ECE 661: Computer Vision

Homework 2, Fall 2018

Arindam Bhanja Chowdhury
abhanjac@purdue.edu

1 Overview

This homework is about finding out the *Homography* between two random images and then map one image onto the other. In other words, transform location of the points of one image to the location of the points of another.

The given images are shown in following Fig 1, 2, 3, 4.

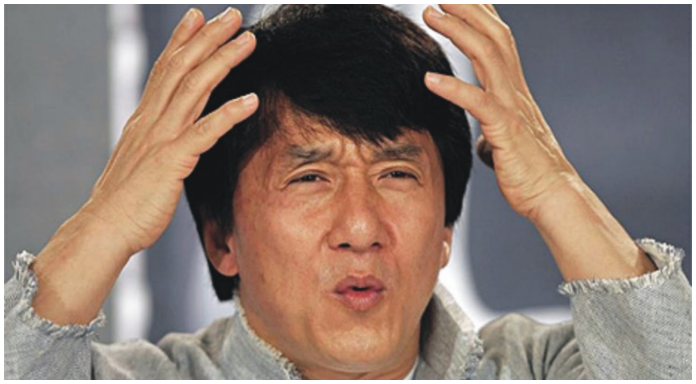

Figure 1: Reference image 1d of homework. Image that has to be mapped inside the painting frame of Fig 2, 3 and 4.

Figure 2: Reference image 1a of homework. The image of Jackie is to be mapped inside the painting frame in this image.

Figure 3: Reference image 1b of homework. The image of Jackie is to be mapped inside the painting frame in this image.

Figure 4: Reference image 1c of homework. The image of Jackie is to be mapped inside the painting frame in this image.

The image of Jackie is to be pasted inside the painting frame shown in Fig 2, 3 and 4.

2 Calculating Homography Matrix

Let $P = [p_x, p_y]^T$ be a point in planar coordinate of the source image and $P_1 = [p_{x1}, p_{y1}]^T$ be the corresponding planar point in the transformed image. Let H be the Homography matrix that does this transformation. And let P_h and P_{h1} are the homogeneous coordinates of the points P and P_1, then the following equation can be written.

$$P_{h1} = HP_h \tag{1}$$

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \tag{2}$$

Now, in the homogeneous coordinate system, only the ratio of the coefficients matter. So any one of the elements of H can be considered as 1. Here, $h_{33} = 1$ is considered.

$$P_h = [p_x, p_y, 1]^T \tag{3}$$

$$P_{h1} = [p'_x, p'_y, p'_z]^T \tag{4}$$

So the P_{h1} can be converted to its planar form P_1 in the following manner.

$$P_1 = [p_{x1}, p_{y1}]^T = \left[\frac{p'_x}{p'_z}, \frac{p'_y}{p'_z}\right]^T \tag{5}$$

Expanding the equation (2) with the assumption of $h_{33} = 1$:

$$\begin{aligned} p'_x &= h_{11}p_x + h_{12}p_y + h_{13} \\ p'_y &= h_{21}p_x + h_{22}p_y + h_{23} \\ p'_z &= h_{31}p_x + h_{32}p_y + 1 \end{aligned} \tag{6}$$

Now, the equations for p_{x1} and p_{y1} can be written with the help of (6) as follows.

$$\begin{aligned} p_{x1} = \frac{p'_x}{p'_z} &= \frac{h_{11}p_x + h_{12}p_y + h_{13}}{h_{31}p_x + h_{32}p_y + 1} \\ p_{y1} = \frac{p'_y}{p'_z} &= \frac{h_{21}p_x + h_{22}p_y + h_{23}}{h_{31}p_x + h_{32}p_y + 1} \end{aligned}$$

which can be simplified as,

$$\begin{aligned} p_{x1} &= h_{11}p_x + h_{12}p_y + h_{13} - h_{31}p_xp_{x1} - h_{32}p_yp_{x1} \\ p_{y1} &= h_{21}p_x + h_{22}p_y + h_{23} - h_{31}p_xp_{y1} - h_{32}p_yp_{y1} \end{aligned} \tag{7}$$

