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

import numpy as np, cv2, os, time

#============================================================================
# ARINDAM BHANJA CHOWDHURY
# abhanjac@purdue.edu
# ECE 661 FALL 2018, HW 3
#============================================================================

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

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

# 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] ]

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

def homography( srcPts=None, dstPts=None ):
    '''
    The srcPts and dstPts are the list of points from which the 3x3 homography
    matrix (H) is to be decuced. The equation will be dstPts = H * srcPts.
    The srcPts and dstPts should be a list of at least 8 points (each point is
    in the homogeneous coordinate form, arranged as a sublist of 3 elements
    [x, y, 1]).
    The homography matrix is 3x3 but since only the ratios matter in homography,
    so one of the element can be taken as 1 and so there are only 8 unknowns.
    Here the last element of H is considered to be 1.
```

```
    '''

    # Number of source and destination points should be the same and should have
    # a one to one correspondence.
    if len(srcPts) != len(dstPts):
        print( 'srcPts and dstPts have different number of points. Aborting.' )
        return

    nPts = len( srcPts )

    # Creating matrix A and X (for the equation A * h = X).
    A, X = [], []
    for i in range( nPts ):
        # With each pair of points from srcPts and dstPts, 2 rows of A are made.
        xs, ys, xd, yd = srcPts[i][0], srcPts[i][1], dstPts[i][0], dstPts[i][1]
        row1 = [ xs, ys, 1, 0,  0,  0, -xs*xd, -ys*xd ]
        row2 = [ 0,  0,  0, xs, ys, 1, -xs*yd, -ys*yd ]
        A.append( row1 )
        A.append( row2 )
        X.append( xd )
        X.append( yd )
    #print( A )
    #print( X )

    # Converting A into nPts x nPts array and X into a nPts x 1 array.
    A, X = np.array( A ), np.reshape( X, ( nPts*2, 1 ) )
    Ainv = np.linalg.inv( A )
    h = np.matmul( Ainv, X )        # A * h = X, so h = Ainv * X.
    #print(h)

    # Appending a 1 for last element
    h = np.insert( h, nPts*2, 1 )
    H = np.reshape( h, (3,3) )      # Reshaping to 3x3.

    return H

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

# 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 to find the min and max x and y coordinates from a list of points.
maxX = lambda listOfPts: sorted( listOfPts, key=lambda x: x[0] )[-1][0]
minX = lambda listOfPts: sorted( listOfPts, key=lambda x: x[0] )[0][0]
maxY = lambda listOfPts: sorted( listOfPts, key=lambda x: x[1] )[-1][1]
minY = lambda listOfPts: sorted( listOfPts, key=lambda x: x[1] )[0][1]

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

def rectifyColor( pt=None, img=None ):
    '''
    This function takes in a point which is in float (not int) and an image and
    gives out a weighted average of the color of the surrounding pixels to
    which the point may be mapped to when converted from float to int.
    It is meant for those points which gets mapped to subpixel locations instead
    of mapping perfectly in an integer location in the given img.

    Format of pt is (x, y), like opencv.
    Returns the values as a numpy array.
    '''
    x1, y1 = np.floor( pt[0] ), np.floor( pt[1] )
```

```
    x2, y2 = np.ceil( pt[0] ),  np.floor( pt[1] )
    x3, y3 = np.floor( pt[0] ), np.ceil( pt[1] )
    x4, y4 = np.ceil( pt[0] ),  np.ceil( pt[1] )

    x, y = int( pt[0] ), int( pt[1] )   # Location where pt is to be mapped.

    # Distances of the potential location from the final location (x, y).
    # The + 0.0000001 is to prevent division by 0.
    dX1 = np.linalg.norm( np.array( [ x-x1, y-y1 ] ) ) + 0.0000001
    dX2 = np.linalg.norm( np.array( [ x-x2, y-y2 ] ) ) + 0.0000001
    dX3 = np.linalg.norm( np.array( [ x-x3, y-y3 ] ) ) + 0.0000001
    dX4 = np.linalg.norm( np.array( [ x-x4, y-y4 ] ) ) + 0.0000001

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

    h, w = img.shape[0], img.shape[1]

    #print( x1, y1, x2, y2, x3, y3, x4, y4 )

    # This is done so that while taking int(), the x, y dont go out of bound.
    x1 = int(x1) if int(x1) < w else w-1
    x2 = int(x2) if int(x2) < w else w-1
    x3 = int(x3) if int(x3) < w else w-1
    x4 = int(x4) if int(x4) < w else w-1

    y1 = int(y1) if int(y1) < h else h-1
    y2 = int(y2) if int(y2) < h else h-1
    y3 = int(y3) if int(y3) < h else h-1
    y4 = int(y4) if int(y4) < h else h-1

    # Color values at the above locations.
    C1, C2, C3, C4 = img[ int(y1) ][ int(x1) ], img[ int(y2) ][ int(x2) ], \
                     img[ int(y3) ][ int(x3) ], img[ int(y4) ][ int(x4) ]

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

    # Final color. So the farther the potential location is from the final
    # location, more is the corresponding distance, and hence less will be the
    # effect of the color of that location on the final color. In the above
    # calculations basically the following equation is evaluated.
    C = ( (C1 / dX1) + (C2 / dX2) + (C3 / dX3) + (C4 / dX4) ) / ( \
                     (1 / dX1) + (1 / dX2) + (1 / dX3) + (1 / dX4) )

    C = np.asarray( C, dtype=np.uint8 )

    return C

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

def mapImages( sourceImg=None, targetImg=None, pqrs=None, H=None ):
    '''
    This function takes in a source image, a target image and the four
    corner points (pqrs) of the target that defines the region where the source
    image is to be projected. It also takes in homography matrix from the target
    to the source image. Returns the projected image.

    pqrs should be a list of points. Each point in the list is a sublist of x
    and y coordinate of the point. These should be in planar (not homogeneous)
    coordinate.
    '''

