ECE 661 Computer Vision
Homework 3

Vineeth Ravi

September 12, 2018

PU ID : 0030019456

EMAIL ID : ravi24@purdue.edu

Theory and Mathematical Background

In the First Section, we present the Theory and Mathematical Equations required for solving the Homework. To eliminate the projective and the affine distortions, we have three methods at our disposal:

1. Using point-to-point correspondences (in exactly the same manner as we did in the previous homework) to find a homography between two images, assuming that one represents the original scene and the other its photograph with projective and affine distortion and then reversing the homography to eliminate the distortion in the latter image.

2. We use what is known as the 2-Step method in which we first remove the projective distortion using the Vanishing Line method discussed in Lecture 4. Subsequently, we remove the affine distortion by using the Cosθ expression with θ equal to 90 degrees. We must first remove the projective distortion before we can remove the affine distortion with the Cosθ based method.

3. We use what is known as the 1-Step method that gets rid of both the projective and the affine distortion in one go.

The above 3 methods are described in detail below:-

1 Point-to-Point Correspondence Method

In this method , we use Point-to-Point Correspondence to estimate the Homography H, to remove distortion from images. We use the height and width of planar objects in the scene, and use them to find the corresponding points in the undistorted image. We now apply the Homography to remove distortion from images. This is a straight-forward method, but in practice requires a large number of correspondences to give a numerically stable solution for estimating the Homography.

$$X_i = HX_w$$

This is the result we used from HW2 , for mapping points from the World Plane to the image Plane. For this HW 3, All the Homography Matrices H computed, refer to the inverse of the Matrix computed in HW2, i.e. :

$$X_w = HX_i$$

X_i is the set of points in the image plane, in homogeneous coordinates, X_w is the set of point points in the world plane, in homogeneous coordinates. H is the Homogrpahy Matrix. We remove both Projective and Affine distortion using this mapping H obtained.

The transformation of a point from 1 plane to another plane can be described by the mathematical equation:-

$$Xw = HX_i$$

where the matrix H is given by:-

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

Let $X_w = [x', y', z']$ and $X_i = [x, y, z]^T$

Then we have the following set of equations :-

$$\begin{aligned} x'_1 &= h_{11}x_1 + h_{12}x_2 + h_{13}x_3 \\ x'_2 &= h_{21}x_1 + h_{22}x_2 + h_{23}x_3 \\ x'_3 &= h_{31}x_1 + h_{32}x_2 + h_{33}x_3 \end{aligned}$$

The point in the physical image plane is given by (x , y) and a point in the physical world plane is given by (x' , y'). So, we have :-

$x = \frac{x_1}{x_3}$, $y = \frac{x_2}{x_3}$

$x' = \frac{x'_1}{x'_3}$, $y' = \frac{x'_2}{x'_3}$

Combining the mathematical equations above we get :-

$$x' = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}}$$

$$y' = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}}$$

We set $h_{33} = 1$, Since only the ratios matter in the matrix. Hence we are left with :-

$x' = h_{11}x + h_{12}y + h_{13} - h_{31}xx' - h_{32}yx'$

$y' = h_{21}x + h_{22}y + h_{23} - h_{31}xy' - h_{32}yy'$

We have 8 unknowns from 2 equations, So we require a minimum of 4 points to find all the unknowns. We write the equations in the form of a matrix :-

$$AH = b$$

$$\text{where A} = \begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -y_1x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & -y_1y'_1 \\ \vdots \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x'_4 & -y_4x'_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y'_4 & -y_4y'_4 \end{bmatrix}$$

$$, \text{H} = \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix}$$

$$\text{and b} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{bmatrix}$$

Solving this matrix equation using pinv (pseudo-inverse) for the general case or (inverse) for the 4 points case we get

$$H = A^{-1}b.$$

Hence we have found the Homography matrix H between the 2 planes (the world plane and the image plane).

Once H is found, we can perform the mapping from the image plane to the world plane.

Programming Notes:-
a) The homography Matrix H calculated satisfies $X_w = HX_i$.
b) The information of World Coordinates is given in Centimeters. So, I assume one pixel is equal to one centimeter.
c) We use 4 corresponding point pairs, to estimate the Homography H.
d) Similar to the reasoning in HW2, we compute H^{-1}, to map the points in the world plane to the image plane and use weighted average method for computing Pixel Values.
e) This part of computing the Homography Matrix H, is used in both the Two-Step Method and One-Step Method.

2 Two-Step Method

The second method of removing the distortions is the Two-Step Method. This involves removing Projective Distortion first, then removing Affine Distortion. Regarding the implementation of the Vanishing Line (VL) method for removing the projective distortion, we will have to estimate the VL in the image plane. For that we'll need to click on the pixels that fall on lines that are parallel in the original scene. Taking the cross-product of two such pixels on any one line in the image will give you the homogeneous representation of that line. Taking the cross-product of 3-vectors for two different lines (which are parallel in the original scene) will give you the homogeneous representation for the Vanishing Point for those two lines. And then taking the cross-product of two such vanishing points for two different pairs of parallel lines will give us the VL you need for getting rid of the projective distortion. The steps involved are described in detail below :-

2.1 Removing Protective Distortion - Vanishing Line Method

We use the concept of Vanishing Lines and l_∞ , to remove Projective Distortion. We compute the Homography that maps the vanishing line back to l_∞ .