Now, in this task, the points $P = [p_x, p_y]^T$ and $P_1 = [p_{x1}, p_{y1}]^T$ are already available from the given images (those have to be picked up manually). So the h elements of the Homography matrix are the actual unknowns. And there are altogether 8 of them. So, at least 8 equations like those in (7) are needed. For this at least 4 points are needed from the images. So 3 more points Q, R and S are also selected from each of the images. These points will also form the same derivation as above

and the 8 equations obtained are

$$\begin{aligned}
p_{x1} &= h_{11}p_x + h_{12}p_y + h_{13} - h_{31}p_x p_{x1} - h_{32}p_y p_{x1} \\
p_{y1} &= h_{21}p_x + h_{22}p_y + h_{23} - h_{31}p_x p_{y1} - h_{32}p_y p_{y1} \\
q_{x1} &= h_{11}q_x + h_{12}q_y + h_{13} - h_{31}q_x q_{x1} - h_{32}q_y q_{x1} \\
q_{y1} &= h_{21}q_x + h_{22}q_y + h_{23} - h_{31}q_x q_{y1} - h_{32}q_y q_{y1} \\
r_{x1} &= h_{11}r_x + h_{12}r_y + h_{13} - h_{31}r_x r_{x1} - h_{32}r_y r_{x1} \\
r_{y1} &= h_{21}r_x + h_{22}r_y + h_{23} - h_{31}r_x r_{y1} - h_{32}r_y r_{y1} \\
s_{x1} &= h_{11}s_x + h_{12}s_y + h_{13} - h_{31}s_x s_{x1} - h_{32}s_y s_{x1} \\
s_{y1} &= h_{21}s_x + h_{22}s_y + h_{23} - h_{31}s_x s_{y1} - h_{32}s_y s_{y1}
\end{aligned} \tag{8}$$

(8) in matrix form is the following

$$\begin{bmatrix} p_{x1} \\ p_{y1} \\ q_{x1} \\ q_{y1} \\ r_{x1} \\ r_{y1} \\ s_{x1} \\ s_{y1} \end{bmatrix} = \begin{bmatrix} p_x & p_y & 1 & 0 & 0 & 0 & -p_x p_{x1} & -p_y p_{x1} \\ 0 & 0 & 0 & p_x & p_y & 1 & -p_x p_{y1} & -p_y p_{y1} \\ q_x & q_y & 1 & 0 & 0 & 0 & -q_x q_{x1} & -q_y q_{x1} \\ 0 & 0 & 0 & q_x & q_y & 1 & -q_x q_{y1} & -q_y q_{y1} \\ r_x & r_y & 1 & 0 & 0 & 0 & -r_x r_{x1} & -r_y r_{x1} \\ 0 & 0 & 0 & r_x & r_y & 1 & -r_x r_{y1} & -r_y r_{y1} \\ s_x & s_y & 1 & 0 & 0 & 0 & -s_x s_{x1} & -s_y s_{x1} \\ 0 & 0 & 0 & s_x & s_y & 1 & -s_x s_{y1} & -s_y s_{y1} \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} \tag{9}$$

Which can be also represented as

$$b_{8\times1} = A_{8\times8} h_{8\times1} \tag{10}$$

So,

$$h_{8\times1} = A^{-1}_{8\times8} b_{8\times1} \tag{11}$$

From this equation all the h is evaluated and then H can be obtained from the (2).

3 Evaluation of Pixel Values

Now after the points have been mapped from one image to another, some of the pixel coordinates do not map to exact integral coordinates in the second image. Suppose a point G is mapped to a location between the junction of four points C, D, E and F. Then the pixel values of all these points are summed up to form a weighted averaged value, which is assigned as the value for G. The weights are the euclidean distances of the C, D, E and F points from the location of G. The following formula summarizes that.

$$g = \frac{c \times Dist_{xg} + d \times Dist_{dg} + e \times Dist_{eg} + f \times Dist_{fg}}{Dist_{xg} + Dist_{dg} + Dist_{eg} + Dist_{fg}} \tag{12}$$

4 Procedure of Task 1a

- 4 points are chosen from each of the corner of the painting in Fig 2.
- Then the 4 corner points of the Fig 1 is taken.

