ECE 661: Computer Vision

Homework 9, Fall 2018

Arindam Bhanja Chowdhury
abhanjac@purdue.edu

1 Overview

The focus of this homework is to implement projective reconstruction of a scent using stereo images. A 3D reconstruction is called projective if it is related by a 4x4 homography to the real scene. This means that what we obtain from projective reconstruction might appear distorted when compared to the actual scene. In practive, depending on how righ the scene structure is and how much prior knowledge one has about the objects in the scene, one may be able to use additional constraints derived from the reconstrction to remove the projective, affine and similarity distortions. In this homework we however focus only on the creating a projective reconstruction of a scene. The two images of the scene are shown in the Fig 1 and 2. Fig 1 shows the left image and Fig 2 shows the right image. They may look similar, but if checked closely, then it will appear that they are slightly different views of the same object.

Figure 1: Image 1 (left image) for original scene.

Figure 2: Image 2 (right image) for original scene.

2 Stereo Reconstruction Method

Given two cameras looking at the same scene, there exists a 3x3 matrix F of rank 2 that captures the most fundamental relationship between the pixels x_{1p} and x_{2p} in the two cameras for the same scene point X. The equation of this fundamental matrix F is given as $x_{2p}^T F x_{1p} = 0$.

There are altogether 9 elements of F, and determinant of F is 0 as it is of rank 2, and only ratio matters in F. So F has 7 degrees of freedom. So to find the F at least 7 corresponding points are needed. So in this homework 8 matching points from the image 1 and 2 are handpicked and then the F is estimated using Linear Least Squares (LLS) method.

If the $x_{1p} = [x_1, y_1]^T$ then in homogeneous coordinates $x_{1h} = [x_1, y_1, 1]^T$ and $x_{2h} = [x_2, y_2, 1]^T$. So the equation of F can be expanded as:

$$[x_2, y_2, 1] \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = 0$$

Or,

$$x_2x_1f_{11} + x_2y_1f_{12} + x_2f_{13} + y_2x_1f_{21} + y_2y_1f_{22} + y_2f_{23} + x_1f_{31} + y_1f_{32} + f_{33} = 0$$

Or,

$$\begin{bmatrix} x_2x_1 & x_2y_1 & x_2 & y_2x_1 & y_2y_1 & y_2 & x_1 & y_1 & 1 \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{bmatrix} = 0 \tag{1}$$

Now if there are 8 points pairs selected (like $[a_1, b_1], [a_2, b_2]$, $[c_1, d_1], [c_2, d_2]$, $[i_1, j_1], [i_2, j_2]$, $[m_1, n_1], [m_2, n_2]$, $[p_1, q_1], [p_2, q_2]$, $[r_1, s_1], [r_2, s_2]$, $[u_1, v_1], [u_2, v_2]$), then for each of the points there will be one such row of the right matrix. So the overall matrix will look like the following:

$$\begin{bmatrix} x_2x_1 & x_2y_1 & x_2 & y_2x_1 & y_2y_1 & y_2 & x_1 & y_1 & 1 \\ a_2a_1 & a_2b_1 & a_2 & b_2a_1 & b_2b_1 & b_2 & a_1 & b_1 & 1 \\ c_2c_1 & c_2d_1 & c_2 & d_2c_1 & d_2d_1 & d_2 & c_1 & d_1 & 1 \\ i_2i_1 & i_2j_1 & i_2 & j_2i_1 & j_2j_1 & j_2 & i_1 & j_1 & 1 \\ m_2m_1 & m_2n_1 & m_2 & n_2m_1 & n_2n_1 & n_2 & m_1 & n_1 & 1 \\ p_2p_1 & p_2q_1 & p_2 & q_2p_1 & q_2q_1 & q_2 & p_1 & q_1 & 1 \\ r_2r_1 & r_2s_1 & r_2 & s_2r_1 & s_2s_1 & s_2 & r_1 & s_1 & 1 \\ u_2u_1 & u_2v_1 & u_2 & v_2u_1 & v_2v_1 & v_2 & u_1 & v_1 & 1 \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2}$$

The more correspondences we have in the two image, the more accurate the resulting F will be. But in this homework we have only considered 8 correspondences.

By finding the SVD of F and taking only the last column of V matrix, we can get an initial estimate of F (this is also shown in the Appendix section). This initial estimate is then refined using Levenbert-Marquardt (LM) algorithm.

3 Image Rectification

Once a refined F is obtained, we need to rectify the images so that their epipoles lie to infinity. This is done by finding the homography matrix H1 and H2, that will rectify the images 1 and 2 and make their epipoles go to infinity.

The first step is to determint the camera projection matrix P_1 and P_2 for the left and right images. Now, we will consider the cameras to be in canonical configuration and so P_1 and P_2 are given by

$$P_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

$$P_2 = [[e_2]_X F \mid e_2]$$

Where e_2 is the left null vector of F matrix such that $e_2^T F = 0$. And $[e_2]_X$ is the matrix representation of e_2 which is

$$[e_2]_X = \begin{bmatrix} 0 & -e_z & e_y \\ e_z & 0 & -e_x \\ -e_y & e_x & 0 \end{bmatrix}$$

The image rectification process is as follows: For finding H_2 and H_1:

- Compute T that send the image center into origin
- Compute R that rotate epipole to $[f, 0, 1]^T$
- Compute G that send $[f, 0, 1]^T$ to $[f, 0, 0]^T$
- Compute T_2 that reserve the original image center
- Set $H^T = T_2GRT$
- H_1 can be calculated in many different ways. We chose a method similar to H_2.

If c_x and c_y be the center of the image 2 and $e_2 = [e_{2x}, e_{2y}]^T$ be the left null vector (in planar coordinates), then we have

$$\alpha = \arctan \frac{-(c_y - e_{2y})}{(c_x - e_{2x})}$$

$$f = (e_{2x} - c_x)\cos\alpha - (e_{2y} - c_y)\sin\alpha$$

$$R = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$T = \begin{bmatrix} 1 & 0 & -c_x \\ 0 & 1 & -c_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{f} & 0 & 1 \end{bmatrix}$$

$$H_2 = G_{3\times3}R_{3\times3}T_{3\times3}$$

$$\begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = H_2 \begin{bmatrix} c_x \\ c_y \\ 1 \end{bmatrix}$$

$$(c_{xr}, c_{yr}) = \left(\frac{c_1}{c_3}, \frac{c_2}{c_3}\right)$$

$$T_1 = \begin{bmatrix} 1 & 0 & c_x - c_{xr} \\ 0 & 1 & c_y - c_{yr} \\ 0 & 0 & 1 \end{bmatrix}$$

$$H_2 = T_1H_2$$

After this H_1 and H_2 is used to rectify the images 1 and 2.

Keypoint is extracted from image 1 and image 2 using SIFT operator. The keypoints of image 1 are then mapped to the rectified image 1 using H_1. Then each of these rectified keypoints are taken and the corresponding row of the rectified image 2 is scanned. Each point of rectified image 2 scanned is mapped back to original image 2 using H_2^{-1}. If there is a keypoint at this location in the original image 2, then the euclidean distance between the descriptor of the keypoint of image 1 (that we started of with) and the descriptor of this keypoint in image 2 (just found) is calculated. In this way the entire row of the rectified image 2 is sanned and the distances from all the keypoints

found is calculated. Then keypoint of image 2 corresponding to the lowest distance is considered a best match. This keypoint of image 2 and the keypoint of image 1 are then used as a matched pair. In this manner all the matches for all the keypoints of image 1 is calculated.

These matched keypoint are now used to find the world points in 3D.

4 3D Mapping Process

Now given a pair of matching correspondences x_1 and x_2 from image 1 and image 2 using the corresponding camera matries P_1 and P_2 it can be written that the world point X is

$$x_1 = \begin{bmatrix} x_{1x} \\ x_{1y} \\ 1 \end{bmatrix} = P_1 X = \begin{bmatrix} P_{1row1}^T \\ P_{1row2}^T \\ P_{1row3}^T \end{bmatrix}_{3\times 4} X_{4\times 1}$$

$$x_2 = \begin{bmatrix} x_{2x} \\ x_{2y} \\ 1 \end{bmatrix} = P_2 X = \begin{bmatrix} P_{2row1}^T \\ P_{2row2}^T \\ P_{2row3}^T \end{bmatrix}_{3\times 4} X_{4\times 1}$$

These two set of 3 equations (each) can be together written as follows:

$$\begin{aligned} x_{1y}P_{1row3}X - P_{1row2}X &= 0 \\ x_{1x}P_{1row3}X - P_{1row1}X &= 0 \\ x_{1x}P_{1row2}X - x_{1y}P_{1row1}X &= 0 \\ x_{2y}P_{2row3}X - P_{2row2}X &= 0 \\ x_{2x}P_{2row3}X - P_{2row1}X &= 0 \\ x_{2x}P_{2row2}X - x_{2y}P_{2row1}X &= 0 \end{aligned}$$

Since they are not independent, so to make the set independent the 3rd and 6th equation are ignored and the resulting equations becomes the following:

$$\begin{aligned} x_{1y}P_{1row3}X - P_{1row2}X &= 0 \\ x_{1x}P_{1row3}X - P_{1row1}X &= 0 \\ x_{2y}P_{2row3}X - P_{2row2}X &= 0 \\ x_{2x}P_{2row3}X - P_{2row1}X &= 0 \end{aligned} \tag{3}$$

This is now combined into a matrix form to create the following equation:

$$A_{4\times 4}X_{4\times 1} = 0_{4\times 1}$$

Or,

$$\begin{bmatrix} x_{1x}P_{1row3} - P_{1row1} \\ x_{1y}P_{1row3} - P_{1row2} \\ x_{2x}P_{2row3} - P_{2row1} \\ x_{2y}P_{2row3} - P_{2row2} \end{bmatrix} X = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{4}$$

Solving this by doing SVD of A (as shown in the appendix) gives us the solution for the world point X, which is shown in the scatter plot of Fig 13.