    # Drawing a black polygon to make all the pixels in the target region = 0,
    # before mapping.
    targetImg = cv2.fillPoly( targetImg, np.array( [ pqrs ] ), (0,0,0) )
```

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

    processingTime = time.time()

    # Mapping the points.
    # Scanning only the region between the points p, q, r, s.
    for r in range( minY( pqrs ), maxY( pqrs ) ):
        for c in range( minX( pqrs ), maxX( pqrs ) ):

            if targetImg[r][c].all() == 0:
                # If point is in the 0 polygon.
                # which indicates that the point is in the black polygon where
                # the source image is to be projected.
                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][c] = sourceImg[ int( sourcePt[1] ) ][ \
                                                 int( sourcePt[0] ) ]
                    #targetImg[r][c] = rectifyColor( pt=sourcePt, img=sourceImg )
                    pass

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

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

#================================================================================
# 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 ]

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

def createLAandLB( line1ProjRemovedPerp=None, line2ProjRemovedPerp=None, \
                   line3ProjRemovedPerp=None, line4ProjRemovedPerp=None ):
    '''
    This function creates the LA and LB matrices, such that LA * [s11, s12]^T = LB,
    that will be used to find the S matrix later.
    '''
    l1, l2 = line1ProjRemovedPerp[0], line1ProjRemovedPerp[1]
    m1, m2 = line2ProjRemovedPerp[0], line2ProjRemovedPerp[1]
    l3, l4 = line3ProjRemovedPerp[0], line3ProjRemovedPerp[1]
    m3, m4 = line4ProjRemovedPerp[0], line4ProjRemovedPerp[1]

    LA = [ [ l1*m1, l1*m2 + l2*m1 ], [ l3*m3, l3*m4 + l4*m3 ] ]
    LB = [ -1*l2*m2, -1*l4*m4 ]

    return np.array(LA), np.array(LB)

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

def modifyImg3( sourceImg=None, H=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] ) ]
                #targetImg[r][c] = rectifyColor( pt=sourcePt, img=sourceImg )
                pass

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

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

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


if __name__ == '__main__':

    # TASK 1.1.

    filePath = './HW3Pics'
    filename1, filename2 = '1.jpg', '2.jpg'

    # Reading the points.

    #pqrsFig1 = selectPts( filePath=os.path.join( filePath, filename1 ) )
    pqrsFig1 = [[238, 1121], [2207, 20], [2332, 1401], [105, 1753]]
    #print( 'Points of {}: {}\n'.format( filename1, pqrsFig1 ) )

    #pqrsFig2 = selectPts( filePath=os.path.join( filePath, filename2 ) )
    pqrsFig2 = [[239, 58], [338, 73], [336, 276], [236, 282]]
    #print( 'Points of {}: {}\n'.format( filename2, pqrsFig2 ) )

    pqrsFig1Orig = [[0, 0], [1260, 0], [1260, 560], [0, 560]]
    #print( 'Points of {}: {}\n'.format( filename1Orig, pqrsFig1Orig ) )

    pqrsFig2Orig = [[0, 0], [160, 0], [160, 320], [0, 320]]
    #print( 'Points of {}: {}\n'.format( filename2Orig, pqrsFig2Orig ) )

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

    # Converting points to homogeneous coordinates.

    pqrsHomFmt1 = [ homogeneousFormat( pt ) for pt in pqrsFig1 ]
    pqrsHomFmt2 = [ homogeneousFormat( pt ) for pt in pqrsFig2 ]
    pqrsHomFmt1Orig = [ homogeneousFormat( pt ) for pt in pqrsFig1Orig ]
    pqrsHomFmt2Orig = [ homogeneousFormat( pt ) for pt in pqrsFig2Orig ]

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

    # Finding the homography.

    # Homography between fig1 and Original Fig1.
    Hbetw1OrigTo1 = homography( srcPts=pqrsHomFmt1Orig, dstPts=pqrsHomFmt1 )
    print( 'Homography between Original Undistorted version -> {}: \n{}\n'.format( \
                            filename1, Hbetw1OrigTo1 ) )

    # Homography between fig2 and Original Fig2.
    Hbetw2OrigTo2 = homography( srcPts=pqrsHomFmt2Orig, dstPts=pqrsHomFmt2 )
    print( 'Homography between Original Undistorted version -> {}: \n{}\n'.format( \
                            filename2, Hbetw2OrigTo2 ) )
```

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

    # Implanting the target into fig 1.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename1 ) )
    srcH, srcW, _ = sourceImg.shape
    H = Hbetw1OrigTo1

    targetImg = modifyImg3( sourceImg=sourceImg, H=H )

    # Resizing for the purpose of display.
    tgtH, tgtW, _ = targetImg.shape
    if tgtH < 480 or tgtW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    targetImgShow = cv2.resize( targetImg, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', targetImgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

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

    # Implanting the target into fig 2.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename2 ) )
    srcH, srcW, _ = sourceImg.shape
    H = Hbetw2OrigTo2

    targetImg = modifyImg3( sourceImg=sourceImg, H=H )

    # Resizing for the purpose of display.
    tgtH, tgtW, _ = targetImg.shape
    print( tgtH, tgtW )
    if tgtH < 480 or tgtW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    targetImgShow = cv2.resize( targetImg, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', targetImgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

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

    # TASK 2.1.

    filePath = './PicsSelf'
    filename1, filename2 = '1.jpg', '2.jpg'

    # Reading the points.

    #pqrsFig1 = selectPts( filePath=os.path.join( filePath, filename1 ) )
    pqrsFig1 = [[661, 1689], [965, 1577], [972, 2409], [661, 2296]]
    #print( 'Points of {}: {}\n'.format( filename1, pqrsFig1 ) )

    #pqrsFig2 = selectPts( filePath=os.path.join( filePath, filename2 ) )
    #pqrsFig2 = [[1456, 492], [2873, 599], [2905, 1564], [1547, 2023]]
    pqrsFig2 = [[127, 12], [429, 62], [432, 182], [169, 205]]
    #print( 'Points of {}: {}\n'.format( filename2, pqrsFig2 ) )

    pqrsFig1Orig = [[0, 0], [209, 0], [209, 130], [0, 130]]
    #print( 'Points of {}: {}\n'.format( filename1Orig, pqrsFig1Orig ) )

    pqrsFig2Orig = [[0, 0], [186, 0], [186, 123], [0, 123]]
```

```
    #print( 'Points of {}: {}\n'.format( filename2Orig, pqrsFig2Orig ) )

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

    # Converting points to homogeneous coordinates.

    pqrsHomFmt1 = [ homogeneousFormat( pt ) for pt in pqrsFig1 ]
    pqrsHomFmt2 = [ homogeneousFormat( pt ) for pt in pqrsFig2 ]
    pqrsHomFmt1Orig = [ homogeneousFormat( pt ) for pt in pqrsFig1Orig ]
    pqrsHomFmt2Orig = [ homogeneousFormat( pt ) for pt in pqrsFig2Orig ]

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

    # Finding the homography.

    # Homography between fig1 and Original Fig1.
    Hbetw1OrigTo1 = homography( srcPts=pqrsHomFmt1Orig, dstPts=pqrsHomFmt1 )
    print( 'Homography between Original Undistorted version -> {}: \n{}\n'.format( \
                            filename1, Hbetw1OrigTo1 ) )

    # Homography between fig2 and Original Fig2.
    Hbetw2OrigTo2 = homography( srcPts=pqrsHomFmt2Orig, dstPts=pqrsHomFmt2 )
    print( 'Homography between Original Undistorted version -> {}: \n{}\n'.format( \
                            filename2, Hbetw2OrigTo2 ) )

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

    # Implanting the target into fig 1.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename1 ) )
    srcH, srcW, _ = sourceImg.shape
    H = Hbetw1OrigTo1

    targetImg = modifyImg3( sourceImg=sourceImg, H=H )

    # Resizing for the purpose of display.
    tgtH, tgtW, _ = targetImg.shape
    if tgtH < 480 or tgtW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    targetImgShow = cv2.resize( targetImg, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', targetImgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

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

    # Implanting the target into fig 2.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename2 ) )
    srcH, srcW, _ = sourceImg.shape
    H = Hbetw2OrigTo2

    targetImg = modifyImg3( sourceImg=sourceImg, H=H )