- Homography between the points of Fig 1 and 2 is calculated, such that this H matrix can transform points from Fig 2 to Fig 1.
- The region inside the painting in Fig 2 is made all black, i.e. the pixels are made all 0 valued.
- Then each of these black pixels are mapped from its original location in Fig 2 to the Fig 1 using H matrix.
- Now the color of this mapped pixel location (on Fig 1) is evaluated using (12).
- Now this color value is assigned to the original pixel location in Fig 2.
- The same procedure is followed for the mapping the Fig 1 to the painting frames inside Fig 3 and 4.

5 Results of Task 1a

Figure 5: Projected Fig 1 into Fig 2.

Figure 6: Projected Fig 1 into Fig 3.

Figure 7: Projected Fig 1 into Fig 3.

6 Procedure of Task 1b

- The Homographies between Fig 2 and 3 and between Fig 3 and 4 are calculated.
- These are multiplied and the resulting matrix is applied to place the image inside the painting in Fig 2 into the painting of Fig 4.

7 Results of Task 1b

Figure 8: Projected Fig 2 into Fig 4.

H by product of to Homographies (applied to image):

$$\begin{bmatrix} 4.96250858 & -1.18237613 \times 10^{-2} & -1.63225444 \times 10^{3} \\ 1.98238062 & 2.91605128 & -3.66189217 \times 10^{3} \\ 1.17327108 \times 10^{-3} & -3.00732284 \times 10^{-5} & 8.92538798 \times 10^{-1} \end{bmatrix} \tag{13}$$

H by direct Homography calculation between Fig 2 and Fig 4 from selected points:

$$\begin{bmatrix} 5.55999200 & -1.32473359 \times 10^{-2} & -1.82877701 \times 10^{3} \\ 2.22105821 & 3.26714232 & -4.10278206 \times 10^{3} \\ 1.31453230 \times 10^{-3} & -3.36940293 \times 10^{-5} & 1.00000000 \end{bmatrix} \tag{14}$$

This shows that the results are pretty close.

8 Procedure of Task 2

- The image of a face to be projected is shown in Fig 9.
- This face image is to be projected onto the brown cardboard region of the Fig 10, 11 and 12.
- The same process is followed to project Fig 9 onto Fig 10, 11 and 12, as done in Section 4.
- The same process is followed to project Fig 11 onto Fig 12 as done in Section 6.

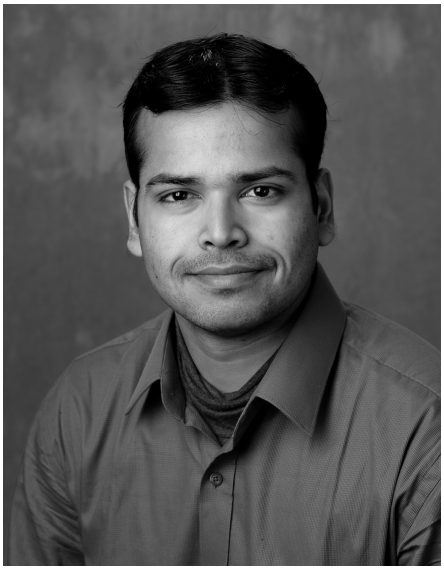

Figure 9: Image that has to be mapped into the brown board region of the images in Fig 10, 11 and 12.

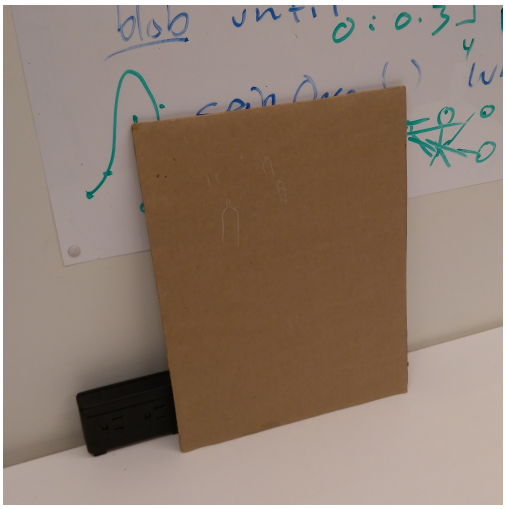

Figure 10: The image of the face is to be mapped inside the brown board region in this image.