5 Procedure

- Selected 8 corresponding pairs of points from two images and used them to find an initial guess of the fundamental matrix (F). Then refined this F using Levenberg-Marquadt (LM) algorithm.
- Rectified the images by finding the H1 and H2 homography matrix using the F matrix. The epipoles now will theoritically be at infinity.
- Interest points are detected from the original images using SIFT algorithm and then they are projected onto the rectified image. Since these are on the rectified image, so the matches should be found in the same row of the other image. So a neighborhood (+/- 5 rows to the top and bottom of the same row) of the same row is searched and the matches are obtained.
- These matched keypoints are then used to create the world points and those are plotted in the scatter plot. The original 8 points which we started of with are also plotted in the scatter plot and are joined by lines so that they give an idea of the boundary of the object.

6 Results

The H1 and H2 and F matrix obtained are shown below.

$$H1 = \begin{bmatrix} 5.57023659 & -1.04349705 & -234.040351 \\ 1.95682232 & 2.87276982 & -231.669839 \\ 0.01877473 & -0.003517153 & 1.0 \end{bmatrix}$$

$$H2 = \begin{bmatrix} 5.37587394 & -0.696203978 & -242.726381 \\ 1.68292125 & 2.92589204 & -204.770290 \\ 0.0175684 & -0.00227521 & 1.0 \end{bmatrix}$$

$$F = \begin{bmatrix} -0.000157682 & -0.003411344 & 0.34839512 \\ 0.003396105 & -0.000356158 & 0.15184797 \\ -0.33170336 & -0.117888499 & 1.0 \end{bmatrix}$$

Number of keypoints detected using Sift in Fig 1: 107

Number of keypoints detected using Sift in Fig 2: 100

Number of matches found among the keypoints using rectified Fig 1 and Fig 2: 78

Number of good matches out of them: 19

Figure 3: 8 handpicked points of Image 1 from Fig 1.

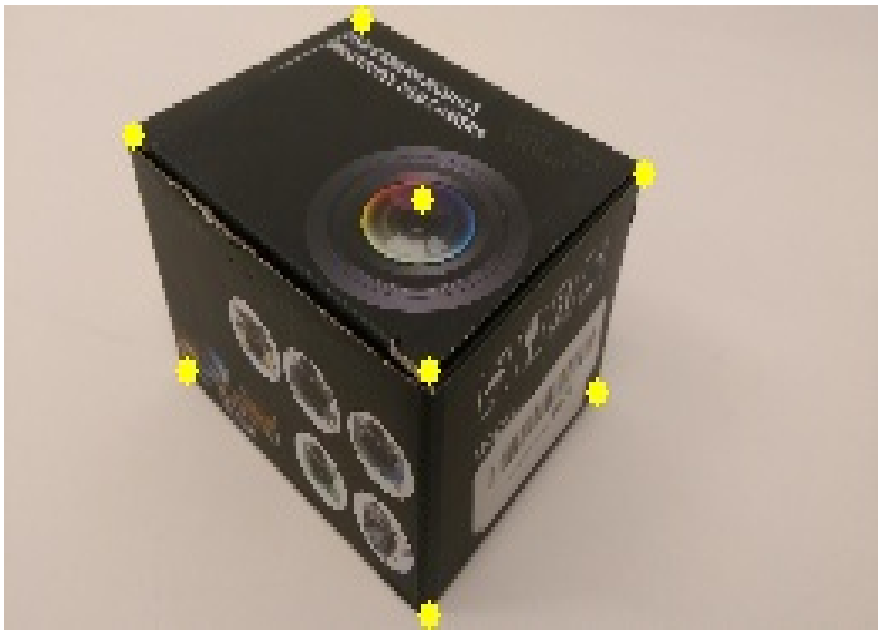

Figure 4: 8 handpicked points of Image 2 from Fig 2.

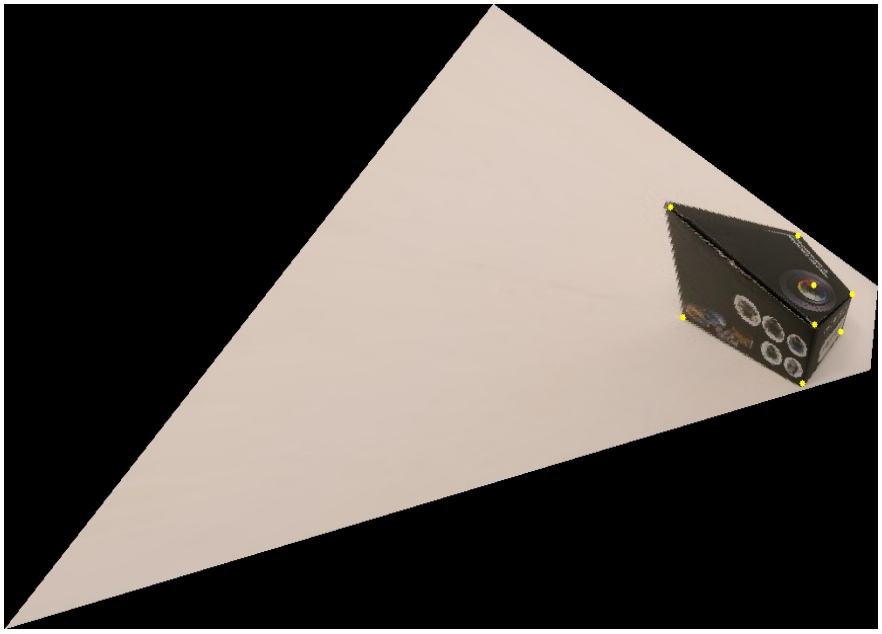

Figure 5: Rectified Image 1 with the handselected points.

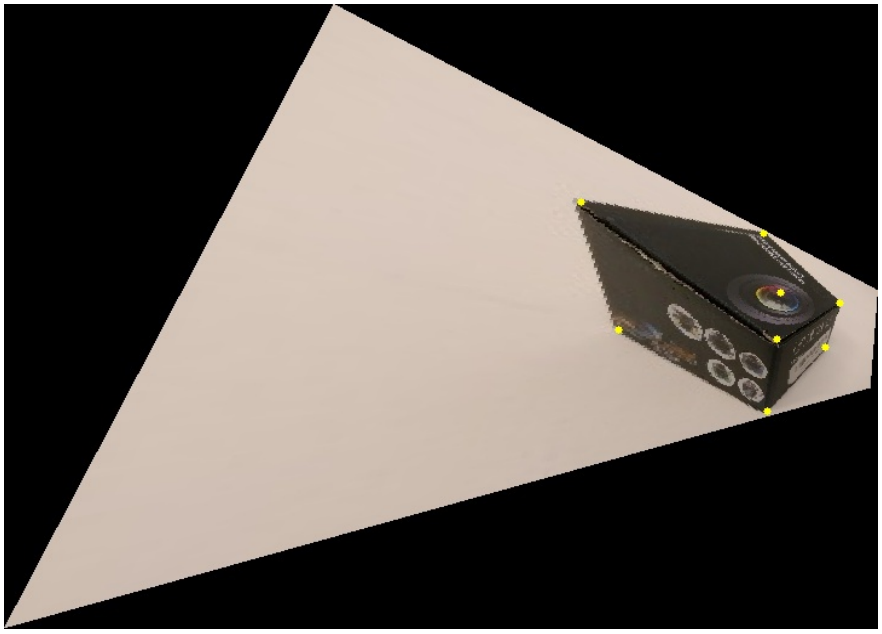

Figure 6: Rectified Image 2 with the handselected points.

Figure 7: SIFT Keypoints detected in Image 1.

Figure 8: SIFT Keypoints detected in Image 2.

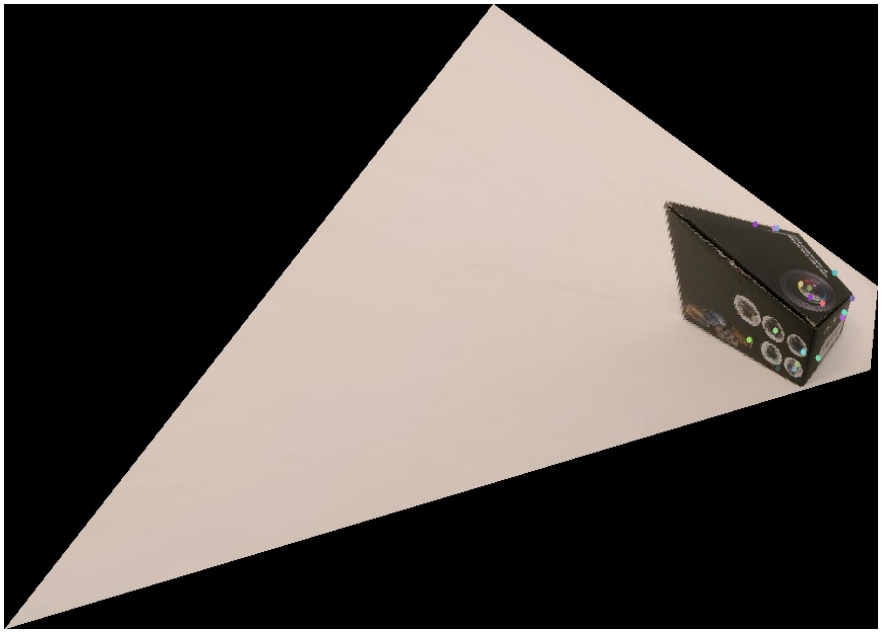

Figure 9: SIFT keypoints projected on the rectified Image 1.