We compute the vanishing line in an image, by finding two Vanishing Points and joining them. We can find 2 vanishing points, by taking two different sets of Parallel lines in the World Plane, and computing their point of intersection in the Image Plane.

Suppose if L_1, M_1 and L_2, M_2 are two parallel line pairs then the vanishing points are :-

$$P_1 = L_1 \times M_1$$
$$P_2 = L_2 \times M_2$$

Then the Vanishing Line is given by $L = P_1 \times P_2$.
Suppose if $L = [l_1, l_2, l_3]^T$, Then the Homography that maps the Vanishing line back to l_∞ is given by:-

$$H_p = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l_1 & l_2 & l_3 \end{bmatrix}$$

When we apply this Homography, we can get rid of the projective distortion in our image.

Programming Notes:-
a) The Homography Matrix H_p is given by X_w=H_p X_i. This is important, when we implement the Mapping Direction.
b) We normalize the Vanishing Points and Vanishing Line, before inserting the values into the Matrix H_p.
c) Once we compute H_p, we can now find the mapping between World Plane and Image Plane, similar to the First Method in HW3.
d) Similar to the reasoning in HW2, we compute H^{-1}, to map the points in the world plane to the image plane and use weighted average method for computing Pixel Values.

2.2 Removing Affine Distortion

Affine Distortion causes angles to not be preserved in the image, and causes unequal scaling in orthogonal directions. We use two physically orthogonal lines to remove affine distortion. If we have two orthogonal lines $L = [l_1, l_2, l_3]^T$ and $M = [m_1, m_2, m_3]^T$, then the angle between these 2 lines is given by:

$$cos\theta = \frac{L^T C^*_\infty M}{\sqrt{(L^T C^*_\infty L)(M^T C^*_\infty M)}}$$

where

$$C^*_\infty = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Now if there is a Homography H, which transforms the World Plane to the Image Plane and is denoted by H_a. The lines are now mapped by $L = H^T L'$ and $M = H^T M'$. The conic is mapped by $C_\infty^{*'} = H^T C_\infty^* H$. Since the lines will be orthogonal in the World Plane we have $cos\theta = 0$. Hence :-

$$L'^T H_a C_\infty^* H_a^T M' = 0$$

where

$$H_a = \begin{bmatrix} A & 0 \\ 0 & 1 \end{bmatrix}$$

Expanding the above equation we get:

$$\begin{bmatrix} l'_1 & l'_2 & l'_3 \end{bmatrix} \begin{bmatrix} AA^T & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} m'_1 \\ m'_2 \\ m'_3 \end{bmatrix} = 0$$

Denoting $S = AA^T = \begin{bmatrix} s_{11} & s_{12} \\ s_{12} & s_{22} \end{bmatrix}$

We get :-

$$s_{11} m'_1 l'_1 + s_{12}(l'_1 m'_2 + l'_2 m'_1) + s_{22} l'_2 m'_2 = 0$$

We set $s_{22} = 1$, Since only the ratios matter. We require two sets of orthogonal line pairs to solve for S. We must ensure that these set of orthogonal lines are not repetitive , but actually different sets from each other, like for example the Adjacent sides and diagonals for a square. Once we obtain S, we compute the SVD(S) the singular value decomposition to get A.

$$S = AA^T = VD^2V^T \ , \ A = VDV^T$$

Once we compute A, we can find H_a, and compute the Affine Homography Matrix. Hence we have eliminated the affine distortion.

Programming Notes:-

a) H_p is the matrix which removes Projective Distortion.
b) H_a is the matrix which removes Affine Distortion.
c) We multiply $H_a^{-1} \times H_p$ to get the resultant Homography Matrix which removes both distortions from the original image directly. This is because H_a is a mapping from the world plane to the image plane.
d) We compute the coordinates of the diagonal lines in the image square, once we remove the Projective Distortion, by Applying the Homography H_p on the original image square points in the code itself. e) We need to estimate the mapping direction properly in implementation.
f) The points for finding H_a are found from the points of the image, after removing Projective Distortion.
g) Similar to the reasoning in HW2, we compute H^{-1}, to map the points in the world plane to the image plane and use weighted average method for computing Pixel Values.
h) Points must be selected appropriately, Else results obtained might not be excellent.