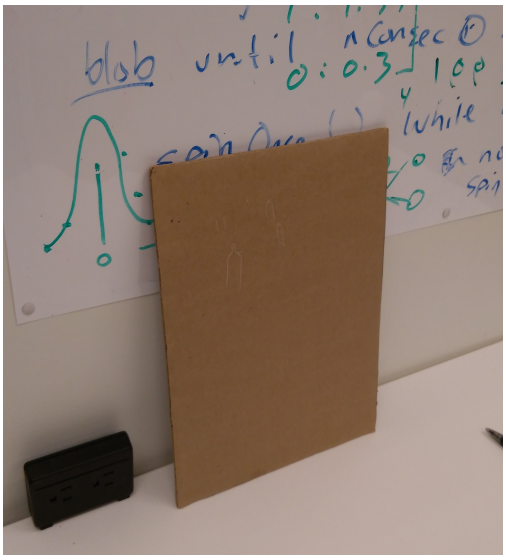

Figure 11: The image of the face is to be mapped inside the brown board region in this image.

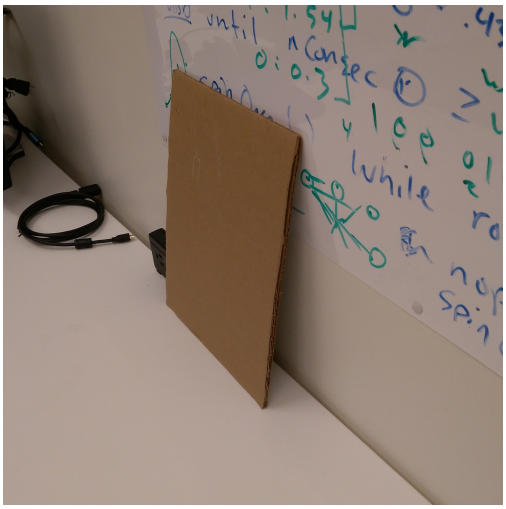

Figure 12: The image of the face is to be mapped inside the brown board region in this image.

9 Results of Task 2

The results after projecting the face are shown in Fig 13, 14 and 15.

Figure 13: Projected Fig 9 into Fig 10.

Figure 14: Projected Fig 9 into Fig 11.

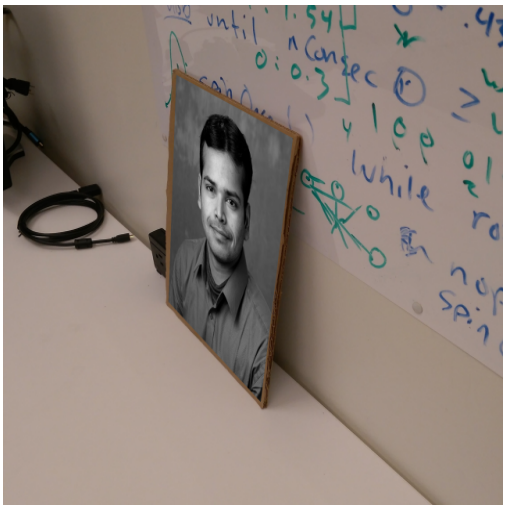

Figure 15: Projected Fig 9 into Fig 12.

The result after projecting the brown board region from Fig 10 to Fig 12 by the H matrix obtained by the product of the Homographies between brown board regions of Fig 10 and 11 and Fig 11 and 12, is shown in Fig 16.