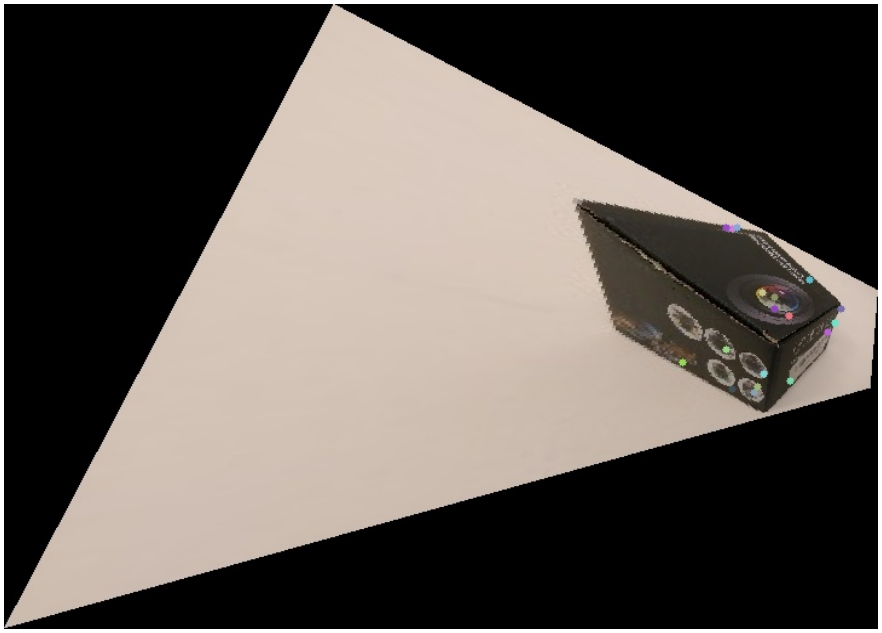

Figure 10: SIFT keypoints projected on the rectified Image 2.

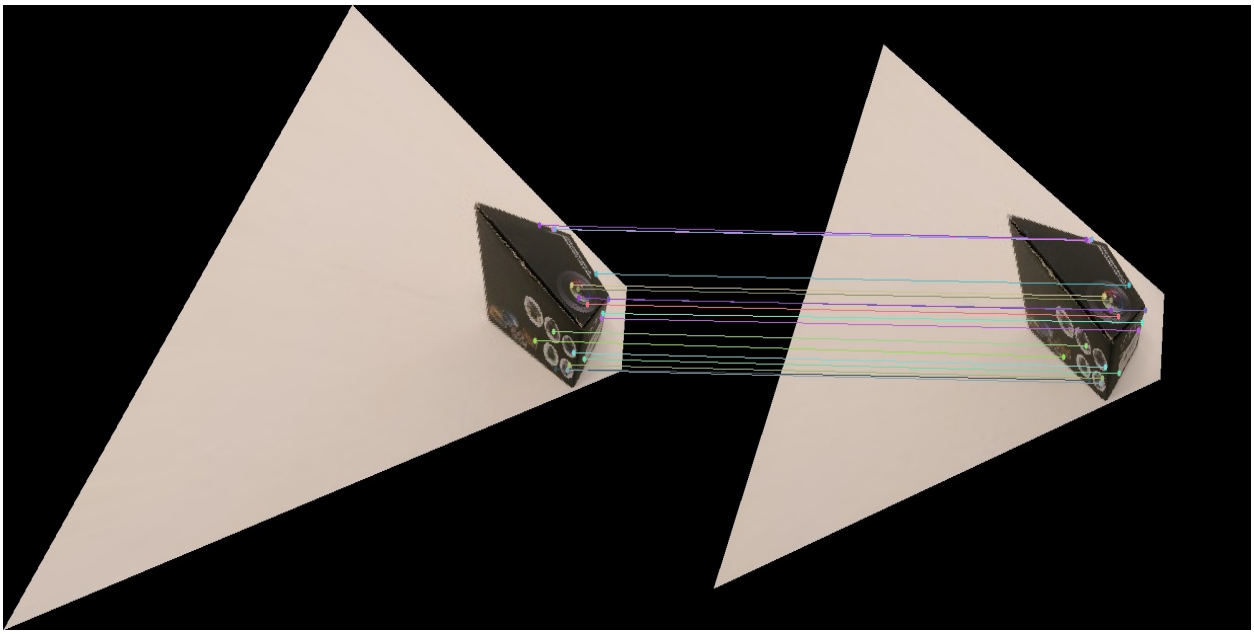

Figure 11: Matches of keypoints of Image 1 found by scanning the rows of Image 2. Line is drawn joining the good matches.

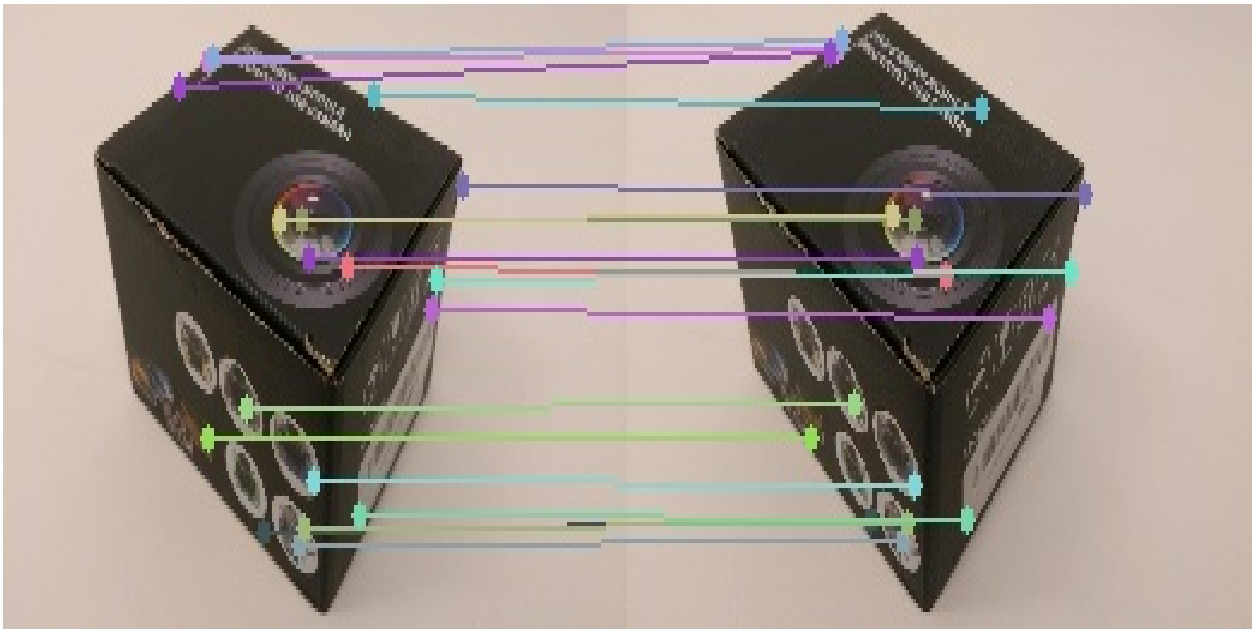

Figure 12: Good matches found from the rectified images are projected back to the original Image 1 and 2.

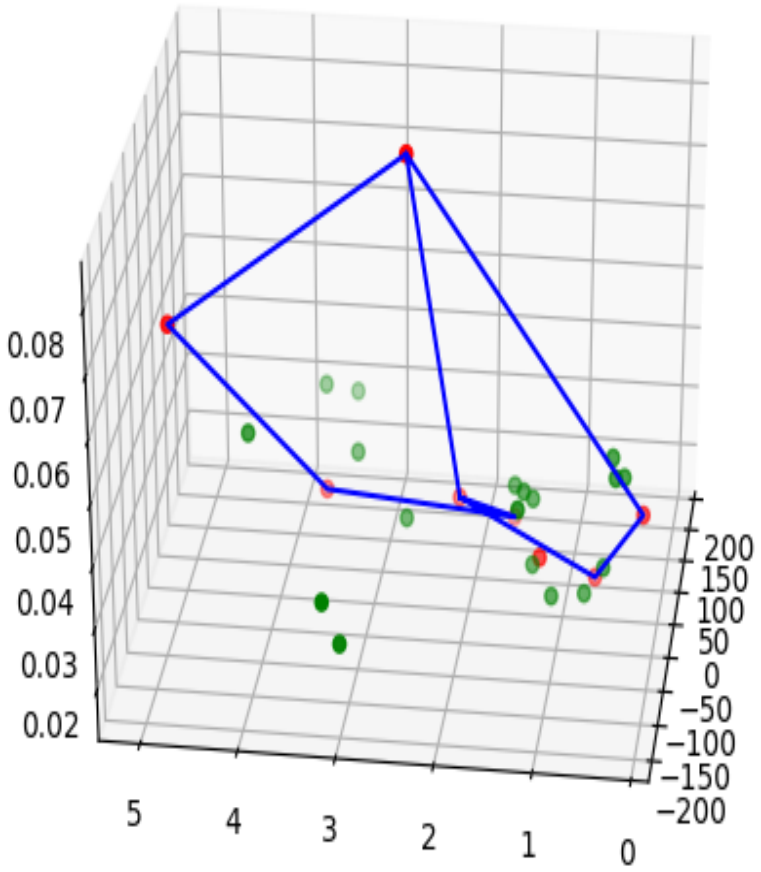

Figure 13: 3D reconstruction of the world points from the keypoint correspondences. The lines show the outline of the object with projective distortion.