3 One-Step Method

The Mathematical equations for this method are similar to that used in Method 2.
In this method we estimate the camera image of $C_\infty^{*'}$ of the Dual Degenerate Conic C_∞^* . From the parameters of the conic images, you estimate the homography that gets rid of both the projective and the affine distortion in one go. We use the concept of dual conics to estimate the Matrix which removes both the projective and affine distortions in a single step. We obtain the general homography :-

$$H = \begin{bmatrix} A & 0 \\ v^T & 1 \end{bmatrix}$$

The dual conic in the image plane is given by:-

$$C_\infty^* = H C_\infty H^T$$

$$C_\infty^* = \begin{bmatrix} AA^T & Av \\ v^T A^T & v^T v \end{bmatrix}$$

$$C_\infty^* = \begin{bmatrix} a & b/2 & c/2 \\ b/2 & c & e/2 \\ d/2 & e/2 & f \end{bmatrix}$$

When we have two lines in the image plane, which are orthogonal to each other in the world plane, i.e. $L = [l'_1, l'_2, l'_3]$ and $M = [m'_1, m'_2, m'_3]^T$, then :-

$$L^{T'} C^{*'}_{\infty} M' = 0 = [l'_1, l'_2, l'_3] \begin{bmatrix} a & b/2 & c/2 \\ b/2 & c & e/2 \\ d/2 & e/2 & f \end{bmatrix} [m'_1, m'_2, m'_3]^T$$

We can set f = 1 , since only the ratios matter in Homographies. We have 5 variables, so we need a minimum 5 sets of equations (orthogonal lines) to solve for them. Once we solve them, we need to normalize them, before applying SVD on the matrix S.

$$S = AA^T =$$

$$\begin{bmatrix} a & b/2 \\ b/2 & c \end{bmatrix}$$

Now we apply SVD to get the Matrix A, Similar to the second method.
We solve for the vector v using the equation $A \times v = [d/2, e/2]^T$.

Hence we have found the homography which gets rid of the projective and affine distortion in 1 step.

Programming Notes:-
a) The homography is given by $X_i = HX_w$, So again we would need compute the inverse, when finding the mapping from the image plane to world plane. So, we need to estimate the mapping direction appropriately.
b) Normalize all the lines, before computing H.
c) Normalize the $C^{*'}_{\infty}$ Matrix , before applying the SVD.
d) Similar to the reasoning in HW2, we compute H^{-1}, to map the points in the world plane to the image plane and use weighted average method for computing Pixel Values.
e) Points must be selected appropriately, Else results obtained might not be excellent.
f) Ensure the atleast 1 set of orthogonal lines are completely different, from the rest, to remove affine distortion.

Tasks and Results

The Red lines represent orthogonal and parallel lines taken from the rectangle. The orthogonal lines are adjacent sides in the rectangle (4 sets can be formed). The parallel lines are opposite sides in the rectangle (2 sets can be formed). The Yellow lines represent Diagonal Lines take from the square. (1 pair can be formed). The opposite red lines and green lines, represent the parallel lines used in computing the affine transform. The adjacent red lines represent the orthogonal lines used in computing the one-step method. One centimeter = 1 pixel is assumed for the point-to-point correspondence method in estimating homography. The results obtained for the Given Image Inputs and Custom Image Inputs are given below :-

