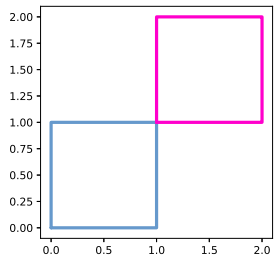
Assignment A02: Alignment

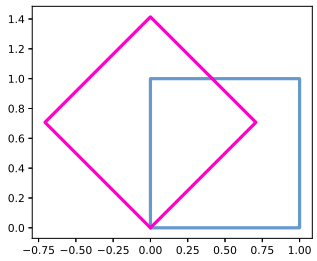
Thalagala B.P. 180631J

1 2-D transformations

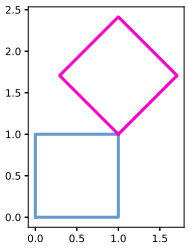
As shown in the figures (a),(b) and (c) Translation and Rotation preserves angles, area and the lengths of the 2D object subjected to the transformation. They also known as *2D Euclidean transformation* due to the preservation of the Euclidean distances under the transformation. The similarity transform(*Scaled rotation*) only preserves the angles between lines and the Affine transformation only preserves the parallelism of the lines while the Projective transformation distort all the mentioned properties and preserves only the straightness of the lines of the object.



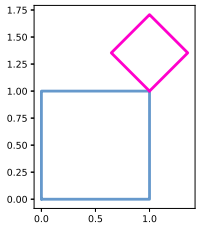
(a) Translation



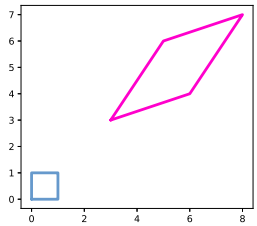
(b) Rotation



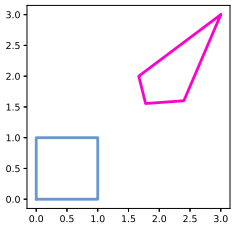
(c) Rotation + Translation



(d) Similarity Transform



(e) Affine



(f) Projective

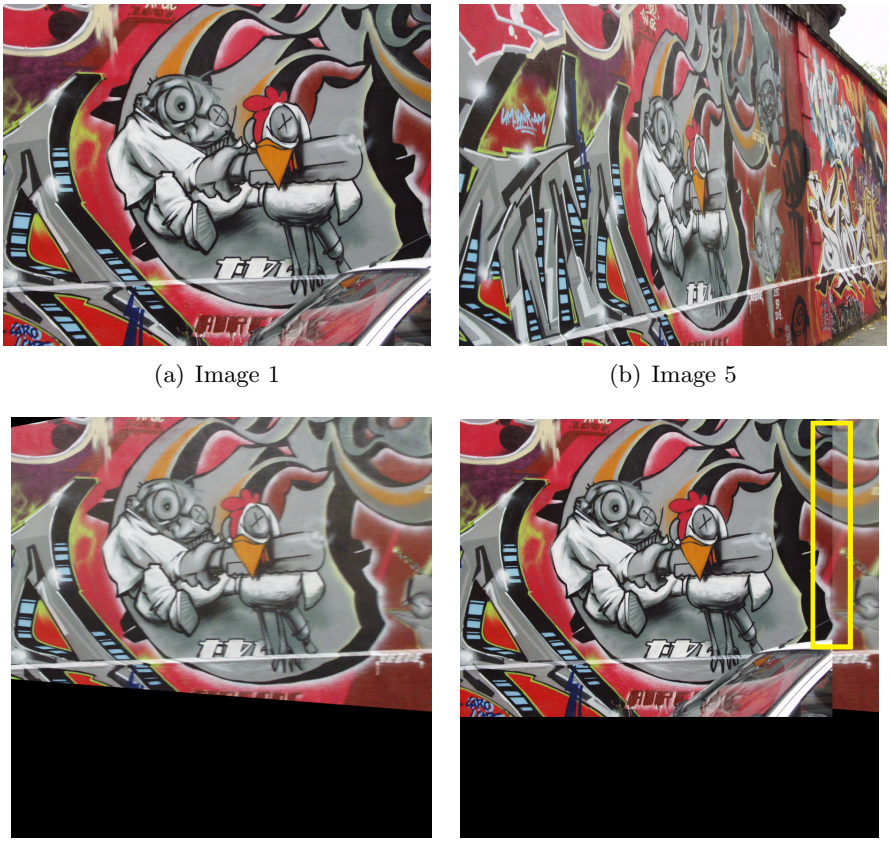
Figure 1: 2-D transformations *Note that: Figures are not in the same scale.*

Following code snippet consists of all the transformation matrices used to generate the above figures in the given order.