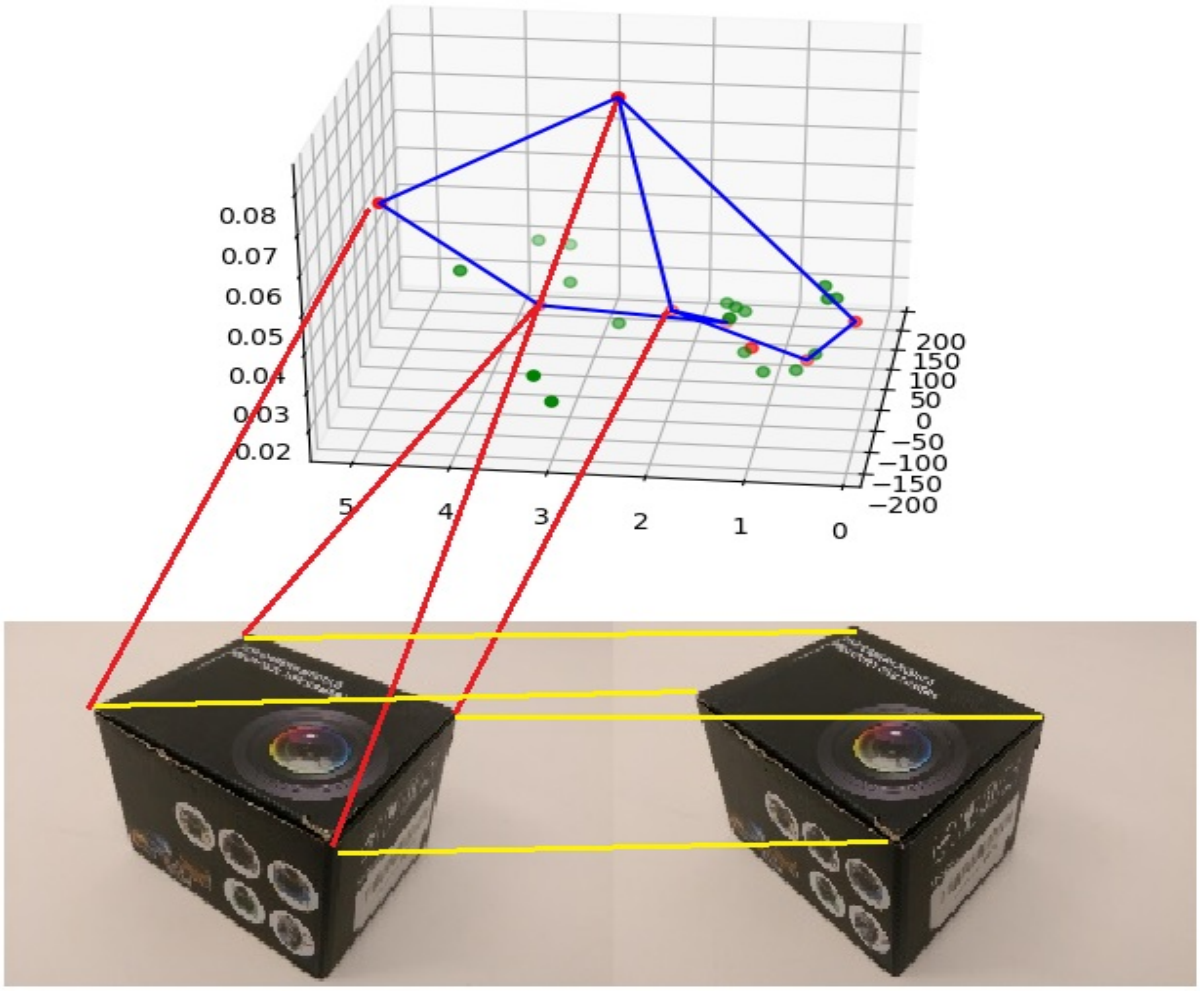

Figure 14: 3D reconstruction with guidelines from actual images.

7 Appendix

$$A_{8\times 9}h_{9\times 1} = [0]_{8\times 1}$$

To find the solution of h, the linear least squares (LLS) method should be used. For this the condition will be to minimize the term Ah, subjected to some constraints on the magnitude of h, otherwise the LLS will return a h = 0 trivial solution which is not useful at all. So we apply the constraint $||h|| = 1$ to prevent having the trivial solution.

Now, to solve this Singular Value Decomposition (SVD) is used.

By SVD, $A_{m\times n} = U_{m\times m}D_{m\times n}V^T_{n\times n}$ where U and V are orthonormal matrices, (so $UU^T = U^TU = I$ and $VV^T = V^TV = I$) and D is a diagonal matrix having the singular values along the diagonal in decreasing order of their magnitudes.

Now, since U and V are orthonormal, so it will also preserve norms, i.e. $||Ux|| = ||x||$ and same is true for V as well.

Also, let $V^Th = y$ or $h = V^{-T}y = Vy$, since $V^{-1} = V^T$ (for orthonormality).

So, $||h||^2 = h^Th = y^TV^TVy = y^T(V^TV)y = y^TIy = y^Ty = ||y||^2$.

So, it can be written that $||h|| = ||y||$.

These results can be used to derive the following,

$||Ah|| = ||UDV^Th|| = ||DV^Th|| = ||DV^Th|| = ||Dy||$.

So, mimimizing Ah with the constraint of $||h|| = 1$ implies mimimizing Dy with the constraint of $||y|| = 1$.

Finding this is trivial, as this corresponds to the smallest singular value in D. So the solution to the above equation is $y_{n\times 1} = [0, 0, 0, 0, 0, ...0, 0, 1]^T_{n\times 1}$. So, since $h = Vy$ (derived earlier), h is equal to the last column vector of V matrix.

This gives the required solution for h.

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

import numpy as np, cv2, os, time, math, copy, matplotlib.pyplot as plt
from scipy import signal, optimize
from mpl_toolkits.mplot3d import Axes3D

#================================================================================
# ARINDAM BHANJA CHOWDHURY
# abhanjac@purdue.edu
# ECE 661 FALL 2018, HW 9
#================================================================================

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

# Global variables that will mark the points in the image by mouse click.
ix, iy = -1, -1

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

def markPoints( event, x, y, flags, params ):
    '''
    This is a function that is called on mouse callback.
    '''
    global ix, iy
    if event == cv2.EVENT_LBUTTONDOWN:
        ix, iy = x, y

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

def selectPts( filePath=None ):
    '''
    This function opens the image and lets user select the points in it.
    These points are returned as a list.
    If the image is bigger than 640 x 480, it is displayed as 640 x 480. But
    the points are mapped and stored as per the original dimension of the image.
    The points are clicked by mouse on the image itself and they are stored in
    the listOfPts.
    '''

    global ix, iy

    img = cv2.imread( filePath )
    h, w = img.shape[0], img.shape[1]

    w1, h1, wRatio, hRatio, resized = w, h, 1, 1, False
    print( 'Image size: {}x{}'.format(w, h) )

    if w > 640:
        w1, resized = 640, True
        wRatio = w / w1
    if h > 480:
        h1, resized = 480, True
        hRatio = h / h1

    if resized:     img = cv2.resize( img, (w1, h1), \
                                      interpolation=cv2.INTER_AREA )

    cv2.namedWindow( 'Image' )
    cv2.setMouseCallback( 'Image', markPoints )  # Function to detect mouseclick
    key = ord('`')

#--------------------------------------------------------------------------------
```

```
    listOfPts = []      # List to collect the selected points.

    while key & 0xFF != 27:         # Press esc to break.

        imgTemp = np.array( img )      # Temporary image.

        # Displaying all the points in listOfPts on the image.
        for i in range( len(listOfPts) ):
            cv2.circle( imgTemp, tuple(listOfPts[i]), 3, (0, 255, 0), -1 )

        # After clicking on the image, press any key (other than esc) to display
        # the point on the image.

        if ix > 0 and iy > 0:
            print( 'New point: ({}, {}). Press \'s\' to save.'.format(ix, iy) )
            cv2.circle( imgTemp, (ix, iy), 3, (0, 0, 255), -1 )
            # Since this point is not saved yet, so it is displayed on the
            # temporary image and not on actual img1.

        cv2.imshow( 'Image', imgTemp )
        key = cv2.waitKey(0)

        # If 's' is pressed then the point is saved to the listOfPts.
        if key == ord('s'):
            listOfPts.append( [ix, iy] )
            cv2.circle( imgTemp, (ix, iy), 3, (0, 255, 0), -1 )
            img1 = imgTemp
            ix, iy = -1, -1
            print( 'Point Saved.' )

        elif key == ord('d'):   ix, iy = -1, -1  # Delete point by pressing 'd'.

    # Map the selected points back to the size of original image using the
    # wRatio and hRatio (if they we resized earlier).
    if resized:   listOfPts = [ [ int( p[0] * wRatio ), int( p[1] * hRatio ) ] \
                                                    for p in listOfPts ]