Given Image Inputs

The image inputs and results are shown in the following manner for all results:-

Input Image
Results from Method 1
Results from Method 2
Results from Method 3

Results for **Given input image 1**

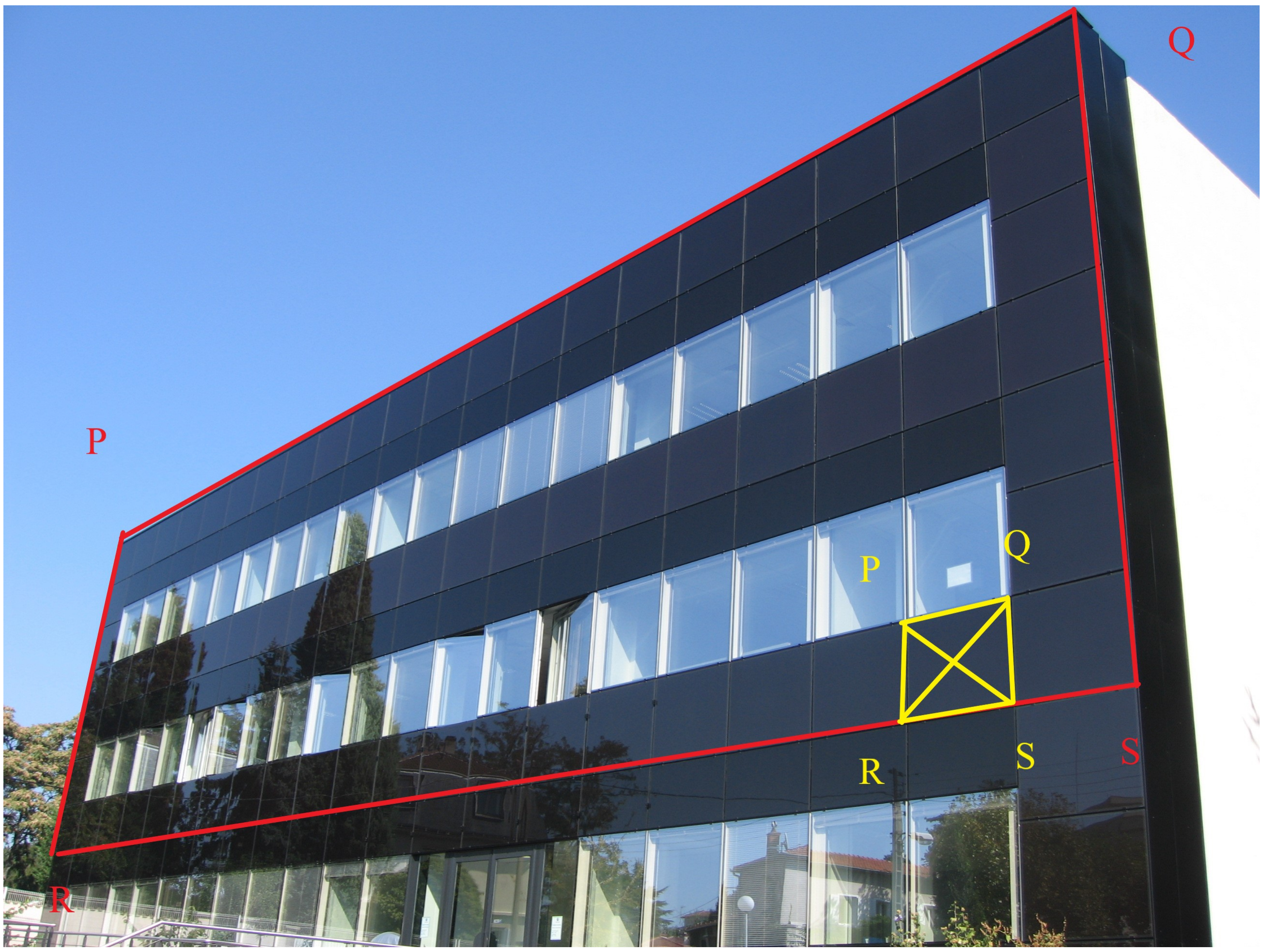

Figure 1: Input Image 1