[1]:
```
# Translation
t = 1 # translation along each axis
H = [[1,0,t],[0,1,t],[0,0,1]]
# Rotation
theta = np.pi/4 # Anti clockwise pi/4 rad rotation
H = [[np.cos(theta), -np.sin(theta), 0.], [np.sin(theta), np.cos(theta), 0.], [0., 0., 1.]]
# Rotation + translation.(2D Euclidean transformation)
H = [[np.cos(theta), -np.sin(theta), t], [np.sin(theta), np.cos(theta), t], [0., 0., 1.]]
# Scaled rotation (similarity transform)
s = 0.5
H = [[s*np.cos(theta), -s*np.sin(theta), t], [s*np.sin(theta), s*np.cos(theta), t], [0., 0.
 ↪, 1.]]
# Affine Transformation: An arbitrary 2 by 3 Matrix
a00 = 3 ; a01 = 2 ; a02 = 3
a10 = 1 ; a11 = 3 ; a12 = 3
H = [[a00, a01, a02],[a10, a11, a12],[0, 0, 1]]
# Projective(Perspective transform or Homography): An arbitrary 3 by 3 Matrix
a20 = 1.5 ; a21 = 2; a22 = 1
H = [[a00, a01, a02],[a10, a11, a12],[a20,a21,a22]]
```

2 Warping Using a Given Homography

Following figure shows the transformation of the Graffiti `img1.ppm` onto `img5.ppm` using the provided homography matrix. Consider the area enclosed using the yellow colored bounding box in the figure(d). The transition between two images is almost unnoticeable and it will be completely seamless if intensities of the pixels around the transition area are perfectly matched.



(a) Image 1 (b) Image 5

(c) Warped Image 5 (d) Stitched Images

Figure 2: Warping Using a Given Homography

$$H1to5p = \begin{bmatrix} 6.2544644e-01 & 5.7759174e-02 & 2.2201217e+02 \\ 2.2240536e-01 & 1.1652147e+00 & -2.5605611e+01 \\ 4.9212545e-04 & -3.6542424e-05 & 1.0000000e+00 \end{bmatrix}$$

[2]:
```
# Loading im1.ppm and im5.ppm
im1 = cv.imread('../images/graf/img1.ppm', cv.IMREAD_ANYCOLOR)
im5 = cv.imread('../images/graf/img5.ppm', cv.IMREAD_ANYCOLOR)
# Loading the given Homography H1to5p
with open('../images/graf/H1to5p') as f:
    H = [[float(x) for x in line.split()] for line in f]
H = np.array(H)
im5_warped = cv.warpPerspective(im5, np.linalg.inv(H), (900,900))
im5_warped[0:im1.shape[0], 0:im1.shape[1]] = im1
```

---

3 Computing the Homography Using Mouse-Clicked Points and Warping

To calculate the homography at least 4 corresponding points are required. Following figures show the points used to calculate the homography matrix given below. There, 5 points marked in the light blue circles on each image were selected. Note that the same sets of points were used to calculate the homography using the custom function defined in the next section since it enables better comparison between the sections of this report.

(a) Image 1 (b) Image 4

Figure 3: Corresponding Points used to Calculate the Homography

Consider the area enclosed using the yellow colored bounding box in the sub figure(b) of Fig.4. Transition between two images is visible than that in the Fig.2. Human made error at the selection of corresponding points and low quality of the chosen points can be reasons for this imperfection as it was done through manual mouse clicking. However some of the values obtained for the elements of homography matrix is nearly closer to that of the original homography given in the aforementioned website.

(a) Warped Image 4 (b) Stitched Images

Figure 4: Warping Using the Homography calculated using `cv.findHomography()` in OpenCV

The homography matrix calculated using the above points and the OpenCV's `cv.findHomography()` function is given below.

$$H = \begin{bmatrix} 6.59894211e-01 & 6.86220378e-01 & -3.13000247e+01 \\ -1.50977725e-01 & 9.56863356e-01 & 1.52906596e+02 \\ 4.12755346e-04 & -1.98886486e-05 & 1.00000000e+00 \end{bmatrix}$$

[3]:
```
H, status = cv.findHomography(p1,p2)
H = np.array(H)
im4_warped = cv.warpPerspective(im4, np.linalg.inv(H), (900,900))
im4_warped[0:im1.shape[0], 0:im1.shape[1]] = im1
```

---

4 Computing the Homogrpahy Using Mouse-Clicked Points and without OpenCV

Following algorithm implements the ***Normalized Direct Linear Transformation (DLT)*** method (described in the *Multiple View Geometry in Computer Vision*(Second Edition), by *Richard Hartley* and *Andrew Zisserman*), to find the homography matrix $\mathtt{M}_{3\times3}$ using 5 pairs of corresponding points. Let $x_i = [x, y, 1]$ and $x_i' = [x\prime, y\prime, w\prime]$ be

two corresponding points in the given two images. Then the transformation is given by $x_i' = Hx_i$. This equation can be represented by using the vector cross product as $x_i' \times Hx_i = 0$ sine both the components have the same direction even though they differ in magnitude. This equation can be further simplified to obtain the following linearly independent system of two equations corresponding to each pair of points. Where h^j indicates the j^{th} row of the Homography matrix and $j = 1, 2, 3$.