    # Resizing for the purpose of display.
    tgtH, tgtW, _ = targetImg.shape
    print( tgtH, tgtW )
    if tgtH < 480 or tgtW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    targetImgShow = cv2.resize( targetImg, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', targetImgShow )
```

```
    cv2.waitKey(0)
    cv2.destroyAllWindows()

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

    # TASK 1.2.1.   Remove projective distortion.

    filePath = './HW3Pics'
    filename1, filename2 = '1.jpg', '2.jpg'

    # Reading the points.

    pqrsFig1 = selectPts( filePath=os.path.join( filePath, filename1 ) )
    pqrsFig1 = [[2029, 405], [251, 1251], [2085, 1441], [149, 1745], [149, 1745], [251, 1251], [2085, 1441],
[2029, 405]]
    #print( 'Points of {}: {}\n'.format( filename1, pqrsFig1 ) )

    pqrsFig2 = selectPts( filePath=os.path.join( filePath, filename2 ) )
    pqrsFig2 = [[46, 121], [139, 128], [209, 355], [344, 339], [237, 282], [237, 57], [498, 267], [495, 26]]
    #print( 'Points of {}: {}\n'.format( filename2, pqrsFig2 ) )

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

    # Converting points to homogeneous coordinates.

    pqrsHomFmt1 = [ homogeneousFormat( pt ) for pt in pqrsFig1 ]
    pqrsHomFmt2 = [ homogeneousFormat( pt ) for pt in pqrsFig2 ]

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

    # Removing projective distortion fig 1.

    # Finding the line joining the points selected in the figure 1.
    # These are the lines which are supposed to be parallal in the undistorted image.
    line1Fig1Para = np.cross( pqrsHomFmt1[0], pqrsHomFmt1[1] )
    line2Fig1Para = np.cross( pqrsHomFmt1[2], pqrsHomFmt1[3] )
    line3Fig1Para = np.cross( pqrsHomFmt1[4], pqrsHomFmt1[5] )
    line4Fig1Para = np.cross( pqrsHomFmt1[6], pqrsHomFmt1[7] )

    # Calculating vanishing points.
    VP1fig1 = np.cross( line1Fig1Para, line2Fig1Para )
    VP2fig1 = np.cross( line3Fig1Para, line4Fig1Para )

    # The magnitude of the vanishing points are too large. But since they are
    # the physical points in homogeneous coordinates, so we can always make the
    # magnitude smaller by dividing by one of the coordinate values and express
    # them equvalently in [x, y, 1] format (instead of having some
    # [very_large_X, very_large_Y, large_number] format).

    # Divide by the 3rd element to make the overall magnitude smaller
    VP1fig1 = VP1fig1 / VP1fig1[2]
    VP2fig1 = VP2fig1 / VP2fig1[2]