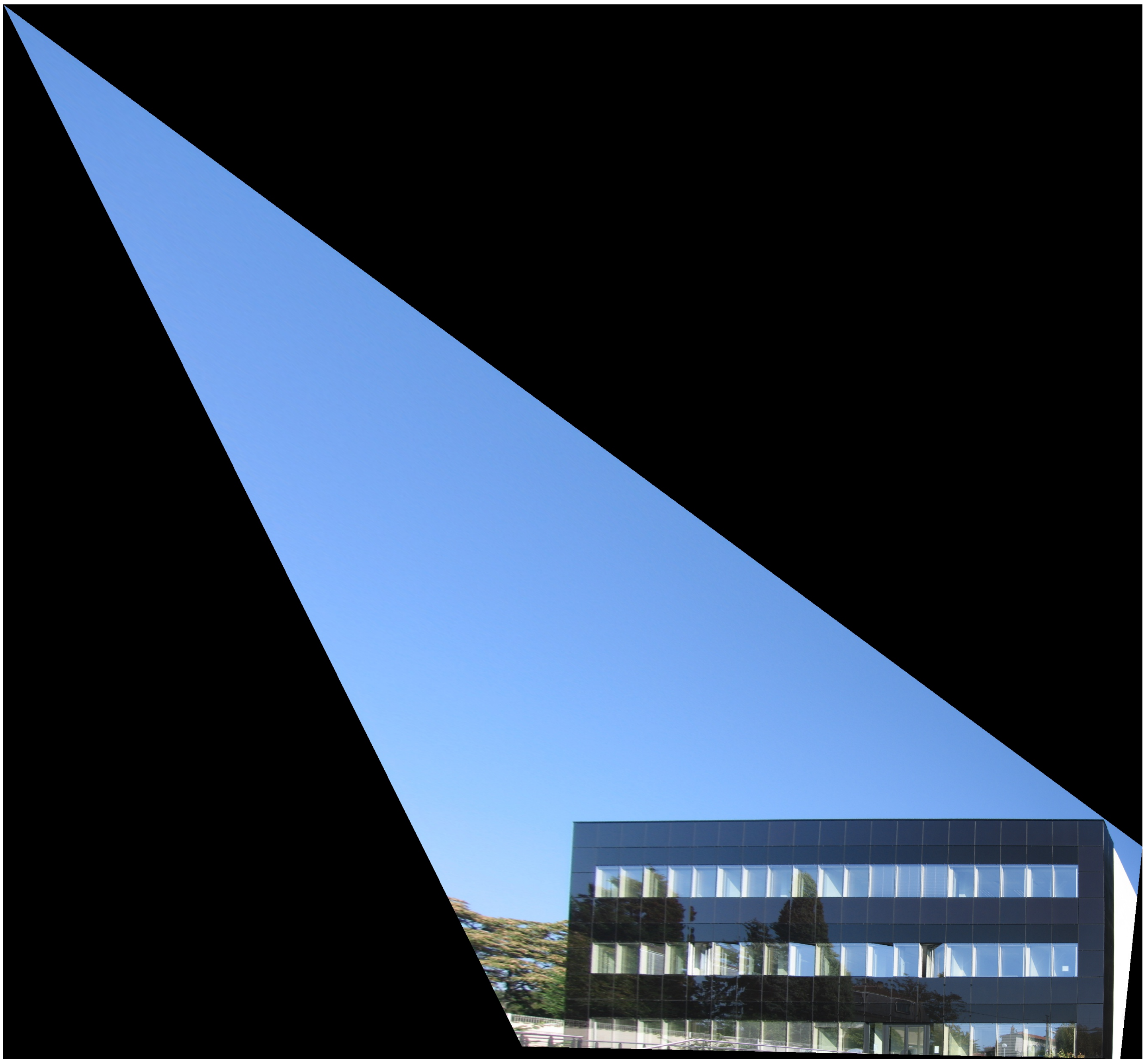

Figure 2: Result obtained using Method 1 :- Point-to-Point Correspondence Method for removing distortion

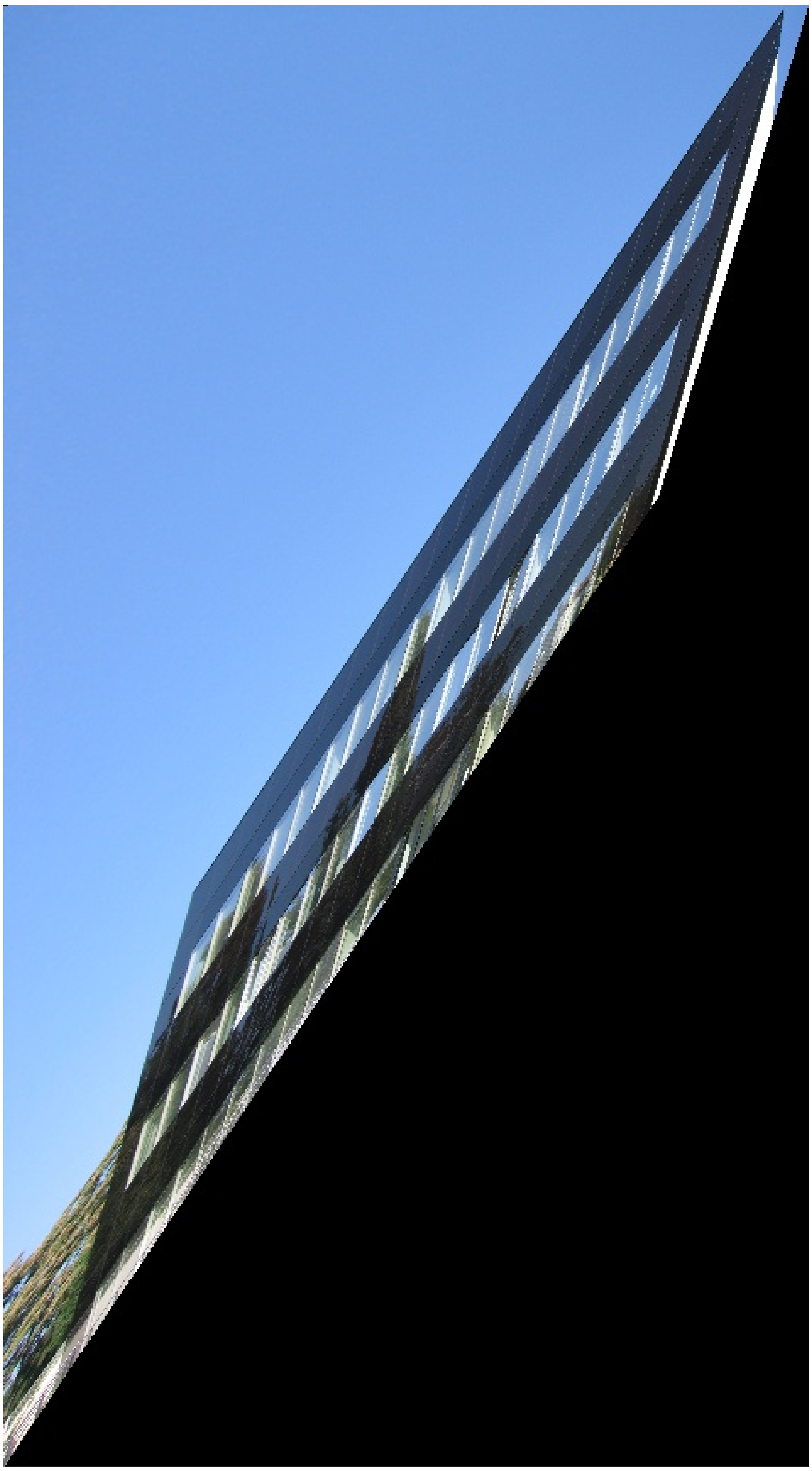

Figure 3: Results obtained using Method 2: Two Step Method :- Removing Projective Distortion