    return listOfPts

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

# 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) ) )

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

```
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 ]

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

def modifyImg3( sourceImg=None, H=None, listOfPts=None ):
    '''
    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. Thencreates 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.
    '''

    # 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 )

    targetImg = np.zeros( (tgtH, tgtW, 3), dtype=np.uint8 )

    sourceH, sourceW = sourceImg.shape[0], sourceImg.shape[1]

    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] ) )

            if sourcePt[0] < sourceW and sourcePt[1] < sourceH and \
               sourcePt[0] > 0 and sourcePt[1] > 0:

                # Mapping the source point pixel to the target point pixel.
                targetImg[ r - tgtMinY ][ c - tgtMinX ] = sourceImg[ \
                                int( sourcePt[1] ) ][ int( sourcePt[0] ) ]

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

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

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

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 = math.inf
        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 = math.inf
    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

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

def refinedFbyLM( initialGuess=None, listOfPts1=None, listOfPts2=None ):
    '''
    This function takes in the initial guess for the F matrix, the set of 8
    correspondences between the left (listOfPts1) and right (listOfPts2) images
    and calculates a refined version of F using the Levenberg Marquardt (LM)
    algorithm. F should be in the form of a 3x3 vector.
    '''
    F = initialGuess
    f = np.reshape( F, (9) )
    f /= f[8]       # Dividing f by the last element of f (h33).

    listOfPts1 = np.array( listOfPts1 )
    listOfPts2 = np.array( listOfPts2 )
    nPts = listOfPts1.shape[0]

    listOfPts1h = np.hstack( ( listOfPts1, np.ones( (nPts, 1) ) ) )
    listOfPts2h = np.hstack( ( listOfPts2, np.ones( (nPts, 1) ) ) )
```

```
#--------------------------------------------------------------------------------

    # Defining a function to find the f. Because to optimize using LM method, the
    # operation has to be defined as a function. This function calculates the
    # correspondences with the f guessed by the LM optimizer and checks the
    # error with the actual given correspondences.
    # 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 f in a vector form, reshapes it internally, calculates
    # the error and finally reshapes the error back to a vector form to return.

    # LM needs the number of residuals (i.e. the size of error vector) to be
    # more or equal to the size of the variables (params) which are optimized.
    # Since we have only 8 points here and 9 parameters in f, so it will be a problem.
    # Hence only the firsh 8 parameters of f are sent into the function (the last
    # parameter is 1 anyway and hence does not need any optimization).
    # Inside the function the 1 is added again.

    def function( fVec ):
        fVec = np.hstack( ( fVec, 1 ) )     # Appending 1 to the fVec.

        F = np.reshape( fVec, (3,3) )
        F = F / F[2,2]

        P1 = np.hstack( ( np.eye(3), np.zeros( (3,1) ) ) )

        # Equation is e2.T*F = 0.T which is equivalent to F.T*e2 = 0. Solving by svd.
        U, D, VT = np.linalg.svd( F.T )
        e2 = np.reshape( VT[-1], (3,1) )
        e2 = e2 / e2[2]
        #print(np.matmul( e2.T, F ))

        # Converting w to skew symmetrix form.
        e2matrix = np.array( [ [ 0, -e2[2], e2[1] ], \
                               [ e2[2], 0, -e2[0] ], \
                               [ -e2[1], e2[0], 0 ] ] )
        P2 = np.hstack( ( np.matmul( e2matrix, F ), e2 ) )

        p11, p12, p13 = P1[0, :], P1[1, :], P1[2, :]     # Rows of P1.
        p21, p22, p23 = P2[0, :], P2[1, :], P2[2, :]     # Rows of P2.

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

        listOfXh = []
        for j in range( nPts ):
            x1, y1 = listOfPts1[j,0], listOfPts1[j,1]
            x2, y2 = listOfPts2[j,0], listOfPts2[j,1]
            A = []
            A.append( ( x1 * p13 - p11 ).tolist() )
            A.append( ( y1 * p13 - p12 ).tolist() )
            A.append( ( x2 * p23 - p21 ).tolist() )
            A.append( ( y2 * p23 - p22 ).tolist() )

            A = np.array( A )
            U, D, VT = np.linalg.svd( A )
            X = VT[-1]

            X = X / X[-1]    # Dividing by last element to make that element 1.

            listOfXh.append( X.tolist() )

        listOfXh = np.array( listOfXh )
        #print( listOfXh )
```

```
#--------------------------------------------------------------------------------

        # Project X to right camera using P2, to get the estimate of listOfPts2h.
        # Project X to left camera using P1, to get the estimate of listOfPts1h.
        # Transpose is because listOfX1h is of shape 8x4 and we need 4x8 for the
        # matrix multiplication.
        listOfEstPts1h = np.matmul( P1, listOfXh.T )
        listOfEstPts2h = np.matmul( P2, listOfXh.T )

        listOfEstPts1h, listOfEstPts2h = listOfEstPts1h.T, listOfEstPts2h.T     # Now shape 8x3.

        # This listOfEstPts1h and listOfEstPts1h 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.
        for j in range( nPts ):     listOfEstPts1h[j] /= listOfEstPts1h[j][2]
        for j in range( nPts ):     listOfEstPts2h[j] /= listOfEstPts2h[j][2]

        error1 = np.linalg.norm( listOfPts1h - listOfEstPts1h, axis=1 )
        error2 = np.linalg.norm( listOfPts2h - listOfEstPts2h, axis=1 )

        error = error1 + error2

        return error

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

    #function(f[:-1])

    # Optimized output.
    fNew = optimize.least_squares( function, f[:-1], method='lm' )
    # The optimized vector is obtained as fNew.x.

    # Appending back the 1 to the fNew.
    fNew = np.hstack( ( fNew.x, 1 ) )
    #print( fNew )

    return fNew

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

if __name__ == '__main__':

    # TASK: 2.1 Image Rectification.

    # Loading the images of given dataset.

    # 1 is left image and 2 is right image for our picture.
    filepath = './images'
    file1, file2 = '3.jpg', '4.jpg'
    #file1, file2 = '1.jpg', '2.jpg'

    img1 = cv2.imread( os.path.join( filepath, file1 ) )
    img2 = cv2.imread( os.path.join( filepath, file2 ) )
    h, w, _ = img1.shape

    #listOfPts1 = selectPts( os.path.join( filepath, file1 ) )
    #print( f'Point selected from {file1}: {listOfPts1}' )
    listOfPts1 = [[40, 36], [104, 7], [191, 37], [137, 85], [54, 90], [133, 140], [178, 89], [129, 46]]
    #listOfPts1 = [[731, 486], [725,590], [308,301], [333,411], [971,143], [652,58], [838,220], [819,396],
[603,147], [340,276], [888,296], [619,274], [957,223]]

    #listOfPts2 = selectPts( os.path.join( filepath, file2 ) )
    #print( f'Point selected from {file2}: {listOfPts2}' )
    listOfPts2 = [[38, 30], [106, 3], [190, 39], [126, 85], [54, 85], [126, 142], [176, 90], [124, 45]]
    #listOfPts2=[[461, 558], [478, 648], [193, 220], [228, 327], [1017,207], [721,16], [790,269], [657,470],
```

```
    #listOfPts2=[[461, 558], [478, 648], [193, 220], [228, 327], [1017,207], [721,16], [790,269], [657,470],
[593,113], [239,201], [814,370], [510,272], [987,299]]

    nPts = len( listOfPts1 )

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

    img1 = cv2.imread( os.path.join( filepath, file1 ) )
    img2 = cv2.imread( os.path.join( filepath, file2 ) )

    for j in range(8):
        cv2.circle( img1, tuple(listOfPts1[j]), 3, (0,255,255), -1 )
        cv2.circle( img2, tuple(listOfPts2[j]), 3, (0,255,255), -1 )

    cv2.imshow( 'img1', img1 )
    cv2.imshow( 'img2', img2 )
    cv2.imwrite( f'handselected_points_{file1}', img1 )
    cv2.imwrite( f'handselected_points_{file2}', img2 )
    cv2.waitKey(0)

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

    img1 = cv2.imread( os.path.join( filepath, file1 ) )
    img2 = cv2.imread( os.path.join( filepath, file2 ) )

    # Finding the initial F matrix.
    A = []
    for i in range( nPts ):
        x1, y1 = listOfPts1[i][0], listOfPts1[i][1]
        x2, y2 = listOfPts2[i][0], listOfPts2[i][1]

        row = [ x2*x1, x2*y1, x2, y2*x1, y2*y1, y2, x1, y1, 1 ]
        A.append( row )

    A = np.array( A )
    U, D, VT = np.linalg.svd( A )
    f = VT[-1]
    F = np.reshape( f, (3,3) )

    # Incorporating the constraint that F is of rank 2, by making the smallest
    # singular value of F to 0 and then reconstructing F from its new singular
    # value decomposition matrices.
    Uf, Df, VTf = np.linalg.svd( F )    # F has rank of 3 now.
    #print(np.linalg.matrix_rank(F))

    Df1 = Df
    Df1[-1] = 0
    Df1 = np.diag( Df1 )
    F = np.matmul( Uf, Df1 )
    F = np.matmul( F, VTf )     # Now F has a rank of 2.
    F = F / F[2,2]      # Since in F only ratios matter.
    #print(np.linalg.matrix_rank(F), F)

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

    fNew = refinedFbyLM( F, listOfPts1, listOfPts2 )
    Fnew = np.reshape( fNew, (3,3) )
    Fnew = Fnew / Fnew[2,2]
    #print(Fnew)
    F = Fnew