    # Calculating vanishing line.
    VLfig1 = np.cross( VP1fig1, VP2fig1 )

    # Divide by the 3rd element to make the overall magnitude smaller.
    VLfig1 = VLfig1 / VLfig1[2]

    # Calculating the Homography matrix.
    Hbetw1OrigTo1Proj = np.eye( 3 )
    Hbetw1OrigTo1Proj[2] = VLfig1

    print( 'Hbetw1OrigTo1Proj: \n{}\n'.format( Hbetw1OrigTo1Proj ) )
```

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

    # Removing projective distortion fig 2.

    # Finding the line joining the points selected in the figure 2.
    # These are the lines which are supposed to be parallal in the undistorted image.
    line1Fig2Para = np.cross( pqrsHomFmt2[0], pqrsHomFmt2[1] )
    line2Fig2Para = np.cross( pqrsHomFmt2[2], pqrsHomFmt2[3] )
    line3Fig2Para = np.cross( pqrsHomFmt2[4], pqrsHomFmt2[5] )
    line4Fig2Para = np.cross( pqrsHomFmt2[6], pqrsHomFmt2[7] )

    # Calculating vanishing points.
    VP1fig2 = np.cross( line1Fig2Para, line2Fig2Para )
    VP2fig2 = np.cross( line3Fig2Para, line4Fig2Para )

    # Divide by the 3rd element to make the overall magnitude smaller
    VP1fig2 = VP1fig2 / VP1fig2[2]
    VP2fig2 = VP2fig2 / VP2fig2[2]

    # Calculating vanishing line.
    VLfig2 = np.cross( VP1fig2, VP2fig2 )

    # Divide by the 3rd element to make the overall magnitude smaller.
    VLfig2 = VLfig2 / VLfig2[2]

    # Calculating the Homography matrix.
    Hbetw2OrigTo2Proj = np.eye( 3 )
    Hbetw2OrigTo2Proj[2] = VLfig2

    print( 'Hbetw2OrigTo2Proj: \n{}\n'.format( Hbetw2OrigTo2Proj ) )

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

    # Implanting the target into figure 1.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename1 ) )
    srcH, srcW = sourceImg.shape[0], sourceImg.shape[1]
    H = Hbetw1OrigTo1Proj

    # Taking the corner points of the distorted image and mapping them using
    # the homography to know the dimensions of the output image.
    outputIndex = []
    for r in [ 0, srcH ]:
        for c in [ 0, srcW ]:
            inputPt = homogeneousFormat( [c, r] )
            outputPt = applyHomography( ( H, inputPt ) )
            outputPt = planarFormat( outputPt )
            outputIndex.append( outputPt )

    # Now there are points which are mapped to negative coordinates. So finding
    # those out so that those coordinates can be made (0, 0) and then all the
    # pixels can then be added by those values to make all of them positive.
    outputIndex = np.array(outputIndex)
    minIndex = np.amin( outputIndex, axis=0 )   # This may be a -ve value.
    newOutIndex = outputIndex - minIndex
    newMaxIndex = np.amax( newOutIndex, axis=0 )    # New max index.

    # Now creating a new blank image where the pixels will be drawn.
    # The +1 is because if the max col, row index is 639, 479, then image is 640 x 480.
    newW, newH = newMaxIndex[0] + 1, newMaxIndex[1] + 1

    print(newW, newH)
    print( srcW, srcH )
```

```
    img = np.zeros( ( newH, newW, 3 ), dtype=np.uint8 )
    for r in range( newH ):
        for c in range( newW ):
            targetPt = homogeneousFormat( [c + minIndex[0], r + minIndex[1] ] )
            sourcePt = applyHomography( [ np.linalg.inv( H ), targetPt ] )
            x, y = planarFormat( sourcePt )
            if x > 0 and y > 0 and x < srcW and y < srcH:
                img[r][c] = sourceImg[y][x]

    # Resizing for the purpose of display.
    imgH, imgW = img.shape[0], img.shape[1]

    #print(imgH, imgW)

    if imgH < 480 or imgW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    imgShow = cv2.resize( img, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', imgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

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

    # Implanting the target into figure 2.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename2 ) )
    srcH, srcW = sourceImg.shape[0], sourceImg.shape[1]
    H = Hbetw2OrigTo2Proj

    # Taking the corner points of the distorted image and mapping them using
    # the homography to know the dimensions of the output image.
    outputIndex = []
    for r in [ 0, srcH ]:
        for c in [ 0, srcW ]:
            inputPt = homogeneousFormat( [c, r] )
            outputPt = applyHomography( ( H, inputPt ) )
            outputPt = planarFormat( outputPt )
            outputIndex.append( outputPt )

    # Now there are points which are mapped to negative coordinates. So finding
    # those out so that those coordinates can be made (0, 0) and then all the
    # pixels can then be added by those values to make all of them positive.
    outputIndex = np.array(outputIndex)
    minIndex = np.amin( outputIndex, axis=0 )   # This may be a -ve value.
    newOutIndex = outputIndex - minIndex
    newMaxIndex = np.amax( newOutIndex, axis=0 )    # New max index.

    # Now creating a new blank image where the pixels will be drawn.
    # The +1 is because if the max col, row index is 639, 479, then image is 640 x 480.
    newW, newH = newMaxIndex[0] + 1, newMaxIndex[1] + 1

    print(newW, newH)
    print( srcW, srcH )

    img = np.zeros( ( newH, newW, 3 ), dtype=np.uint8 )
    for r in range( newH ):
        for c in range( newW ):
            targetPt = homogeneousFormat( [c + minIndex[0], r + minIndex[1] ] )
            sourcePt = applyHomography( [ np.linalg.inv( H ), targetPt ] )
            x, y = planarFormat( sourcePt )
            if x > 0 and y > 0 and x < srcW and y < srcH:
                img[r][c] = sourceImg[y][x]
```

```
    # Resizing for the purpose of display.
    imgH, imgW = img.shape[0], img.shape[1]

    #print(imgH, imgW)

    if imgH < 480 or imgW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    imgShow = cv2.resize( img, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', imgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

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

    # TASK 1.2.2.   Remove affine distortion.

    # Removing affine distortion figure 1.

    #pqrsFig1 = selectPts( filePath=os.path.join( filePath, filename1 ) )
    pqrsFig1 = [[238, 1117], [2215, 16], [2361, 1802], [2219, 125], [1255, 765], [1668, 1300], [1668, 571],
[1210, 1413]]
    #print( 'Points of {}: {}\n'.format( filename1, pqrsFig1 ) )

    #pqrsFig2 = selectPts( filePath=os.path.join( filePath, filename2 ) )
    pqrsFig2 = [[237, 58], [338, 74], [336, 276], [338, 86], [245, 72], [326, 174], [326, 82], [245, 167]]
    #print( 'Points of {}: {}\n'.format( filename2, pqrsFig2 ) )

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

    # Converting points to homogeneous coordinates.

    pqrsHomFmt1 = [ homogeneousFormat( pt ) for pt in pqrsFig1 ]
    pqrsHomFmt2 = [ homogeneousFormat( pt ) for pt in pqrsFig2 ]

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

    # Removing affine distortion from figure 1.

    # These are the lines which are supposed to be perpendicular in the undistorted image.
    # Need to find s11, s12, s21, s22. Now s12 = s21 and s22 can be assumed to be 1.
    # So need to find s11 and s12. With two perpendicular lines one equation of
    # containing s11 and s12 can be formed. So four perpendicular lines are needed
    # to have two equations containing s11 and s12. Then they need to be solved.
    line1Fig1Perp = np.cross( pqrsHomFmt1[0], pqrsHomFmt1[1] )
    line2Fig1Perp = np.cross( pqrsHomFmt1[2], pqrsHomFmt1[3] )
    line3Fig1Perp = np.cross( pqrsHomFmt1[4], pqrsHomFmt1[5] )
    line4Fig1Perp = np.cross( pqrsHomFmt1[6], pqrsHomFmt1[7] )

    # Creating the line free of perspective distortion.
    H = Hbetw1OrigTo1Proj
    HinvT = np.linalg.inv( np.transpose( H ) )

    line1Fig1ProjRemovedPerp = np.matmul( HinvT, line1Fig1Perp )
    line2Fig1ProjRemovedPerp = np.matmul( HinvT, line2Fig1Perp )
    line3Fig1ProjRemovedPerp = np.matmul( HinvT, line3Fig1Perp )
    line4Fig1ProjRemovedPerp = np.matmul( HinvT, line4Fig1Perp )

    # Divide by the 3rd element to make the overall magnitude smaller.
    line1Fig1ProjRemovedPerp /= line1Fig1ProjRemovedPerp[2]
    line2Fig1ProjRemovedPerp /= line2Fig1ProjRemovedPerp[2]