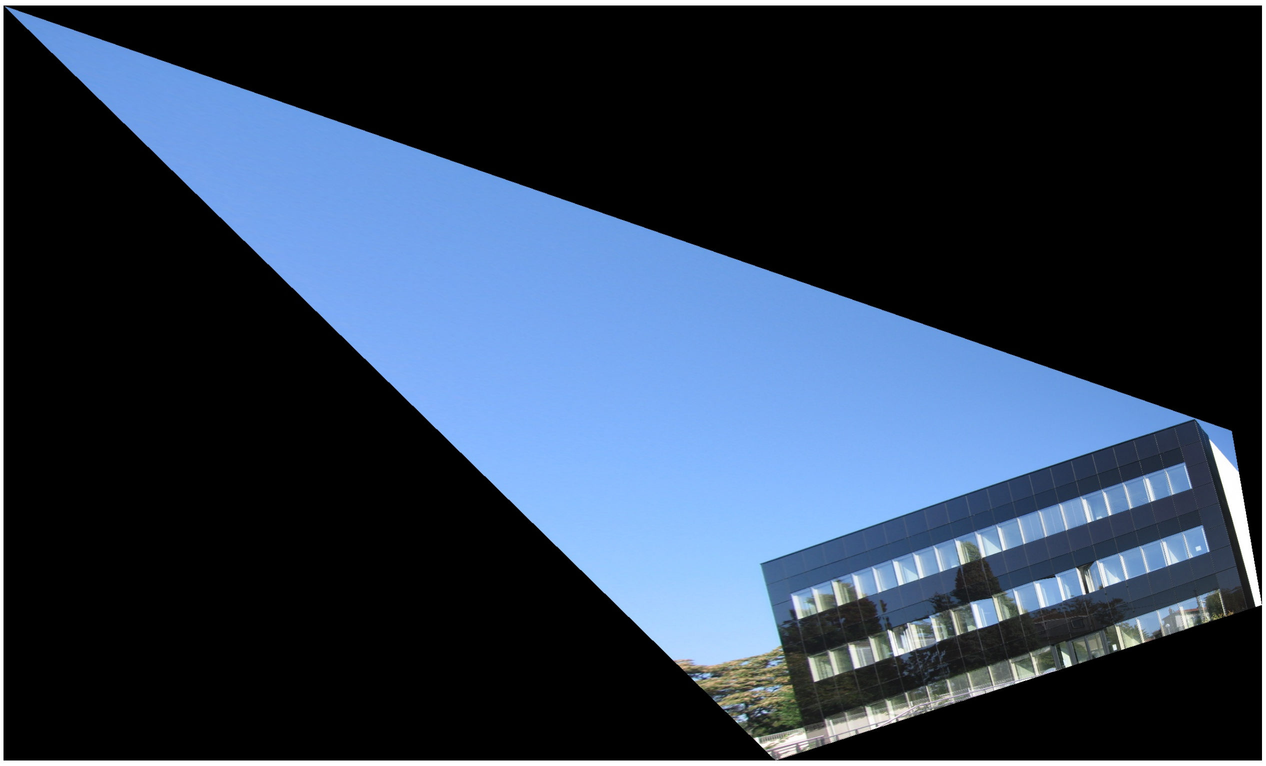

Figure 4: Results obtained using Method 2: Two Step Method :- Removing Affine Distortion as well as Projective Distortion