$$\begin{bmatrix} 0^\top & -w\prime x_i^\top & y\prime x_i^\top \\ w\prime x_i^\top & 0^\top & -x\prime x_i^\top \end{bmatrix} \begin{bmatrix} h^1 \\ h^2 \\ h^3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & -w\prime.x & -w\prime.y & -w\prime.1 & y\prime.x & y\prime.y & y\prime.1 \\ w\prime.x & w\prime.y & w\prime.1 & 0 & 0 & 0 & -x\prime.x & -x\prime.y & -x\prime.1 \end{bmatrix} \begin{bmatrix} h^1 \\ h^2 \\ h^3 \end{bmatrix}$$

We can obtain 5 pairs of such equations and they can be put into a single matrix to solve for the unknown h^js through ***Singular Value Decomposition*** as described in the `Algorithm 4.1, 4.2` in the above mentioned reference book. When Considering the area enclosed using the yellow colored bounding box in the sub figure(b) of Fig.5, discontinuous transition between two images is clearly visible than both the Fig.2 and the Fig.4. In addition to the aforementioned reasons for this imperfection, low quality of the used algorithm may also be a reason since it is a very basic algorithm for homography calculation.

(a) Warped Image 4 (b) Stitched Images

Figure 5: Warping Using the Calculated Homography

The homography calculated using the same points mentioned earlier and the `calcHomography()` custom function defined below is given below. Even though some values of its elements are nearly closer to the ideal values it has unable to provide good results due to the previously mentioned reasons.

$$H = \begin{bmatrix} 6.60355290e-01 & 6.85735356e-01 & -3.13168167e+01 \\ -1.50699933e-01 & 9.56965356e-01 & 1.52763838e+02 \\ 4.13108386e-04 & -2.02092511e-05 & 1.00000000e+00 \end{bmatrix}$$

[4]:
```
#============================ Normalization ======================
def normalizePoints(points):
    """
       Normalizing the point cloud(pre-conditioning)
    """
    # Calculating the centroid of the points.
    centroid = sum(points)/len(points)
    # Calculating the scaling factor.
    total_dist = sum(np.sqrt(np.sum(((points - centroid)**2),axis = 1)))
    avg_dist  = total_dist/len(points)
    # For average distance from the origin to be sqrt(2) after scaling
    scale = np.sqrt(2)/avg_dist
    # Defining Similarity transformation: translation and scaling
    xt, yt = centroid
    transform = np.array([[scale, 0, -xt*scale],
                          [0, scale, -yt*scale],
                          [0, 0,1]])
    # Making the points homogeneous by adding 1 at the end
    points = np.concatenate((points, np.ones((len(points),1))), axis =1)
```

```
    # Similarity transformation through matrix multiplication
    normalized_points = transform.dot(points.T).T
    return transform, normalized_points
```

Normalization of data points was also done using the above function in order to improve the performance of the algorithm defined below and to obtain a better homography.

[4]:
```
#==================== Calculating homography ====================
def calcHomography(p1,p2):
    """
        The normalized DLT for 2D homographies.Given in the
        "Multiple View Geometry in Computer Vision" Second Edition
        by Richard Hartley & Andrew Zisserman. Algorithm 4.2
    """
    # Normalizing the points using predefined function
    T1,p1 = normalizePoints(p1)
    T2,p2 = normalizePoints(p2)
    #Initialising an array to keep the coefficint matrix
    A = np.zeros((2*len(p1), 9))
    row = 0
    # Filling rows of the matrix according to the expressions
    for point1, point2 in zip(p1,p2):
        # Coefficients of the current row
        A[row, 3:6] = -point2[2]*point1
        A[row, 6:9] =  point2[1]*point1
        # Coefficients of the next row
        A[row+1, 0:3] =  point2[2]*point1
        A[row+1, 6:9] = -point2[0]*point1
        row+=2
    # Singular Value decomposition of A
    U, D, VT = np.linalg.svd(A)
    # unit singular vector corresponding to the smallest
    # singular value, is the solution h. That is last column of V.
    # i.e. Last row of the V^T
    h =   VT[-1]
    # Reshaping to get 3x3 homography
    H = h.reshape((3,3))
    # Denormalization
    H = np.linalg.inv(T2).dot(H).dot(T1)
    H = H/H[-1,-1]
    return H

# Calculating Homography using the above function
myH = calcHomography(p1,p2)
myH = np.array(H)
# Warping
im4_warped = cv.warpPerspective(im4, np.linalg.inv(myH), (900,900))
# Stiching two images
im4_warped[0:im1.shape[0], 0:im1.shape[1]] = im1
```

Executable code for this assignment can be found here.