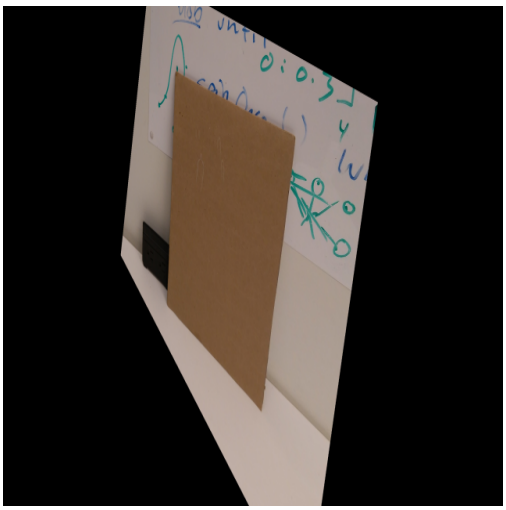

Figure 16: Projected Fig 10 into Fig 12.

H by product of to Homographies (applied to image):

$$\begin{bmatrix} 1.14834654 \times 10^{1} & 1.17602690 & -7.67224643 \times 10^{3} \\ -2.61144893 & 4.59152448 & 2.51312541 \times 10^{3} \\ 2.42230063 \times 10^{-3} & 1.54493845 \times 10^{-4} & 1.23879552 \end{bmatrix} \tag{15}$$

H by direct Homography calculation between Fig 10 and Fig 12 from selected points:

$$\begin{bmatrix} 9.26986355 & 9.49330929 \times 10^{-1} & -6.19331143 \times 10^{3} \\ -2.10805488 & 3.70644260 & 2.02868462 \times 10^{3} \\ 1.95536761 \times 10^{-3} & 1.24712952 \times 10^{-4} & 1.00000000 \end{bmatrix} \tag{16}$$

This also shows that the results are pretty close.

10 Comments

- Mapping Fig 1 to 2 took 54.18sec. Mapping Fig 1 to 3 took 58.79sec. Mapping Fig 1 to 4 took 70.05sec. But it also depends on the other applications running on the cpu.
- Mapping Fig 9 to 10 took 30.42sec. Mapping Fig 9 to 11 took 38.70sec. Mapping Fig 9 to 12 took 18.69sec. The images in the task 2 are smaller, and so the overall processing is faster.
- The modification of the color of the pixels using the (12) in Section 3, makes the process slow.
- In the code a separate function was used to find the pixel values of the points in the images, which may not be used if other tools like GIMP is used.
- In general if an image from picture S (Jackie) is to be projected to picture T (painting), then the Homography (H) to change points from T to S is needed. This H will be applied to each point (p_t) in T to first map them to a location p_s in S, and then the value of the point p_s on S will be copied to the location p_t in T. This way the whole image in S will be projected onto T.
- In the second case, the S was full picture of Fig 2 and T was a blank picture (BP) of the size of the Fig 4 (not the Fig 4 itself). The Homography (H) to change points from T to S is the H that will map points from BP to Fig 2. Each pixel location p_t of the BP (T) will be first multiplied by H to map it to a location p_s on Fig 2 (S) and then this value of the point p_s will be copied to the empty pixels at location p_t on BP (Note that the Homography matrix works on the pixel coordinates, not with the pixel RGB values. So even if the RGB values of the BP are all zeros, that does not mean that the pixel coordinates are zero). This way the Fig 2 is transformed into the posture of Fig 4. The black regions of the resulting figure are the leftover regions of BP where nothing was mapped.
- The task 2 was also performed with the same logic.

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

import numpy as np, cv2, os, time

#==============================================================================
# ARINDAM BHANJA CHOWDHURY
# abhanjac@purdue.edu
# ECE 661 FALL 2018, HW 2
#==============================================================================


# 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:     img1 = 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( img1 )      # Temporary image.
```

```
        # Displaying all the points in listOfPts on the image.
        for i in range( len(listOfPts) ):
            cv2.circle( imgTemp, 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) )

    targetH, targetW = targetImg.shape[0], targetImg.shape[1]
    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.