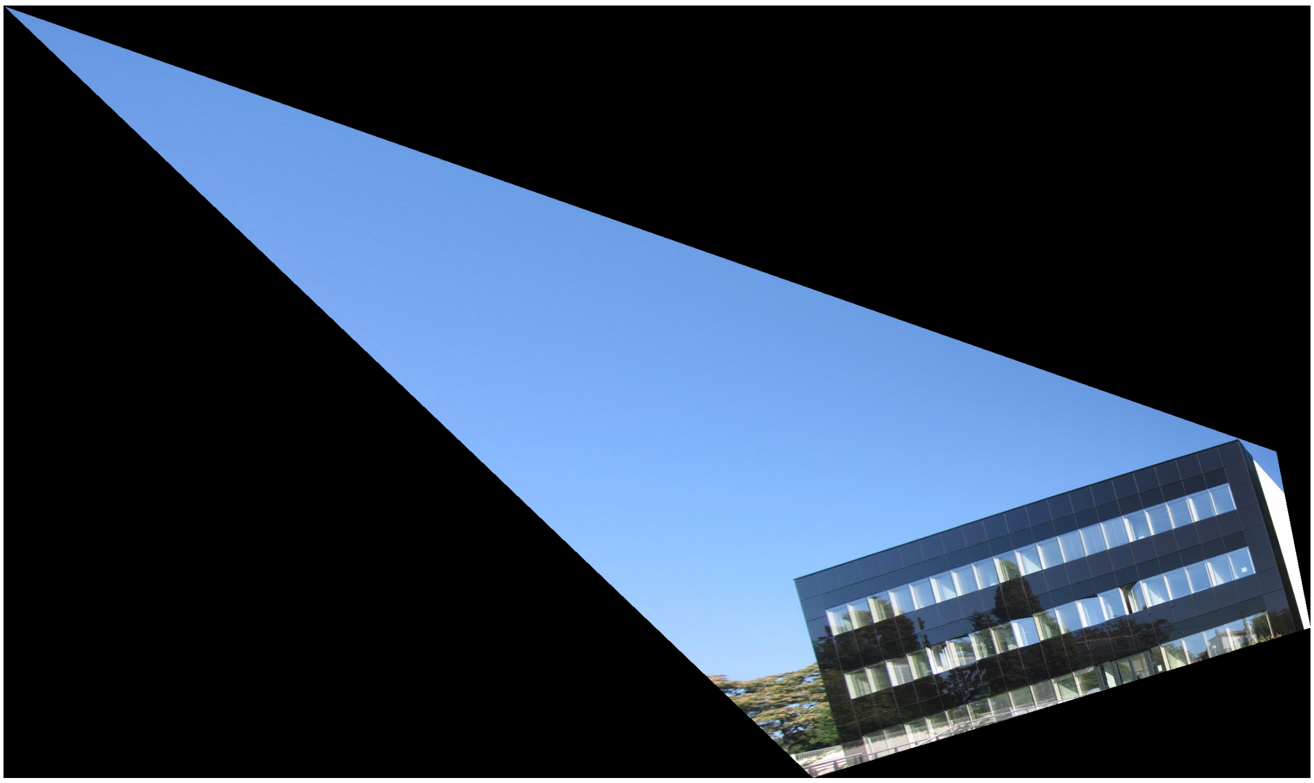

Figure 5: Results obtained using Method 3: One Step Method :- Removing Affine Distortion as well as Projective Distortion