    line3Fig1ProjRemovedPerp /= line3Fig1ProjRemovedPerp[2]
    line4Fig1ProjRemovedPerp /= line4Fig1ProjRemovedPerp[2]

    # Creating the LA and LB matrices. Such that LA * [s11, s12]^T = LB.
```

```
    LA, LB = createLAandLB( line1Fig1ProjRemovedPerp, line2Fig1ProjRemovedPerp, \
                      line3Fig1ProjRemovedPerp, line4Fig1ProjRemovedPerp )

    # Finding the s11 and s12.
    LAinv = np.linalg.inv( LA )
    s = np.matmul( LAinv, LB )
    S = [ [ s[0], s[1] ], [ s[1], 1 ] ]
    S = np.array( S )
    print( 'S: ', S )

    # Finding the A matrix to remove the affine distortion.
    U, E, V_T = np.linalg.svd( S )
    E, V = np.diag( E ), np.transpose( V_T )
    D = np.sqrt( E )
    A = np.matmul( np.matmul( V, D ), V_T )
    print( 'A: ', A )

    # Finding the Homography to remove affine distortion.
    Hbetw1OrigTo1Aff = [ [ A[0,0], A[1,0], 0 ], [ A[0,1], A[1,1], 0 ], [ 0, 0, 1 ] ]
    Hbetw1OrigTo1Aff = np.array( Hbetw1OrigTo1Aff )
    Hbetw1OrigTo1Aff = np.linalg.inv( Hbetw1OrigTo1Aff )
    print( 'Hbetw1OrigTo1Aff: \n{}\n'.format( Hbetw1OrigTo1Aff ) )

    Hbetw1OrigTo1 = np.matmul( Hbetw1OrigTo1Aff, Hbetw1OrigTo1Proj )
    print( 'Hbetw1OrigTo1: \n{}\n'.format( Hbetw1OrigTo1 ) )

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

    # Implanting the target into figure 1.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename1 ) )
    srcH, srcW = sourceImg.shape[0], sourceImg.shape[1]
    H = Hbetw1OrigTo1

    # Taking the corner points of the distorted image and mapping them using
    # the homography to know the dimensions of the output image.
    outputIndex = []
    for r in [ 0, srcH ]:
        for c in [ 0, srcW ]:
            inputPt = homogeneousFormat( [c, r] )
            outputPt = applyHomography( ( H, inputPt ) )
            outputPt = planarFormat( outputPt )
            outputIndex.append( outputPt )

    # Now there are points which are mapped to negative coordinates. So finding
    # those out so that those coordinates can be made (0, 0) and then all the
    # pixels can then be added by those values to make all of them positive.
    outputIndex = np.array(outputIndex)
    minIndex = np.amin( outputIndex, axis=0 )   # This may be a -ve value.
    newOutIndex = outputIndex - minIndex
    newMaxIndex = np.amax( newOutIndex, axis=0 )    # New max index.

    # Now creating a new blank image where the pixels will be drawn.
    # The +1 is because if the max col, row index is 639, 479, then image is 640 x 480.
    newW, newH = newMaxIndex[0] + 1, newMaxIndex[1] + 1

    print(newW, newH)
    print( srcW, srcH )

    img = np.zeros( ( newH, newW, 3 ), dtype=np.uint8 )
    for r in range( newH ):
        for c in range( newW ):
            targetPt = homogeneousFormat( [c + minIndex[0], r + minIndex[1] ] )
            sourcePt = applyHomography( [ np.linalg.inv( H ), targetPt ] )
```

```
            x, y = planarFormat( sourcePt )
            if x > 0 and y > 0 and x < srcW and y < srcH:
                img[r][c] = sourceImg[y][x]

    # Resizing for the purpose of display.
    imgH, imgW = img.shape[0], img.shape[1]

    #print(imgH, imgW)

    if imgH < 480 or imgW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    imgShow = cv2.resize( img, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', imgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

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

    # Removing affine distortion figure 2.

    # These are the lines which are supposed to be perpendicular in the undistorted image.
    # Need to find s11, s12, s21, s22. Now s12 = s21 and s22 can be assumed to be 1.
    # So need to find s11 and s12. With two perpendicular lines one equation of
    # containing s11 and s12 can be formed. So four perpendicular lines are needed
    # to have two equations containing s11 and s12. Then they need to be solved.
    line1Fig2Perp = np.cross( pqrsHomFmt2[0], pqrsHomFmt2[1] )
    line2Fig2Perp = np.cross( pqrsHomFmt2[2], pqrsHomFmt2[3] )
    line3Fig2Perp = np.cross( pqrsHomFmt2[4], pqrsHomFmt2[5] )
    line4Fig2Perp = np.cross( pqrsHomFmt2[6], pqrsHomFmt2[7] )

    # Creating the line free of perspective distortion.
    H = Hbetw2OrigTo2Proj
    HinvT = np.linalg.inv( np.transpose( H ) )

    line1Fig2ProjRemovedPerp = np.matmul( HinvT, line1Fig2Perp )
    line2Fig2ProjRemovedPerp = np.matmul( HinvT, line2Fig2Perp )
    line3Fig2ProjRemovedPerp = np.matmul( HinvT, line3Fig2Perp )
    line4Fig2ProjRemovedPerp = np.matmul( HinvT, line4Fig2Perp )

    # Divide by the 3rd element to make the overall magnitude smaller.
    line1Fig2ProjRemovedPerp /= line1Fig2ProjRemovedPerp[2]
    line2Fig2ProjRemovedPerp /= line2Fig2ProjRemovedPerp[2]
    line3Fig2ProjRemovedPerp /= line3Fig2ProjRemovedPerp[2]
    line4Fig2ProjRemovedPerp /= line4Fig2ProjRemovedPerp[2]

    # Creating the LA and LB matrices. Such that LA * [s11, s12]^T = LB.
    LA, LB = createLAandLB( line1Fig2ProjRemovedPerp, line2Fig2ProjRemovedPerp, \
                        line3Fig2ProjRemovedPerp, line4Fig2ProjRemovedPerp )

    # Finding the s11 and s12.
    LAinv = np.linalg.inv( LA )
    s = np.matmul( LAinv, LB )
    S = [ [ s[0], s[1] ], [ s[1], 1 ] ]
    S = np.array( S )
    print( 'S: ', S )

    # Finding the A matrix to remove the affine distortion.
    U, E, V_T = np.linalg.svd( S )
    E, V = np.diag( E ), np.transpose( V_T )
    D = np.sqrt( E )
    A = np.matmul( np.matmul( V, D ), V_T )
    print( 'A: ', A )

    # Finding the Homography to remove affine distortion.
```

```
    Hbetw2OrigTo2Aff = [ [ A[0,0], A[0,1], 0 ], [ A[1,0], A[1,1], 0 ], [ 0, 0, 1 ] ]
    Hbetw2OrigTo2Aff = np.array( Hbetw2OrigTo2Aff )
    Hbetw2OrigTo2Aff = np.linalg.inv( Hbetw2OrigTo2Aff )
    print( 'Hbetw2OrigTo2Aff: \n{}\n'.format( Hbetw2OrigTo2Aff ) )

    Hbetw2OrigTo2 = np.matmul( Hbetw2OrigTo2Aff, Hbetw2OrigTo2Proj )
    print( 'Hbetw2OrigTo2: \n{}\n'.format( Hbetw2OrigTo2 ) )

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

    # Implanting the target into figure 2.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename2 ) )
    srcH, srcW = sourceImg.shape[0], sourceImg.shape[1]
    H = Hbetw2OrigTo2

    # Taking the corner points of the distorted image and mapping them using
    # the homography to know the dimensions of the output image.
    outputIndex = []
    for r in [ 0, srcH ]:
        for c in [ 0, srcW ]:
            inputPt = homogeneousFormat( [c, r] )
            outputPt = applyHomography( ( H, inputPt ) )
            outputPt = planarFormat( outputPt )
            outputIndex.append( outputPt )

    # Now there are points which are mapped to negative coordinates. So finding
    # those out so that those coordinates can be made (0, 0) and then all the
    # pixels can then be added by those values to make all of them positive.
    outputIndex = np.array(outputIndex)
    minIndex = np.amin( outputIndex, axis=0 )   # This may be a -ve value.
    newOutIndex = outputIndex - minIndex
    newMaxIndex = np.amax( newOutIndex, axis=0 )    # New max index.

    # Now creating a new blank image where the pixels will be drawn.
    # The +1 is because if the max col, row index is 639, 479, then image is 640 x 480.
    newW, newH = newMaxIndex[0] + 1, newMaxIndex[1] + 1

    print(newW, newH)
    print( srcW, srcH )

    img = np.zeros( ( newH, newW, 3 ), dtype=np.uint8 )
    for r in range( newH ):
        for c in range( newW ):
            targetPt = homogeneousFormat( [c + minIndex[0], r + minIndex[1] ] )
            sourcePt = applyHomography( [ np.linalg.inv( H ), targetPt ] )
            x, y = planarFormat( sourcePt )
            if x > 0 and y > 0 and x < srcW and y < srcH:
                img[r][c] = sourceImg[y][x]

    # Resizing for the purpose of display.
    imgH, imgW = img.shape[0], img.shape[1]

    #print(imgH, imgW)

    if imgH < 480 or imgW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    imgShow = cv2.resize( img, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', imgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

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