    # Incorporating the constraint that F is of rank 2, by making the smallest
    # singular value of F to 0 and then reconstructing F from its new singular
    # value decomposition matrices.
    Uf, Df, VTf = np.linalg.svd( F )    # F has rank of 3 now.
```

```
    #print(np.linalg.matrix_rank(F))

    Df1 = Df
    Df1[-1] = 0
    Df1 = np.diag( Df1 )
    F = np.matmul( Uf, Df1 )
    F = np.matmul( F, VTf )     # Now F has a rank of 2.
    F = F / F[2,2]      # Since in F only ratios matter.
    #print(np.linalg.matrix_rank(F), F)

    print(f'F:\n{F}')

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

    # Rectifying the Images.
    # We rectify an image by sending the epipole to infinity along the x-axis.

    # Calculating e2.
    # Equation is e2.T*F = 0.T which is equivalent to F.T*e2 = 0. Solving by svd.
    U, D, VT = np.linalg.svd( F.T )
    e2 = np.reshape( VT[-1], (3,1) )
    e2 = e2 / e2[2]
    #print(np.matmul( e2.T, F ))    # Check e2.T*F = 0 or not (should be true for null vectors).

    # Finding the angle, f, R, G, T, T2 and finally H2.
    cx, cy = w/2, h/2
    e2x, e2y = e2[0] / e2[2], e2[1] / e2[2]

    # We will rotate the image so that it is parallel to the epipolar line.
    alpha = math.atan( -1*(cy - e2y) / (cx - e2x) )
    #alpha = math.atan2( -1*(cy - e2y), (cx - e2x) )

    cosAlpha, sinAlpha = np.cos( alpha ), np.sin( alpha )

    f = cosAlpha * ( e2x - cx ) - sinAlpha * ( e2y - cy )

    R = np.array( [ [ cosAlpha, -1 * sinAlpha, 0 ], \
                    [ sinAlpha, cosAlpha,      0 ], \
                    [ 0,        0,             1 ] ] )

    T = np.array( [ [ 1, 0, -cx ], [ 0, 1, -cy ], [ 0, 0, 1 ] ] )

    G = np.array( [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ -1/f, 0, 1 ] ] )

    RT = np.matmul( R, T )
    H2 = np.matmul( G, RT )

    # Keeping the center point in the image as constant.
    cPt = np.array( [ cx, cy, 1 ] )
    cPt = np.reshape( cPt, (3,1) )
    cPtNew = np.matmul( H2, cPt )
    cxNew, cyNew = cPtNew[0] / cPtNew[2], cPtNew[1] / cPtNew[2]

    T2 = np.array( [ [ 1, 0, cx-cxNew ], [ 0, 1, cy-cyNew ], [ 0, 0, 1 ] ] )

    H2 = np.matmul( T2, H2 )

    H2 = H2 / H2[2,2]

    print(f'H2:\n{H2}')

    # Create and save the rectified image.
    rectifiedImg2 = modifyImg3( img2, np.linalg.inv( H2 ), listOfPts2 )
    cv2.imshow( 'rectifiedImg2', rectifiedImg2 )
    cv2.waitKey(0)
```

```
    cv2.imwrite( f'rectified_{file2}', rectifiedImg2 )
    print( f'Saved rectified {file2}' )

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

    # Calculating e1.
    # Equation is F*e1 = 0. Solving by svd.
    U, D, VT = np.linalg.svd( F )
    e1 = np.reshape( VT[-1], (3,1) )
    e1 = e1 / e1[2]
    #print(np.matmul( F, e1 ))    # Check e2.T*F = 0 or not (should be true for null vectors).

    # Finding the angle, f, R, G, T, T1 and finally H1.
    cx, cy = w/2, h/2
    e1x, e1y = e1[0] / e1[2], e1[1] / e1[2]

    # We will rotate the image so that it is parallel to the epipolar line.
    alpha = np.arctan( -1 * ( cy - e1y ) / ( cx - e1x ) )
    cosAlpha, sinAlpha = np.cos( alpha )[0], np.sin( alpha )[0]

    f = cosAlpha * ( e1x - cx ) - sinAlpha * ( e1y - cy )

    R = np.array( [ [ cosAlpha, -1 * sinAlpha, 0 ], \
                    [ sinAlpha, cosAlpha,      0 ], \
                    [ 0,        0,             1 ] ] )

    T = np.array( [ [ 1, 0, -cx ], [ 0, 1, -cy ], [ 0, 0, 1 ] ] )

    G = np.array( [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ -1/f, 0, 1 ] ] )

    RT = np.matmul( R, T )
    H1 = np.matmul( G, RT )

    # Keeping the center point in the image as constant.
    cPt = [ cx, cy, 1 ]
    cPtNew = np.matmul( H1, cPt )
    cxNew, cyNew = cPtNew[0] / cPtNew[2], cPtNew[1] / cPtNew[2]

    T1 = np.array( [ [ 1, 0, cx-cxNew ], [ 0, 1, cy-cyNew ], [ 0, 0, 1 ] ] )

    H1 = np.matmul( T1, H1 )

    H1 = H1 / H1[2,2]

    print(f'H1:\n{H1}')

    # Create and save the rectified image.
    rectifiedImg1 = modifyImg3( img1, np.linalg.inv( H1 ), listOfPts1 )
    cv2.imshow( 'rectifiedImg1', rectifiedImg1 )
    cv2.waitKey(0)
    cv2.imwrite( f'rectified_{file1}', rectifiedImg1 )
    print( f'Saved rectified {file1}' )

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

    # Making the rectified files of the same size.

    # The rectified files may be of different sizes, as they are rectified by
    # different homographies. So making them of the same size by cropping the
    # bigger file.

    # First check if files already exists or not (as they can be already created
    # in an earlier run of the code).
    rectSizeMatchedFile1 = f'rectified_size_matched_{file1}'
    rectSizeMatchedFile2 = f'rectified_size_matched_{file2}'
```

```
    print( 'Matching the sizes of the rectified images.' )

    rectFile1 = cv2.imread( f'rectified_{file1}' )
    rectFile2 = cv2.imread( f'rectified_{file2}' )

    rectFile1H, rectFile1W, _ = rectFile1.shape
    rectFile2H, rectFile2W, _ = rectFile2.shape

    leftBorder1, topBorder1, leftBorder2, topBorder2 = 0, 0, 0, 0

    # Matching the widths by appending borders to the left and rightt.
    if rectFile1W > rectFile2W:
        difference = rectFile1W - rectFile2W
        leftBorderW = int( difference * 0.5 )
        leftBorder = np.zeros( (rectFile2H, leftBorderW, 3), dtype=np.uint8 )
        rightBorderW = difference - leftBorderW
        rightBorder = np.zeros( (rectFile2H, rightBorderW, 3), dtype=np.uint8 )
        rectFile2 = np.hstack( ( leftBorder, rectFile2, rightBorder ) )
        leftBorder2 = leftBorderW       # This offset may be used for plotting.

    elif rectFile2W > rectFile1W:
        difference = rectFile2W - rectFile1W
        leftBorderW = int( difference * 0.5 )
        leftBorder = np.zeros( (rectFile1H, leftBorderW, 3), dtype=np.uint8 )
        rightBorderW = difference - leftBorderW
        rightBorder = np.zeros( (rectFile1H, rightBorderW, 3), dtype=np.uint8 )
        rectFile1 = np.hstack( ( leftBorder, rectFile1, rightBorder ) )
        leftBorder1 = leftBorderW       # This offset may be used for plotting.

    rectFile1H, rectFile1W, _ = rectFile1.shape     # Updating the shape.
    rectFile2H, rectFile2W, _ = rectFile2.shape

    # Matching the widths by appending borders to the top and bottom.
    if rectFile1H > rectFile2H:
        difference = rectFile1H - rectFile2H
        topBorderH = int( difference * 0.5 )
        topBorder = np.zeros( (topBorderH, rectFile2W, 3), dtype=np.uint8 )
        botBorderH = difference - topBorderH
        botBorder = np.zeros( (botBorderH, rectFile2W, 3), dtype=np.uint8 )
        rectFile2 = np.vstack( ( topBorder, rectFile2, botBorder ) )
        topBorder2 = topBorderH       # This offset may be used for plotting.

    elif rectFile2H > rectFile1H:
        difference = rectFile2H - rectFile1H
        topBorderH = int( difference * 0.5 )
        topBorder = np.zeros( (topBorderH, rectFile1W, 3), dtype=np.uint8 )
        botBorderH = difference - topBorderH
        botBorder = np.zeros( (botBorderH, rectFile1W, 3), dtype=np.uint8 )
        rectFile1 = np.vstack( ( topBorder, rectFile1, botBorder ) )
        topBorder1 = topBorderH       # This offset may be used for plotting.

    # Saving the images again.
    cv2.imwrite( rectSizeMatchedFile1, rectFile1 )
    cv2.imwrite( rectSizeMatchedFile2, rectFile2 )

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

    # Plotting the original 8 points onto the rectified images.
    rectFile1 = cv2.imread( f'rectified_{file1}' )
    rectFile2 = cv2.imread( f'rectified_{file2}' )

    arrayOfPts1h = np.hstack( ( np.array( listOfPts1 ), np.ones( (nPts, 1) ) ) )
    arrayOfPts2h = np.hstack( ( np.array( listOfPts2 ), np.ones( (nPts, 1) ) ) )
```

```
    arrayOfPts1hMapped = np.matmul( H1, arrayOfPts1h.T ).T
    arrayOfPts2hMapped = np.matmul( H2, arrayOfPts2h.T ).T

    # This arrayOfPts1hMapped and arrayOfPts2hMapped 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.
    for j in range( nPts ):     arrayOfPts1hMapped[j] /= arrayOfPts1hMapped[j][2]
    for j in range( nPts ):     arrayOfPts2hMapped[j] /= arrayOfPts2hMapped[j][2]

    # After applying the homography the pixels in the image often gets mapped to
    # negative coordinates. Hence first where the corners of the image is mapped is
    # found out and those corner offsets are added to all the mapped pixel coordinates.
    # So finding these offsets. The same thing is also done inside modifyImg3 function.
    _, _, tgtMaxX1, tgtMaxY1, tgtMinX1, tgtMinY1 = dimsOfModifiedImg( img1, H1 )
    _, _, tgtMaxX2, tgtMaxY2, tgtMinX2, tgtMinY2 = dimsOfModifiedImg( img2, H2 )

    # Showing the original points in the rectified images.
    for j in range( nPts ):
        pt1 = [ int( arrayOfPts1hMapped[j][0] - tgtMinX1 ), \
                int( arrayOfPts1hMapped[j][1] - tgtMinY1 ) ]
        pt2 = [ int( arrayOfPts2hMapped[j][0] - tgtMinX2 ), \
                int( arrayOfPts2hMapped[j][1] - tgtMinY2 ) ]

        cv2.circle( rectFile1, tuple(pt1), 3, (0,255,255), -1 )
        cv2.circle( rectFile2, tuple(pt2), 3, (0,255,255), -1 )

    cv2.imshow( f'Original 8 points of {file1}', rectFile1 )
    cv2.imshow( f'Original 8 points of {file2}', rectFile2 )
    cv2.waitKey(0)
    cv2.imwrite( f'8_points_on_rectified_{file1}', rectFile1 )
    cv2.imwrite( f'8_points_on_rectified_{file2}', rectFile2 )

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

    # Showing that the correspondences in the rectified images lie on same straight lines.
