------------

        ## Save the images showing the matches.
        #img = np.hstack( ( img1, img2 ) )
        #for idx, kp in enumerate( goodMatches1to2 ):
            #pt1 = kp[0]
            #pt2 = [ kp[1][0] + imgW, kp[1][1] ]
            #cv2.line( img, tuple(pt1), tuple(pt2), (255,0,255), 1 )

            #cv2.circle( img, tuple(pt1), 2, (0,255,0), -1 )
            #cv2.circle( img, tuple(pt1), 3, (0,0,255), 1 )
```

```
            #cv2.circle( img, tuple(pt2), 2, (0,255,0), -1 )
            #cv2.circle( img, tuple(pt2), 3, (0,0,255), 1 )

        ##cv2.imshow( 'Matches', img )
        ##cv2.waitKey(0)
        #cv2.imwrite( f'sift_matches_{img1Name}_{img2Name}.jpg', img )

#-------------------------------------------------------------------------------

        # Initializing the parameters for RANSAC.

        # Number of randomly chosen correspondence in each trial of RANSAC.
        # Should be between 4 and 10 as specified in notes.
        nCorrespPerTrial = 8

        # Std of the noise induced displacement of the inlier from its true location.
        sigma = 2

        delta = 3*sigma   # Distance (in pixels) threshold to accept a point as inlier.

        epsilon = 0.3   # Prob of a correspondence to be false (outlier).
        prob = 0.9999     # Prob of one of the N trials of RANSAC is free of outliers.
        # So if this is higher, then it means we are demanding for a more refined
        # set of inliers to be detected. Hence the N will increase to achieve that.

        N = int( math.log( 1 - prob ) / \
                 math.log( 1 - math.pow( ( 1 - epsilon ), nCorrespPerTrial ) ) )
        # This is the number of trials.

        nTotalCorresp = len( goodMatches1to2 )

        M = int( ( 1 - epsilon ) * nTotalCorresp )    # Number of inliers in the data.

#-------------------------------------------------------------------------------

        # Run RANSAC.
        h, inlierSrcSet, inlierDstSet = findHomographyByRANSAC( goodMatches1to2, \
                                                                n=nCorrespPerTrial, \
                                                                delta=delta, N=N, M=M )

        # Showing the outliers and inliers in the images.
        img = np.hstack( ( img1, img2 ) )
        for idx, kp in enumerate( goodMatches1to2 ):
            pt1 = kp[0]
            pt2 = [ kp[1][0] + imgW, kp[1][1] ]
            cv2.circle( img, tuple(pt1), 4, (0,0,255), -1 )
            cv2.circle( img, tuple(pt2), 4, (0,0,255), -1 )

        for idx, kp in enumerate( inlierSrcSet ):
            pt1 = [ kp[0], kp[1] ]
            pt2 = [ inlierDstSet[idx][0] + imgW, inlierDstSet[idx][1] ]
            cv2.circle( img, tuple(pt1), 4, (0,255,0), -1 )
            cv2.circle( img, tuple(pt2), 4, (0,255,0), -1 )

        #cv2.imshow( 'Matches', img )
        #cv2.waitKey(0)
        cv2.imwrite( f'inliers_outliers_{img1Name}_{img2Name}.jpg', img )

#-------------------------------------------------------------------------------

        H = np.reshape( h, (3,3) )
        H = H / H[2, 2]

        print( 'H by ransac' )
```

```
        print( H )

#-------------------------------------------------------------------------------

        # Use Levenberg-Marquardt algorithm to find more precise version of H.
        if inlierSrcSet is None or inlierDstSet is None:
            print( '\nERROR. Considerably large inlier set not found.\n' )
        else:
            hNew = refinedHomographyByLM( h, inlierSrcSet, inlierDstSet )
            Hnew = np.reshape( hNew, (3,3) )
            Hnew = Hnew / Hnew[2, 2]
            print( 'H refined by LM' )
            print( Hnew )

        listOfH.append( Hnew )

        i += 1

#-------------------------------------------------------------------------------

    # Creating the mapped image.
    print( '\n' )

    hAtoC = np.linalg.inv( np.matmul( listOfH[1], listOfH[0] ) )
    hBtoC = np.linalg.inv( listOfH[1] )
    hDtoC = listOfH[2]
    hEtoC = np.matmul( listOfH[3], listOfH[2] )

#-------------------------------------------------------------------------------

    # Finding the dimensions of the modified images to create the canvas.
    _, _, AmodifiedMaxX, AmodifiedMaxY, AmodifiedMinX, AmodifiedMinY = \
                                    dimsOfModifiedImg( imgA, hAtoC )

    _, _, BmodifiedMaxX, BmodifiedMaxY, BmodifiedMinX, BmodifiedMinY = \
                                    dimsOfModifiedImg( imgB, hBtoC )

    CmodifiedMaxX, CmodifiedMaxY, CmodifiedMinX, CmodifiedMinY = imgW, imgH, 0, 0

    _, _, DmodifiedMaxX, DmodifiedMaxY, DmodifiedMinX, DmodifiedMinY = \
                                    dimsOfModifiedImg( imgD, hDtoC )

    _, _, EmodifiedMaxX, EmodifiedMaxY, EmodifiedMinX, EmodifiedMinY = \
                                    dimsOfModifiedImg( imgE, hEtoC )

    # Finding the min and max offsets for the overall canvas.
    canvasMinX = min( [ AmodifiedMinX, BmodifiedMinX, CmodifiedMinX, \
                        DmodifiedMinX, EmodifiedMinX ] )
    canvasMaxX = max( [ AmodifiedMaxX, BmodifiedMaxX, CmodifiedMaxX, \
                        DmodifiedMaxX, EmodifiedMaxX ] )
    canvasMinY = min( [ AmodifiedMinY, BmodifiedMinY, CmodifiedMinY, \
                        DmodifiedMinY, EmodifiedMinY ] )
    canvasMaxY = max( [ AmodifiedMaxY, BmodifiedMaxY, CmodifiedMaxY, \
                        DmodifiedMaxY, EmodifiedMaxY ] )

    canvasW, canvasH = canvasMaxX - canvasMinX + 1, canvasMaxY - canvasMinY + 1
    canvas = np.zeros( ( canvasH, canvasW, 3 ), dtype=np.uint8 )

#-------------------------------------------------------------------------------

    # Mapping the images to canvas.
    canvas1 = modifyImg5( imgA, hAtoC, canvas, canvasMinX, canvasMinY )
    canvas2 = modifyImg5( imgB, hBtoC, canvas1, canvasMinX, canvasMinY )
    canvas3 = copy.deepcopy( canvas2 )
    canvas3[ CmodifiedMinY - canvasMinY : CmodifiedMaxY - canvasMinY, \
```

```
            CmodifiedMinX - canvasMinX : CmodifiedMaxX - canvasMinX ] = \
                                                imgC[ 0 : imgH, 0 : imgW ]
    canvas4 = modifyImg5( imgD, hDtoC, canvas3, canvasMinX, canvasMinY )
    canvas5 = modifyImg5( imgE, hEtoC, canvas4, canvasMinX, canvasMinY )

    cv2.imshow( 'canvas', canvas5 )
    cv2.waitKey(0)
```