```
    # TASK 2.2.1.   Remove projective distortion.

    filePath = './PicsSelf'
    filename1, filename2 = '1.jpg', '2.jpg'

    # Reading the points.

    #pqrsFig1 = selectPts( filePath=os.path.join( filePath, filename1 ) )
    #pqrsFig1 = [[2029, 405], [251, 1251], [2085, 1441], [149, 1745], [149, 1745], [251, 1251], [2085, 1441],
[2029, 405]]
    #pqrsFig1 = [[661, 1689], [965, 1577], [972, 2409], [661, 2296], [661, 2296], [965, 1577], [972, 2409],
[661, 1689]]
    pqrsFig1 = [[661, 1672], [965, 1560], [665, 2279], [972, 2409], [1144, 1100], [1129, 3172], [1868, 606],
[1820, 3839]]
    #print( 'Points of {}: {}\n'.format( filename1, pqrsFig1 ) )

    #pqrsFig2 = selectPts( filePath=os.path.join( filePath, filename2 ) )
    #pqrsFig2 = [[46, 121], [139, 128], [209, 355], [344, 339], [237, 282], [237, 57], [498, 267], [495, 26]]
    #pqrsFig2 = [[1456, 492], [2873, 599], [2905, 1564], [1547, 2023], [1547, 2023], [2873, 599], [2905,
1564], [1456, 492]]
    #pqrsFig2 = [[1456, 497], [2860, 599], [1547, 2013], [2899, 1564], [1592, 580], [1670, 1896], [2827,
648], [2827, 1535]]
    pqrsFig2 = [[128, 12], [427, 63], [167, 206], [431, 183], [129, 25], [166, 195], [428, 78], [431, 169]]
    #pqrsFig2 = [[234, 52], [780, 268], [307, 892], [789, 784], [235, 116], [303, 840], [782, 333], [789,
710]]
    #print( 'Points of {}: {}\n'.format( filename2, pqrsFig2 ) )

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

    # Converting points to homogeneous coordinates.

    pqrsHomFmt1 = [ homogeneousFormat( pt ) for pt in pqrsFig1 ]
    pqrsHomFmt2 = [ homogeneousFormat( pt ) for pt in pqrsFig2 ]

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

    # Removing projective distortion fig 1.

    # Finding the line joining the points selected in the figure 1.
    # These are the lines which are supposed to be parallal in the undistorted image.
    line1Fig1Para = np.cross( pqrsHomFmt1[0], pqrsHomFmt1[1] )
    line2Fig1Para = np.cross( pqrsHomFmt1[2], pqrsHomFmt1[3] )
    line3Fig1Para = np.cross( pqrsHomFmt1[4], pqrsHomFmt1[5] )
    line4Fig1Para = np.cross( pqrsHomFmt1[6], pqrsHomFmt1[7] )

    # Calculating vanishing points.
    VP1fig1 = np.cross( line1Fig1Para, line2Fig1Para )
    VP2fig1 = np.cross( line3Fig1Para, line4Fig1Para )

    # The magnitude of the vanishing points are too large. But since they are
    # the physical points in homogeneous coordinates, so we can always make the
    # magnitude smaller by dividing by one of the coordinate values and express
    # them equvalently in [x, y, 1] format (instead of having some
    # [very_large_X, very_large_Y, large_number] format).

    # Divide by the 3rd element to make the overall magnitude smaller
    VP1fig1 = VP1fig1 / VP1fig1[2]
    VP2fig1 = VP2fig1 / VP2fig1[2]

    # Calculating vanishing line.
    VLfig1 = np.cross( VP1fig1, VP2fig1 )

    # Divide by the 3rd element to make the overall magnitude smaller.
    VLfig1 = VLfig1 / VLfig1[2]
```

```
    # Calculating the Homography matrix.
    Hbetw1OrigTo1Proj = np.eye( 3 )
    Hbetw1OrigTo1Proj[2] = VLfig1

    print( 'Hbetw1OrigTo1Proj: \n{}\n'.format( Hbetw1OrigTo1Proj ) )

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

    # Removing projective distortion fig 2.

    # Finding the line joining the points selected in the figure 2.
    # These are the lines which are supposed to be parallal in the undistorted image.
    line1Fig2Para = np.cross( pqrsHomFmt2[0], pqrsHomFmt2[1] )
    line2Fig2Para = np.cross( pqrsHomFmt2[2], pqrsHomFmt2[3] )
    line3Fig2Para = np.cross( pqrsHomFmt2[4], pqrsHomFmt2[5] )
    line4Fig2Para = np.cross( pqrsHomFmt2[6], pqrsHomFmt2[7] )

    # Calculating vanishing points.
    VP1fig2 = np.cross( line1Fig2Para, line2Fig2Para )
    VP2fig2 = np.cross( line3Fig2Para, line4Fig2Para )

    # Divide by the 3rd element to make the overall magnitude smaller
    VP1fig2 = VP1fig2 / VP1fig2[2]
    VP2fig2 = VP2fig2 / VP2fig2[2]

    # Calculating vanishing line.
    VLfig2 = np.cross( VP1fig2, VP2fig2 )

    # Divide by the 3rd element to make the overall magnitude smaller.
    VLfig2 = VLfig2 / VLfig2[2]

    # Calculating the Homography matrix.
    Hbetw2OrigTo2Proj = np.eye( 3 )
    Hbetw2OrigTo2Proj[2] = VLfig2

    print( 'Hbetw2OrigTo2Proj: \n{}\n'.format( Hbetw2OrigTo2Proj ) )

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

    # Implanting the target into figure 1.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename1 ) )
    srcH, srcW = sourceImg.shape[0], sourceImg.shape[1]
    H = Hbetw1OrigTo1Proj

    # Taking the corner points of the distorted image and mapping them using
    # the homography to know the dimensions of the output image.
    outputIndex = []
    for r in [ 0, srcH ]:
        for c in [ 0, srcW ]:
            inputPt = homogeneousFormat( [c, r] )
            outputPt = applyHomography( ( H, inputPt ) )
            outputPt = planarFormat( outputPt )
            outputIndex.append( outputPt )

    # Now there are points which are mapped to negative coordinates. So finding
    # those out so that those coordinates can be made (0, 0) and then all the
    # pixels can then be added by those values to make all of them positive.
    outputIndex = np.array(outputIndex)
    minIndex = np.amin( outputIndex, axis=0 )   # This may be a -ve value.
    newOutIndex = outputIndex - minIndex
    newMaxIndex = np.amax( newOutIndex, axis=0 )    # New max index.
```

```
    # Now creating a new blank image where the pixels will be drawn.
    # The +1 is because if the max col, row index is 639, 479, then image is 640 x 480.
    newW, newH = newMaxIndex[0] + 1, newMaxIndex[1] + 1

    print(newW, newH)
    print( srcW, srcH )

    img = np.zeros( ( newH, newW, 3 ), dtype=np.uint8 )
    for r in range( newH ):
        for c in range( newW ):
            targetPt = homogeneousFormat( [c + minIndex[0], r + minIndex[1] ] )
            sourcePt = applyHomography( [ np.linalg.inv( H ), targetPt ] )
            x, y = planarFormat( sourcePt )
            if x > 0 and y > 0 and x < srcW and y < srcH:
                img[r][c] = sourceImg[y][x]

    # Resizing for the purpose of display.
    imgH, imgW = img.shape[0], img.shape[1]

    #print(imgH, imgW)

    if imgH < 480 or imgW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    imgShow = cv2.resize( img, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', imgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

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

    # Implanting the target into figure 2.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename2 ) )
    srcH, srcW = sourceImg.shape[0], sourceImg.shape[1]
    H = Hbetw2OrigTo2Proj

    # Taking the corner points of the distorted image and mapping them using
    # the homography to know the dimensions of the output image.
    outputIndex = []
    for r in [ 0, srcH ]:
        for c in [ 0, srcW ]:
            inputPt = homogeneousFormat( [c, r] )
            outputPt = applyHomography( ( H, inputPt ) )
            outputPt = planarFormat( outputPt )
            outputIndex.append( outputPt )

    # Now there are points which are mapped to negative coordinates. So finding
    # those out so that those coordinates can be made (0, 0) and then all the
    # pixels can then be added by those values to make all of them positive.
    outputIndex = np.array(outputIndex)
    minIndex = np.amin( outputIndex, axis=0 )   # This may be a -ve value.
    newOutIndex = outputIndex - minIndex
    newMaxIndex = np.amax( newOutIndex, axis=0 )    # New max index.