    sizeMatchedImg1 = cv2.imread( f'rectified_size_matched_{file1}' )
    sizeMatchedImg2 = cv2.imread( f'rectified_size_matched_{file2}' )

    sizeMatchedImgH, sizeMatchedImgW, _ = sizeMatchedImg1.shape

    # The border offsets used to make the two images of same size, should also be
    # considered while plotting the points.
    img = np.hstack( ( sizeMatchedImg1, sizeMatchedImg2 ) )
    for j in range( nPts ):
        pt1 = [ int( arrayOfPts1hMapped[j][0] - tgtMinX1 + leftBorder1 ), \
                int( arrayOfPts1hMapped[j][1] - tgtMinY1 + topBorder1 ) ]
        pt2 = [ int( arrayOfPts2hMapped[j][0] - tgtMinX2 + sizeMatchedImgW + leftBorder2 ), \
                int( arrayOfPts2hMapped[j][1] - tgtMinY2 + topBorder2 ) ]

        cv2.line( img, tuple(pt1), tuple(pt2), (255,0,255), 1 )
        cv2.circle( img, tuple(pt1), 3, (0,255,0), -1 )
        cv2.circle( img, tuple(pt2), 3, (0,255,0), -1 )

    cv2.imshow( f'Correspondences.jpg', img )
    cv2.waitKey(0)
    cv2.imwrite( f'Correspondences_on_rectified_images_{file1[:-4]}_{file2[:-4]}.jpg', img )

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

    # TASK 2.2 Interest point detection.

    # Reading the files afresh.
    img1 = cv2.imread( os.path.join( filepath, file1 ) )
    img2 = cv2.imread( os.path.join( filepath, file2 ) )
```

```
    h, w, _ = img1.shape

    img = np.hstack( ( img1, img2 ) )

    rectFile1 = cv2.imread( f'rectified_{file1}' )
    rectFile2 = cv2.imread( f'rectified_{file2}' )

    rectFile1H, rectFile1W, _ = rectFile1.shape
    rectFile2H, rectFile2W, _ = rectFile2.shape

    # After applying the homography the pixels in the image often gets mapped to
    # negative coordinates. Hence first where the corners of the image is mapped is
    # found out and those corner offsets are added to all the mapped pixel coordinates.
    # So finding these offsets. The same thing is also done inside modifyImg3 function.
    _, _, tgtMaxX1, tgtMaxY1, tgtMinX1, tgtMinY1 = dimsOfModifiedImg( img1, H1 )
    _, _, tgtMaxX2, tgtMaxY2, tgtMinX2, tgtMinY2 = dimsOfModifiedImg( img2, H2 )


    # Showing that the correspondences in the rectified images lie on same straight lines.
    sizeMatchedImg1 = cv2.imread( f'rectified_size_matched_{file1}' )
    sizeMatchedImg2 = cv2.imread( f'rectified_size_matched_{file2}' )

    sizeMatchedImgH, sizeMatchedImgW, _ = sizeMatchedImg1.shape

    sizeMatchedImg = np.hstack( ( sizeMatchedImg1, sizeMatchedImg2 ) )

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

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

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

    nKp1, nKp2 = len( kp1 ), len( kp2 )

    print( f'Number of keypoints detected using Sift in {file1}: {nKp1}' )
    print( f'Number of keypoints detected using Sift in {file2}: {nKp2}' )

    # Only extracting the pixel coordinates from the keypoint class and storing in
    # the list.
    listOfKp1 = [ [ int( j.pt[0] ), int( j.pt[1] ) ] for j in kp1 ]
    listOfDes1 = des1.tolist()      # Converting the descriptor array into lists.
    listOfKp2 = [ [ int( j.pt[0] ), int( j.pt[1] ) ] for j in kp2 ]
    listOfDes2 = des2.tolist()      # Converting the descriptor array into lists.

    # Sorting the keypoints and the descriptors together as per the y coordinates
    # of the keypoints.
    sortedKp1, sortedDes1 = zip( *sorted( zip( listOfKp1, listOfDes1 ), key=lambda x: x[0][1] ) )
    sortedKp2, sortedDes2 = zip( *sorted( zip( listOfKp2, listOfDes2 ), key=lambda x: x[0][1] ) )

    # The sortedKp1 obtained in this manner are tuples of keypoints. Converting
    # them to lists for future purposes.
    sortedKp1, sortedKp2 = list( sortedKp1 ), list( sortedKp2 )
    sortedDes1, sortedDes2 = list( sortedDes1 ), list( sortedDes2 )

    # We will be removing the keypoints of image 2 and their descriptors from the
    # sortedKp2 and sortedDes2 lists, once a match is found with those of keypoints
    # of image 1. Hence keeping a copy of those just in case we need them.
    sortedKp2copy = copy.deepcopy( sortedKp2 )
    sortedDes2copy = copy.deepcopy( sortedDes2 )

#-------------------------------------------------------------------------------
```

```
    rectImgCombined = np.hstack( ( sizeMatchedImg1, sizeMatchedImg2 ) )
    imgCombined = np.hstack( ( img1, img2 ) )

    rowBuffer = 5

    H2inv = np.linalg.inv( H2 )     # This inverse will be used in the loop a lot.
    H2inv = H2inv / H2inv[2,2]

    matchedKptPairs1To2, matchedDesPair1to2, matchedIdxPair1to2, minDistList = [], [], [], []
    matchedRectKptPair1To2 = []

    for j in range( nKp1 ):
        # Scanning all the keypoints from the left image.
        # Mapping them onto the left rectified image.
        # Scanning a small neighborhood (+/- rowBuffer no. of rows) of the same
        # row on the right rectified image.
        # Mapping these points on the right rectified image onto the original right image.
        # Checking if any of these are among the keypoint set detected from the right
        # original image or not.
        # If so, then checking the euclidean distance between the descriptors of this
        # keypoint (in the right image) and the descriptor of the first keypoint
        # (detected in the left image that we started of with) for a potential match.
        kpt1 = sortedKp1[j]
        descriptor1 = sortedDes1[j]

        kpt1h = homogeneousFormat( kpt1 )
        rectKpt1h = applyHomography( [ H1, kpt1h ] )      # Mapped kpt1 to rectified img1.
        rectKpt1h /= rectKpt1h[-1]      # Making last element of homogeneous format 1.

        # The range of points searched will be from the minimum possible x index to
        # the max possible x index (of image2) that can be achieved by the homography.
        # The range of y index will be +/- the rowBuffer amount from the y index of
        # the rectified keypoint in image 1.
        yStart, yEnd = int( rectKpt1h[1][0] - rowBuffer ), int( rectKpt1h[1][0] + rowBuffer )
        xStart, xEnd = tgtMinX2, tgtMaxX2

        minDist = math.inf

        # matchFound == True indicates that matching keypoint in image 2 is found.
        matchFound = False

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

        for yh in range( yStart, yEnd, 1 ):
            for xh in range( xStart, xEnd, 1 ):
                # Taking each point from this range (in the rectified image 2) and
                # finding the corresponding point in the original unrectified image 2.
                rectKpt2h = homogeneousFormat( [ xh, yh, 1 ] )
                kpt2h = applyHomography( [ H2inv, rectKpt2h ] )
                kpt2 = planarFormat( kpt2h )

                # Check if this keypoint is at all there or not in sortedKp2.
                if kpt2 in sortedKp2:

                    idxKp2 = sortedKp2.index( kpt2 )
                    descriptor2 = sortedDes2[ idxKp2 ]

                    # Calculating the euclidean distance between the kpt1 and kpt2.
                    dist = ( np.array( descriptor1 ) - np.array( descriptor2 ) )**2
                    dist = np.sqrt( np.sum( dist ) )

                    if minDist > dist:
                        # Calculating the minimum euclidean distance value and storing
                        # the matched elements into individual lists for plotting.
                        minDist = dist
```

```
                    matchedKpt1, matchedKpt2 = kpt1, kpt2
                    matchedPtIdx1, matchedPtIdx2 = j, idxKp2
                    matchedDes1, matchedDes2 = descriptor1, descriptor2
                    matchedRectKpt1 = planarFormat( rectKpt1h )
                    matchedRectKpt2 = planarFormat( rectKpt2h )
                    matchFound = True       # Updating the flag indicating match found.

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

        if matchFound:
            # Saving the minimum distance, keypoints, descriptors and keypoint indices.
            matchedKptPairs1To2.append( [ matchedKpt1, matchedKpt2 ] )
            matchedDesPair1to2.append( [ matchedDes1, matchedDes2 ] )
            matchedIdxPair1to2.append( [ matchedPtIdx1, matchedPtIdx2 ] )
            minDistList.append( minDist )
            matchedRectKptPair1To2.append( [ matchedRectKpt1, matchedRectKpt2 ] )

            # Once a match is found, remove the corresponding keypoint and descriptor
            # from the lists of keypoints and descriptors of image 2.
            # This is so that another keypoint from image 1 does not get matched
            # to the same keypoint in image 2.
            sortedKp2.remove( matchedKpt2 )
            sortedDes2.remove( matchedDes2 )

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

    # Sorting and plotting the matches.

    # Sort the lists of matches as per the distance of the match in ascending order.
    matchedKptPairs1To2, matchedDesPair1to2, matchedIdxPair1to2, \
        minDistList, matchedRectKptPair1To2 = zip( *sorted( zip( matchedKptPairs1To2, \
            matchedDesPair1to2, matchedIdxPair1to2, minDistList, matchedRectKptPair1To2 ), \
                                                key=lambda x: x[3] ) )

    # The sorted objects obtained in this manner are tuples. Converting them to lists.
    matchedKptPairs1To2, matchedDesPair1to2, matchedIdxPair1to2, minDistList, \
            matchedRectKptPair1To2 = list(matchedKptPairs1To2), list(matchedDesPair1to2), \
                    list(matchedIdxPair1to2), list(minDistList), list(matchedRectKptPair1To2)