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

if __name__ == '__main__':

    # TASK 1a.

    filePath = './PicsHw2'
    filename1, filename2, filename3 = '1.jpg', '2.jpg', '3.jpg'
    faceFileName = 'Jackie.jpg'

    # Reading the points.

    #pqrsFig1 = selectPts( filePath=os.path.join( filePath, filename1 ) )
    pqrsFig1 = [[1518, 180], [2940, 728], [2996, 2031], [1495, 2240]]
    #print( 'Points of {}: {}\n'.format( filename1, pqrsFig1 ) )

    #pqrsFig2 = selectPts( filePath=os.path.join( filePath, filename2 ) )
    pqrsFig2 = [[1331, 344], [3002, 626], [3025, 1896], [1309, 2009]]
    #print( 'Points of {}: {}\n'.format( filename2, pqrsFig2 ) )

    #pqrsFig3 = selectPts( filePath=os.path.join( filePath, filename3 ) )
    pqrsFig3 = [[931, 744], [2793, 395], [2855, 2223], [903, 2093]]
    #print( 'Points of {}: {}\n'.format( filename3, pqrsFig3 ) )

    #pqrsFace = selectPts( filePath=os.path.join( filePath, faceFileName ) )
    pqrsFace = [[0, 0], [1280, 0], [1280, 720], [0, 720]]
    #print( 'Points of {}: {}\n'.format( faceFileName, pqrsFace ) )

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

    # Converting points to homogeneous coordinates.

    pqrsHomFmt1 = [ homogeneousFormat( pt ) for pt in pqrsFig1 ]
    pqrsHomFmt2 = [ homogeneousFormat( pt ) for pt in pqrsFig2 ]
    pqrsHomFmt3 = [ homogeneousFormat( pt ) for pt in pqrsFig3 ]
    pqrsHomFmtF = [ homogeneousFormat( pt ) for pt in pqrsFace ]