    # Now creating a new blank image where the pixels will be drawn.
    # The +1 is because if the max col, row index is 639, 479, then image is 640 x 480.
    newW, newH = newMaxIndex[0] + 1, newMaxIndex[1] + 1

    print(newW, newH)
    print( srcW, srcH )

    img = np.zeros( ( newH, newW, 3 ), dtype=np.uint8 )
    for r in range( newH ):
```

```
        for c in range( newW ):
            targetPt = homogeneousFormat( [c + minIndex[0], r + minIndex[1] ] )
            sourcePt = applyHomography( [ np.linalg.inv( H ), targetPt ] )
            x, y = planarFormat( sourcePt )
            if x > 0 and y > 0 and x < srcW and y < srcH:
                img[r][c] = sourceImg[y][x]

    # Resizing for the purpose of display.
    imgH, imgW = img.shape[0], img.shape[1]

    #print(imgH, imgW)

    if imgH < 480 or imgW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    imgShow = cv2.resize( img, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', imgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

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

    # TASK 2.2.2.   Remove affine distortion.

    # Removing affine distortion figure 1.

    #pqrsFig1 = selectPts( filePath=os.path.join( filePath, filename1 ) )
    pqrsFig1 = [[665, 1672], [968, 1560], [961, 2435], [965, 1715], [1137, 1109], [1831, 3007], [1872, 606],
[1140, 2608]]
    #print( 'Points of {}: {}\n'.format( filename1, pqrsFig1 ) )

    #pqrsFig2 = selectPts( filePath=os.path.join( filePath, filename2 ) )
    pqrsFig2 = [[129, 12], [427, 63], [432, 181], [429, 78], [169, 204], [356, 187], [355, 187], [338, 47]]
    #pqrsFig2 = [[129, 13], [428, 63], [431, 182], [430, 73], [170, 205], [359, 188], [359, 188], [339, 47]]
    #print( 'Points of {}: {}\n'.format( filename2, pqrsFig2 ) )

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

    # Converting points to homogeneous coordinates.

    pqrsHomFmt1 = [ homogeneousFormat( pt ) for pt in pqrsFig1 ]
    pqrsHomFmt2 = [ homogeneousFormat( pt ) for pt in pqrsFig2 ]

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

    # Removing affine distortion from figure 1.

    # These are the lines which are supposed to be perpendicular in the undistorted image.
    # Need to find s11, s12, s21, s22. Now s12 = s21 and s22 can be assumed to be 1.
    # So need to find s11 and s12. With two perpendicular lines one equation of
    # containing s11 and s12 can be formed. So four perpendicular lines are needed
    # to have two equations containing s11 and s12. Then they need to be solved.
    line1Fig1Perp = np.cross( pqrsHomFmt1[0], pqrsHomFmt1[1] )
    line2Fig1Perp = np.cross( pqrsHomFmt1[2], pqrsHomFmt1[3] )
    line3Fig1Perp = np.cross( pqrsHomFmt1[4], pqrsHomFmt1[5] )
    line4Fig1Perp = np.cross( pqrsHomFmt1[6], pqrsHomFmt1[7] )

    # Creating the line free of perspective distortion.
    H = Hbetw1OrigTo1Proj
    HinvT = np.linalg.inv( np.transpose( H ) )

    line1Fig1ProjRemovedPerp = np.matmul( HinvT, line1Fig1Perp )
    line2Fig1ProjRemovedPerp = np.matmul( HinvT, line2Fig1Perp )
    line3Fig1ProjRemovedPerp = np.matmul( HinvT, line3Fig1Perp )
    line4Fig1ProjRemovedPerp = np.matmul( HinvT, line4Fig1Perp )
```

```
    # Divide by the 3rd element to make the overall magnitude smaller.
    line1Fig1ProjRemovedPerp /= line1Fig1ProjRemovedPerp[2]
    line2Fig1ProjRemovedPerp /= line2Fig1ProjRemovedPerp[2]
    line3Fig1ProjRemovedPerp /= line3Fig1ProjRemovedPerp[2]
    line4Fig1ProjRemovedPerp /= line4Fig1ProjRemovedPerp[2]

    # Creating the LA and LB matrices. Such that LA * [s11, s12]^T = LB.
    LA, LB = createLAandLB( line1Fig1ProjRemovedPerp, line2Fig1ProjRemovedPerp, \
                      line3Fig1ProjRemovedPerp, line4Fig1ProjRemovedPerp )

    # Finding the s11 and s12.
    LAinv = np.linalg.inv( LA )
    s = np.matmul( LAinv, LB )
    S = [ [ s[0], s[1] ], [ s[1], 1 ] ]
    S = np.array( S )
    print( 'S: ', S )

    # Finding the A matrix to remove the affine distortion.
    U, E, V_T = np.linalg.svd( S )
    E, V = np.diag( E ), np.transpose( V_T )
    D = np.sqrt( E )
    A = np.matmul( np.matmul( V, D ), V_T )
    print( 'A: ', A )

    # Finding the Homography to remove affine distortion.
    Hbetw1OrigTo1Aff = [ [ A[0,0], A[1,0], 0 ], [ A[0,1], A[1,1], 0 ], [ 0, 0, 1 ] ]
    Hbetw1OrigTo1Aff = np.array( Hbetw1OrigTo1Aff )
    Hbetw1OrigTo1Aff = np.linalg.inv( Hbetw1OrigTo1Aff )
    print( 'Hbetw1OrigTo1Aff: \n{}\n'.format( Hbetw1OrigTo1Aff ) )

    Hbetw1OrigTo1 = np.matmul( Hbetw1OrigTo1Aff, Hbetw1OrigTo1Proj )
    print( 'Hbetw1OrigTo1: \n{}\n'.format( Hbetw1OrigTo1 ) )

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

    # Implanting the target into figure 1.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename1 ) )
    srcH, srcW = sourceImg.shape[0], sourceImg.shape[1]
    H = Hbetw1OrigTo1

    # Taking the corner points of the distorted image and mapping them using
    # the homography to know the dimensions of the output image.
    outputIndex = []
    for r in [ 0, srcH ]:
        for c in [ 0, srcW ]:
            inputPt = homogeneousFormat( [c, r] )
            outputPt = applyHomography( ( H, inputPt ) )
            outputPt = planarFormat( outputPt )
            outputIndex.append( outputPt )

    # Now there are points which are mapped to negative coordinates. So finding
    # those out so that those coordinates can be made (0, 0) and then all the
    # pixels can then be added by those values to make all of them positive.
    outputIndex = np.array(outputIndex)
    minIndex = np.amin( outputIndex, axis=0 )   # This may be a -ve value.
    newOutIndex = outputIndex - minIndex
    newMaxIndex = np.amax( newOutIndex, axis=0 )    # New max index.