    # Not all the matches in this list are good matches. So the matches are first
    # arranged in ascending order of their min distances and only a subset of the
    # lowest min distance matches is taken as good matches (like 30% of the total number
    # of matched points).
    nMatches = len( matchedKptPairs1To2 )
    nGoodMatches = int( nMatches * 25 / 100 )

    print( f'Number of matches found among keypoints from {file1} and {file2}: {nMatches}' )
    print( f'Number of good matches: {nGoodMatches}' )

    # Colors to draw the points.
    color = np.random.randint( 60, 240, (nGoodMatches, 3) )
    color = color.tolist()

    for j in range( nGoodMatches ):
        # Drawing the matched points with the same colors on img1 and img2 and
        # also on the combined img (img1 and img2 stacked side by side) and joining
        # them with a line.
        ptColor = tuple( color[j] )

        # Mark points on the original images.
        matchedKpt1, matchedKpt2 = matchedKptPairs1To2[j][0], matchedKptPairs1To2[j][1]

        cv2.circle( img1, tuple(matchedKpt1), 3, ptColor, -1 )
        cv2.circle( img2, tuple(matchedKpt2), 3, ptColor, -1 )
```

```
        # Mark correspondences on the stacked original images.
        cv2.circle( img, tuple( matchedKpt1 ), 3, ptColor, -1 )
        cv2.circle( img, ( matchedKpt2[0] + w, matchedKpt2[1] ), 3, ptColor, -1 )
        cv2.line( img, tuple( matchedKpt1 ), ( matchedKpt2[0] + w, matchedKpt2[1] ), ptColor, 1 )

        # Mark points on the rectified images.
        matchedRectKpt1, matchedRectKpt2 = matchedRectKptPair1To2[j][0], matchedRectKptPair1To2[j][1]

        pt1 = [ int( matchedRectKpt1[0] - tgtMinX1 ), int( matchedRectKpt1[1] - tgtMinY1 ) ]
        pt2 = [ int( matchedRectKpt2[0] - tgtMinX2 ), int( matchedRectKpt2[1] - tgtMinY2 ) ]

        cv2.circle( rectFile1, tuple(pt1), 3, ptColor, -1 )
        cv2.circle( rectFile2, tuple(pt2), 3, ptColor, -1 )

        # Mark correspondences on the stacked rectified images.
        pt1 = [ int( matchedRectKpt1[0] - tgtMinX1 + leftBorder1 ), \
                int( matchedRectKpt1[1] - tgtMinY1 + topBorder1 ) ]
        pt2 = [ int( matchedRectKpt2[0] - tgtMinX2 + leftBorder2 + sizeMatchedImgW ), \
                int( matchedRectKpt2[1] - tgtMinY2 + topBorder2 ) ]

        cv2.line( sizeMatchedImg, tuple(pt1), tuple(pt2), ptColor, 1 )
        cv2.circle( sizeMatchedImg, tuple(pt1), 3, ptColor, -1 )
        cv2.circle( sizeMatchedImg, tuple(pt2), 3, ptColor, -1 )

    cv2.imshow( 'img1', img1 )
    cv2.imshow( 'img2', img2 )
    cv2.imshow( 'img', img )
    cv2.imshow( 'rectFile1', rectFile1 )
    cv2.imshow( 'rectFile2', rectFile2 )
    cv2.imshow( 'sizeMatchedImg', sizeMatchedImg )
    cv2.waitKey(0)
    cv2.imwrite( f'keypoints_{file1}', img1 )
    cv2.imwrite( f'keypoints_{file2}', img2 )
    cv2.imwrite( f'keypoints_matching_on_{file1[:-4]}_and_{file2}', img )
    cv2.imwrite( f'rectified_keypoints_{file1}', rectFile1 )
    cv2.imwrite( f'rectified_keypoints_{file2}', rectFile2 )
    cv2.imwrite( f'keypoints_matching_on_rectified_{file1[:-4]}_and_{file2}', sizeMatchedImg )

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

    # TASK 2.3 Projective reconstruction.

    # Calculating the projective matrices.
    P1 = np.hstack( ( np.eye(3), np.zeros( (3,1) ) ) )

    H2inv = np.linalg.inv( H2 )
    H1inv = np.linalg.inv( H1 )
    FH1inv = np.matmul( F, H1inv )
    F = np.matmul( H2inv.T, FH1inv )
    F = F / F[2,2]

    # Calculating e2.
    # Equation is e2.T*F = 0.T which is equivalent to F.T*e2 = 0. Solving by svd.
    U, D, VT = np.linalg.svd( F.T )
    e2 = np.reshape( VT[-1], (3,1) )
    e2 = e2 / e2[2]

    # Converting w to skew symmetrix form.
    e2matrix = np.array( [ [ 0, -e2[2], e2[1] ], \
                           [ e2[2], 0, -e2[0] ], \
                           [ -e2[1], e2[0], 0 ] ] )
    P2 = np.hstack( ( np.matmul( e2matrix, F ), e2 ) )

    p11, p12, p13 = P1[0, :], P1[1, :], P1[2, :]     # Rows of P1.
    p21, p22, p23 = P2[0, :], P2[1, :], P2[2, :]     # Rows of P2.
```

```
#-------------------------------------------------------------------------------

    listOfXh = []
    for j in range( nGoodMatches ):
        # Find the world points from the matching keypoints.
        x1, y1 = matchedKptPairs1To2[j][0][0], matchedKptPairs1To2[j][0][1]
        x2, y2 = matchedKptPairs1To2[j][1][0], matchedKptPairs1To2[j][1][1]

        # The matchedKptPairs1To2 is a list where every element is a sublist of
        # two matching keypoints. And every keypoint is also a 2-element list of
        # the x and y coordinates of that keypoint.
        # So matchedKptPairs1To2 is a list of list of list.

        A = []
        A.append( ( x1 * p13 - p11 ).tolist() )
        A.append( ( y1 * p13 - p12 ).tolist() )
        A.append( ( x2 * p23 - p21 ).tolist() )
        A.append( ( y2 * p23 - p22 ).tolist() )

        A = np.array( A )
        U, D, VT = np.linalg.svd( A )
        X = VT[-1]

        X = X / X[-1]    # Dividing by last element to make that element 1.

        # Only storing the x, y, z (as w==1 anyways).
        listOfXh.append( X[:-1].tolist() )

    listOfXh = np.array( listOfXh )
    #print( listOfXh )

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

    # Also converting the original hand picked 8 points into world points.
    listOfXhForOriginal8pts = []
    for j in range( nPts ):
        x1, y1 = listOfPts1[j][0], listOfPts1[j][1]
        x2, y2 = listOfPts2[j][0], listOfPts2[j][1]

        A = []
        A.append( ( x1 * p13 - p11 ).tolist() )
        A.append( ( y1 * p13 - p12 ).tolist() )
        A.append( ( x2 * p23 - p21 ).tolist() )
        A.append( ( y2 * p23 - p22 ).tolist() )

        A = np.array( A )
        U, D, VT = np.linalg.svd( A )
        X = VT[-1]

        X = X / X[-1]    # Dividing by last element to make that element 1.

        # Only storing the x, y, z (as w==1 anyways).
        listOfXhForOriginal8pts.append( X[:-1].tolist() )

    listOfXhForOriginal8pts = np.array( listOfXhForOriginal8pts )
    #print( listOfXhForOriginal8pts )

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

    # Creating a boundary around the 3d points corresponding to the handpicked 8 points.
    # Rearranging the points into a new list such that when the points of this list
    # are traversed in sequence, a boundary of a box will be created.
    x3DptsList = [  listOfXhForOriginal8pts[0][0],
                    listOfXhForOriginal8pts[1][0],
```

```
                    listOfXhForOriginal8pts[2][0],
                    listOfXhForOriginal8pts[3][0],
                    listOfXhForOriginal8pts[0][0],
                    listOfXhForOriginal8pts[4][0],
                    listOfXhForOriginal8pts[5][0],
                    listOfXhForOriginal8pts[6][0],
                    listOfXhForOriginal8pts[3][0],
                    listOfXhForOriginal8pts[2][0],
                    listOfXhForOriginal8pts[3][0],
                    listOfXhForOriginal8pts[5][0] ]

    y3DptsList = [  listOfXhForOriginal8pts[0][1],
                    listOfXhForOriginal8pts[1][1],
                    listOfXhForOriginal8pts[2][1],
                    listOfXhForOriginal8pts[3][1],
                    listOfXhForOriginal8pts[0][1],
                    listOfXhForOriginal8pts[4][1],
                    listOfXhForOriginal8pts[5][1],
                    listOfXhForOriginal8pts[6][1],
                    listOfXhForOriginal8pts[3][1],
                    listOfXhForOriginal8pts[2][1],
                    listOfXhForOriginal8pts[3][1],
                    listOfXhForOriginal8pts[5][1] ]

    z3DptsList = [  listOfXhForOriginal8pts[0][2],
                    listOfXhForOriginal8pts[1][2],
                    listOfXhForOriginal8pts[2][2],
                    listOfXhForOriginal8pts[3][2],
                    listOfXhForOriginal8pts[0][2],
                    listOfXhForOriginal8pts[4][2],
                    listOfXhForOriginal8pts[5][2],
                    listOfXhForOriginal8pts[6][2],
                    listOfXhForOriginal8pts[3][2],
                    listOfXhForOriginal8pts[2][2],
                    listOfXhForOriginal8pts[3][2],
                    listOfXhForOriginal8pts[5][2] ]

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

    fig1 = plt.figure(1)
    fig1.gca().cla()

    ax = fig1.add_subplot(1,1,1, projection='3d')
    # Have to include the package 'from mpl_toolkits.mplot3d import Axes3D' for
    # the above line to work.

    x, y, z = listOfXh[:,0].tolist(), listOfXh[:,1].tolist(), listOfXh[:,2].tolist()
    ax.scatter( x, y, z, c='g', marker='o' )

    x = listOfXhForOriginal8pts[:,0].tolist()
    y = listOfXhForOriginal8pts[:,1].tolist()
    z = listOfXhForOriginal8pts[:,2].tolist()
    ax.scatter( x, y, z, c='r', marker='o' )

    ax.plot( x3DptsList, y3DptsList, z3DptsList, c='b' )

    plt.show()

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

    img1 = cv2.imread( os.path.join( filepath, file1 ) )
    img2 = cv2.imread( os.path.join( filepath, file2 ) )

    for j in range(8):
        cv2.circle( img1, tuple(listOfPts1[j]), 3, (0,255,255), -1 )
```

```
        cv2.circle( img2, tuple(listOfPts2[j]), 3, (0,255,255), -1 )

    cv2.imshow( 'img1', img1 )
    cv2.imshow( 'img2', img2 )
    cv2.imwrite( f'handselected_points_{file1}', img1 )
    cv2.imwrite( f'handselected_points_{file2}', img2 )
    cv2.waitKey(0)
```