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

```
    # Finding the homography.

    # Homography between target and fig1.
    Hbetw1ToFace = homography( srcPts=pqrsHomFmt1, dstPts=pqrsHomFmtF )
    #print( 'Homography between {} -> {}: \n{}\n'.format( filename1, \
                                        #faceFileName, Hbetw1ToFace ) )

    # Homography between target and fig2.
    Hbetw2ToFace = homography( srcPts=pqrsHomFmt2, dstPts=pqrsHomFmtF )
    #print( 'Homography between {} -> {}: \n{}\n'.format( filename2, \
                                        #faceFileName, Hbetw2ToFace ) )

    # Homography between target and fig3.
    Hbetw3ToFace = homography( srcPts=pqrsHomFmt3, dstPts=pqrsHomFmtF )
    #print( 'Homography between {} -> {}: \n{}\n'.format( filename3, \
                                        #faceFileName, Hbetw3ToFace ) )

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

    # Implanting the target into figs.

    # NOTE:
    # The target image is the image to be mapped into the source image.
    # This is the way the nomenclature is defined in this code.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, faceFileName ) )
    targetImg = cv2.imread( os.path.join( filePath, filename1 ) )

    targetImg = mapImages( sourceImg=sourceImg, targetImg=targetImg, \
                                        pqrs=pqrsFig1, H=Hbetw1ToFace )

    # Resizing for the purpose of display.
    targetImg = cv2.resize( targetImg, (640,480), interpolation=cv2.INTER_AREA )
    cv2.imshow( 'Mapped Image', targetImg )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, faceFileName ) )
    targetImg = cv2.imread( os.path.join( filePath, filename2 ) )

    targetImg = mapImages( sourceImg=sourceImg, targetImg=targetImg, \
                                        pqrs=pqrsFig2, H=Hbetw2ToFace )

    # Resizing for the purpose of display.
    targetImg = cv2.resize( targetImg, (640,480), interpolation=cv2.INTER_AREA )
    cv2.imshow( 'Mapped Image', targetImg )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, faceFileName ) )
    targetImg = cv2.imread( os.path.join( filePath, filename3 ) )

    targetImg = mapImages( sourceImg=sourceImg, targetImg=targetImg, \
                                        pqrs=pqrsFig3, H=Hbetw3ToFace )

    # Resizing for the purpose of display.
    targetImg = cv2.resize( targetImg, (640,480), interpolation=cv2.INTER_AREA )
    cv2.imshow( 'Mapped Image', targetImg )
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

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

    # TASK 1b.

    # Applying the product of H from 1 to 2 and H from 2 to 3 to the fig1.

    # NOTE:
    # The target image is the image to be mapped into the source image.
    # This is the way the nomenclature is defined in this code.

    Hbetw2To1 = homography( srcPts=pqrsHomFmt2, dstPts=pqrsHomFmt1 )
    Hbetw3To2 = homography( srcPts=pqrsHomFmt3, dstPts=pqrsHomFmt2 )

    Hbetw3To1 = homography( srcPts=pqrsHomFmt3, dstPts=pqrsHomFmt1 )
    print( 'H by direct homography calculation: \n{}'.format( Hbetw3To1 ) )

    Hbetw3To1 = np.matmul( Hbetw2To1, Hbetw3To2 )
    print( 'H by product of to homographies (applied to image): \n{}'.format( \
                                                            Hbetw3To1 ) )
    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename1 ) )
    targetImg = np.zeros( sourceImg.shape )
    tgtH, tgtW = targetImg.shape[0], targetImg.shape[1]
    H = Hbetw3To1

    processingTime = time.time()

    for r in range( tgtH ):
        for c in range( tgtW ):
            targetPt = homogeneousFormat( [ c, r ] )
            targetPt = applyHomography( [H, targetPt] )

            targetPt = planarFormat( targetPt )
            #print( r, c )

            if targetPt[0] > 0 and targetPt[0] < tgtW and \
               targetPt[1] > 0 and targetPt[1] < tgtH:
                targetImg[r][c] = sourceImg[ targetPt[1] ][ targetPt[0] ]

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

    # Resizing for the purpose of display.
    targetImg = np.asarray( targetImg, dtype=np.uint8 )
    targetImg = cv2.resize( targetImg, (640,480), interpolation=cv2.INTER_AREA )
    cv2.imshow( 'Transformed Image', targetImg )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

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

    # TASK 2.

    filePath = './PicsSelf'
    filename1, filename2, filename3 = '1.jpg', '2.jpg', '3.jpg'
    faceFileName = 'self.jpg'

    # Reading the points.

    #pqrsFig1 = selectPts( filePath=os.path.join( filePath, filename1 ) )
    pqrsFig1 = [[616, 584], [1864, 399], [1875, 1777], [853, 2065]]
    #print( 'Points of {}: {}\n'.format( filename1, pqrsFig1 ) )

    #pqrsFig2 = selectPts( filePath=os.path.join( filePath, filename2 ) )
    pqrsFig2 = [[711, 716], [1791, 545], [1714, 1797], [824, 2104]]
    #print( 'Points of {}: {}\n'.format( filename2, pqrsFig2 ) )
```

```
    #pqrsFig3 = selectPts( filePath=os.path.join( filePath, filename3 ) )