    # Now creating a new blank image where the pixels will be drawn.
    # The +1 is because if the max col, row index is 639, 479, then image is 640 x 480.
    newW, newH = newMaxIndex[0] + 1, newMaxIndex[1] + 1
```

```
    print(newW, newH)
    print( srcW, srcH )

    img = np.zeros( ( newH, newW, 3 ), dtype=np.uint8 )
    for r in range( newH ):
        for c in range( newW ):
            targetPt = homogeneousFormat( [c + minIndex[0], r + minIndex[1] ] )
            sourcePt = applyHomography( [ np.linalg.inv( H ), targetPt ] )
            x, y = planarFormat( sourcePt )
            if x > 0 and y > 0 and x < srcW and y < srcH:
                img[r][c] = sourceImg[y][x]

    # Resizing for the purpose of display.
    imgH, imgW = img.shape[0], img.shape[1]

    #print(imgH, imgW)

    if imgH < 480 or imgW < 640:    interpolation = cv2.INTER_LINEAR
    else:   interpolation = cv2.INTER_AREA

    imgShow = cv2.resize( img, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', imgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

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

    # Removing affine distortion figure 2.

    # These are the lines which are supposed to be perpendicular in the undistorted image.
    # Need to find s11, s12, s21, s22. Now s12 = s21 and s22 can be assumed to be 1.
    # So need to find s11 and s12. With two perpendicular lines one equation of
    # containing s11 and s12 can be formed. So four perpendicular lines are needed
    # to have two equations containing s11 and s12. Then they need to be solved.
    line1Fig2Perp = np.cross( pqrsHomFmt2[0], pqrsHomFmt2[1] )
    line2Fig2Perp = np.cross( pqrsHomFmt2[2], pqrsHomFmt2[3] )
    line3Fig2Perp = np.cross( pqrsHomFmt2[4], pqrsHomFmt2[5] )
    line4Fig2Perp = np.cross( pqrsHomFmt2[6], pqrsHomFmt2[7] )

    # Creating the line free of perspective distortion.
    H = Hbetw2OrigTo2Proj
    HinvT = np.linalg.inv( np.transpose( H ) )

    line1Fig2ProjRemovedPerp = np.matmul( HinvT, line1Fig2Perp )
    line2Fig2ProjRemovedPerp = np.matmul( HinvT, line2Fig2Perp )
    line3Fig2ProjRemovedPerp = np.matmul( HinvT, line3Fig2Perp )
    line4Fig2ProjRemovedPerp = np.matmul( HinvT, line4Fig2Perp )

    # Divide by the 3rd element to make the overall magnitude smaller.
    line1Fig2ProjRemovedPerp /= line1Fig2ProjRemovedPerp[2]
    line2Fig2ProjRemovedPerp /= line2Fig2ProjRemovedPerp[2]
    line3Fig2ProjRemovedPerp /= line3Fig2ProjRemovedPerp[2]
    line4Fig2ProjRemovedPerp /= line4Fig2ProjRemovedPerp[2]

    # Creating the LA and LB matrices. Such that LA * [s11, s12]^T = LB.
    LA, LB = createLAandLB( line1Fig2ProjRemovedPerp, line2Fig2ProjRemovedPerp, \
                        line3Fig2ProjRemovedPerp, line4Fig2ProjRemovedPerp )

    # Finding the s11 and s12.
    LAinv = np.linalg.inv( LA )
    s = np.matmul( LAinv, LB )
    S = [ [ s[0], s[1] ], [ s[1], 1 ] ]
    S = np.array( S )
    print( 'S: ', S )
```

```
    # Finding the A matrix to remove the affine distortion.
    U, E, V_T = np.linalg.svd( S )
    E, V = np.diag( E ), np.transpose( V_T )
    D = np.sqrt( E )
    A = np.matmul( np.matmul( V, D ), V_T )
    print( 'A: ', A )

    # Finding the Homography to remove affine distortion.
    Hbetw2OrigTo2Aff = [ [ A[0,0], A[0,1], 0 ], [ A[1,0], A[1,1], 0 ], [ 0, 0, 1 ] ]
    Hbetw2OrigTo2Aff = np.array( Hbetw2OrigTo2Aff )
    Hbetw2OrigTo2Aff = np.linalg.inv( Hbetw2OrigTo2Aff )
    print( 'Hbetw2OrigTo2Aff: \n{}\n'.format( Hbetw2OrigTo2Aff ) )

    Hbetw2OrigTo2 = np.matmul( Hbetw2OrigTo2Aff, Hbetw2OrigTo2Proj )
    print( 'Hbetw2OrigTo2: \n{}\n'.format( Hbetw2OrigTo2 ) )

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

    # Implanting the target into figure 2.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename2 ) )
    srcH, srcW = sourceImg.shape[0], sourceImg.shape[1]
    H = Hbetw2OrigTo2

    # Taking the corner points of the distorted image and mapping them using
    # the homography to know the dimensions of the output image.
    outputIndex = []
    for r in [ 0, srcH ]:
        for c in [ 0, srcW ]:
            inputPt = homogeneousFormat( [c, r] )
            outputPt = applyHomography( ( H, inputPt ) )
            outputPt = planarFormat( outputPt )
            outputIndex.append( outputPt )

    # Now there are points which are mapped to negative coordinates. So finding
    # those out so that those coordinates can be made (0, 0) and then all the
    # pixels can then be added by those values to make all of them positive.
    outputIndex = np.array(outputIndex)
    minIndex = np.amin( outputIndex, axis=0 )   # This may be a -ve value.
    newOutIndex = outputIndex - minIndex
    newMaxIndex = np.amax( newOutIndex, axis=0 )    # New max index.

    # Now creating a new blank image where the pixels will be drawn.
    # The +1 is because if the max col, row index is 639, 479, then image is 640 x 480.
    newW, newH = newMaxIndex[0] + 1, newMaxIndex[1] + 1

    print(newW, newH)
    print( srcW, srcH )

    img = np.zeros( ( newH, newW, 3 ), dtype=np.uint8 )
    for r in range( newH ):
        for c in range( newW ):
            targetPt = homogeneousFormat( [c + minIndex[0], r + minIndex[1] ] )
            sourcePt = applyHomography( [ np.linalg.inv( H ), targetPt ] )
            x, y = planarFormat( sourcePt )
            if x > 0 and y > 0 and x < srcW and y < srcH:
                img[r][c] = sourceImg[y][x]

    # Resizing for the purpose of display.
    imgH, imgW = img.shape[0], img.shape[1]

    #print(imgH, imgW)

    if imgH < 480 or imgW < 640:    interpolation = cv2.INTER_LINEAR
```

```
    else:   interpolation = cv2.INTER_AREA

    imgShow = cv2.resize( img, (640,480), interpolation=interpolation )
    cv2.imshow( 'Undistorted Image', imgShow )
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```