    pqrsFig3 = [[809, 326], [1353, 623], [1203, 1855], [773, 1388]]
    #print( 'Points of {}: {}\n'.format( filename3, pqrsFig3 ) )

    #pqrsFace = selectPts( filePath=os.path.join( filePath, faceFileName ) )
    pqrsFace = [[5, 15], [1188, 12], [1186, 1478], [9, 1481]]
    #print( 'Points of {}: {}\n'.format( faceFileName, pqrsFace ) )

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

    # Converting points to homogeneous coordinates.

    pqrsHomFmt1 = [ homogeneousFormat( pt ) for pt in pqrsFig1 ]
    pqrsHomFmt2 = [ homogeneousFormat( pt ) for pt in pqrsFig2 ]
    pqrsHomFmt3 = [ homogeneousFormat( pt ) for pt in pqrsFig3 ]
    pqrsHomFmtF = [ homogeneousFormat( pt ) for pt in pqrsFace ]

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

    # Finding the homography.

    # Homography between target and fig1.
    Hbetw1ToFace = homography( srcPts=pqrsHomFmt1, dstPts=pqrsHomFmtF )
    #print( 'Homography between {} -> {}: \n{}\n'.format( filename1, \
                                        #faceFileName, Hbetw1ToFace ) )

    # Homography between target and fig2.
    Hbetw2ToFace = homography( srcPts=pqrsHomFmt2, dstPts=pqrsHomFmtF )
    #print( 'Homography between {} -> {}: \n{}\n'.format( filename2, \
                                        #faceFileName, Hbetw2ToFace ) )

    # Homography between target and fig3.
    Hbetw3ToFace = homography( srcPts=pqrsHomFmt3, dstPts=pqrsHomFmtF )
    #print( 'Homography between {} -> {}: \n{}\n'.format( filename3, \
                                        #faceFileName, Hbetw3ToFace ) )

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

    # Implanting the target into figs.

    # NOTE:
    # The target image is the image to be mapped into the source image.
    # This is the way the nomenclature is defined in this code.

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, faceFileName ) )
    targetImg = cv2.imread( os.path.join( filePath, filename1 ) )

    targetImg = mapImages( sourceImg=sourceImg, targetImg=targetImg, \
                                        pqrs=pqrsFig1, H=Hbetw1ToFace )

    # Resizing for the purpose of display.
    targetImg = cv2.resize( targetImg, (640,480), interpolation=cv2.INTER_AREA )
    cv2.imshow( 'Mapped Image', targetImg )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, faceFileName ) )
    targetImg = cv2.imread( os.path.join( filePath, filename2 ) )

    targetImg = mapImages( sourceImg=sourceImg, targetImg=targetImg, \
                                        pqrs=pqrsFig2, H=Hbetw2ToFace )
```

```
    # Resizing for the purpose of display.
    targetImg = cv2.resize( targetImg, (640,480), interpolation=cv2.INTER_AREA )
    cv2.imshow( 'Mapped Image', targetImg )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, faceFileName ) )
    targetImg = cv2.imread( os.path.join( filePath, filename3 ) )

    targetImg = mapImages( sourceImg=sourceImg, targetImg=targetImg, \
                                        pqrs=pqrsFig3, H=Hbetw3ToFace )

    # Resizing for the purpose of display.
    targetImg = cv2.resize( targetImg, (640,480), interpolation=cv2.INTER_AREA )
    cv2.imshow( 'Mapped Image', targetImg )
    cv2.waitKey(0)
    cv2.destroyAllWindows()

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

    # Applying the product of H from 1 to 2 and H from 2 to 3 to the fig1.

    # NOTE:
    # The target image is the image to be mapped into the source image.
    # This is the way the nomenclature is defined in this code.

    Hbetw2To1 = homography( srcPts=pqrsHomFmt2, dstPts=pqrsHomFmt1 )
    Hbetw3To2 = homography( srcPts=pqrsHomFmt3, dstPts=pqrsHomFmt2 )

    Hbetw3To1 = homography( srcPts=pqrsHomFmt3, dstPts=pqrsHomFmt1 )
    print( 'H by direct homography calculation: \n{}'.format( Hbetw3To1 ) )

    Hbetw3To1 = np.matmul( Hbetw2To1, Hbetw3To2 )
    print( 'H by product of to homographies (applied to image): \n{}'.format( \
                                                            Hbetw3To1 ) )
    # Reading the images.
    sourceImg = cv2.imread( os.path.join( filePath, filename1 ) )
    targetImg = np.zeros( sourceImg.shape )
    tgtH, tgtW = targetImg.shape[0], targetImg.shape[1]
    H = Hbetw3To1

    processingTime = time.time()

    for r in range( tgtH ):
        for c in range( tgtW ):
            targetPt = homogeneousFormat( [ c, r ] )
            targetPt = applyHomography( [H, targetPt] )

            targetPt = planarFormat( targetPt )
            print( r, c )

            if targetPt[0] > 0 and targetPt[0] < tgtW and \
               targetPt[1] > 0 and targetPt[1] < tgtH:
                targetImg[r][c] = sourceImg[ targetPt[1] ][ targetPt[0] ]

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

    # Resizing for the purpose of display.
    targetImg = np.asarray( targetImg, dtype=np.uint8 )
    targetImg = cv2.resize( targetImg, (640,480), interpolation=cv2.INTER_AREA )
    cv2.imshow( 'Transformed Image', targetImg )
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```