Results for **Given input image 2**

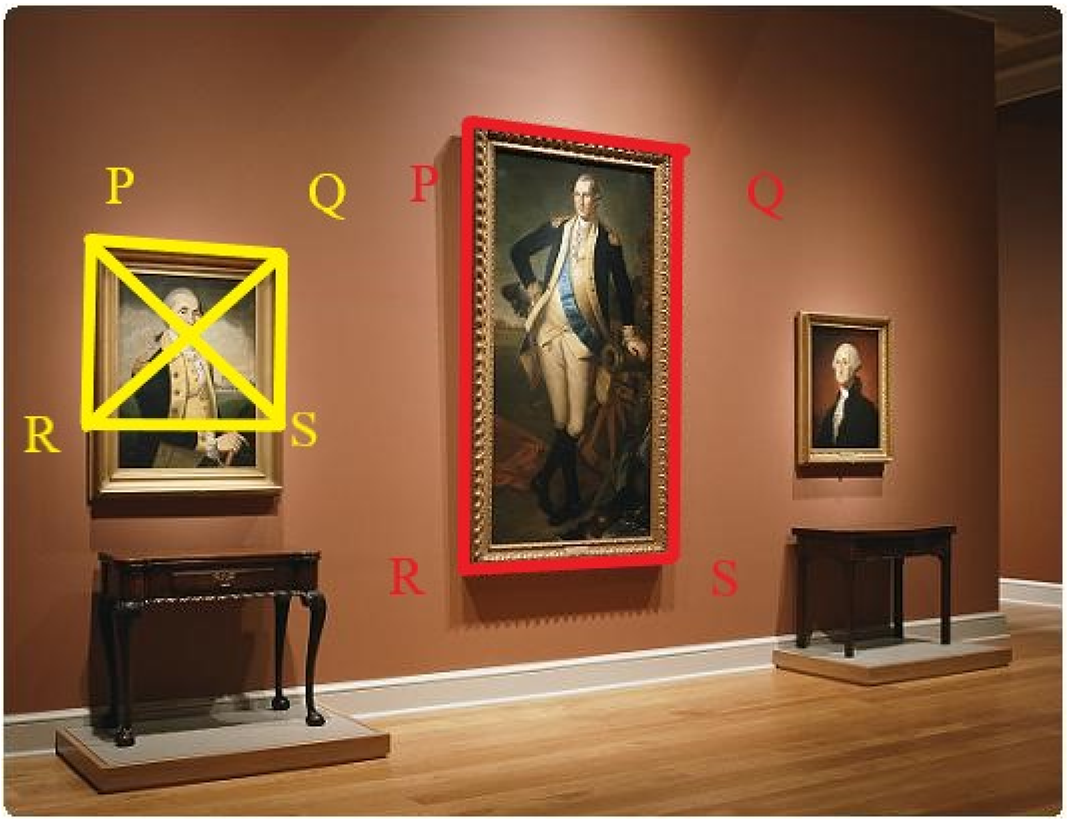

Figure 6: Input Image 2

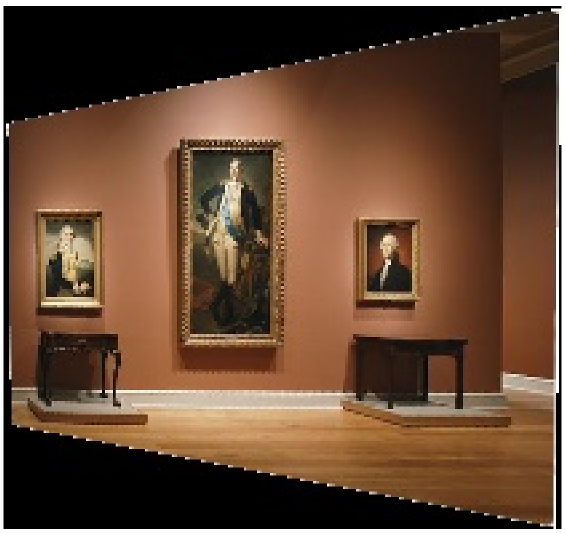

Figure 7: Result obtained using Method 1 :- Point-to-Point Correspondence Method for removing distortion

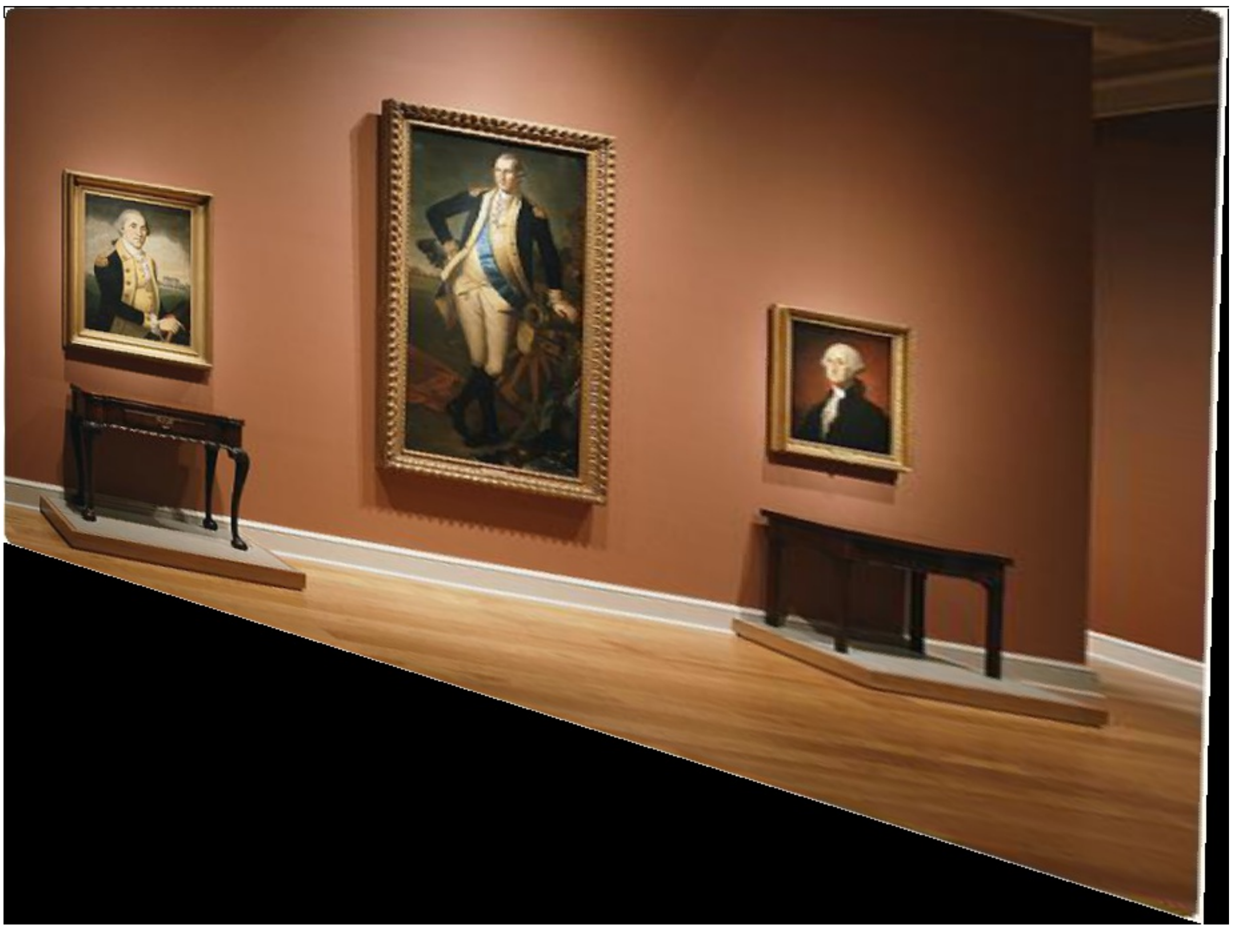

Figure 8: Results obtained using Method 2: Two Step Method :- Removing Projective Distortion

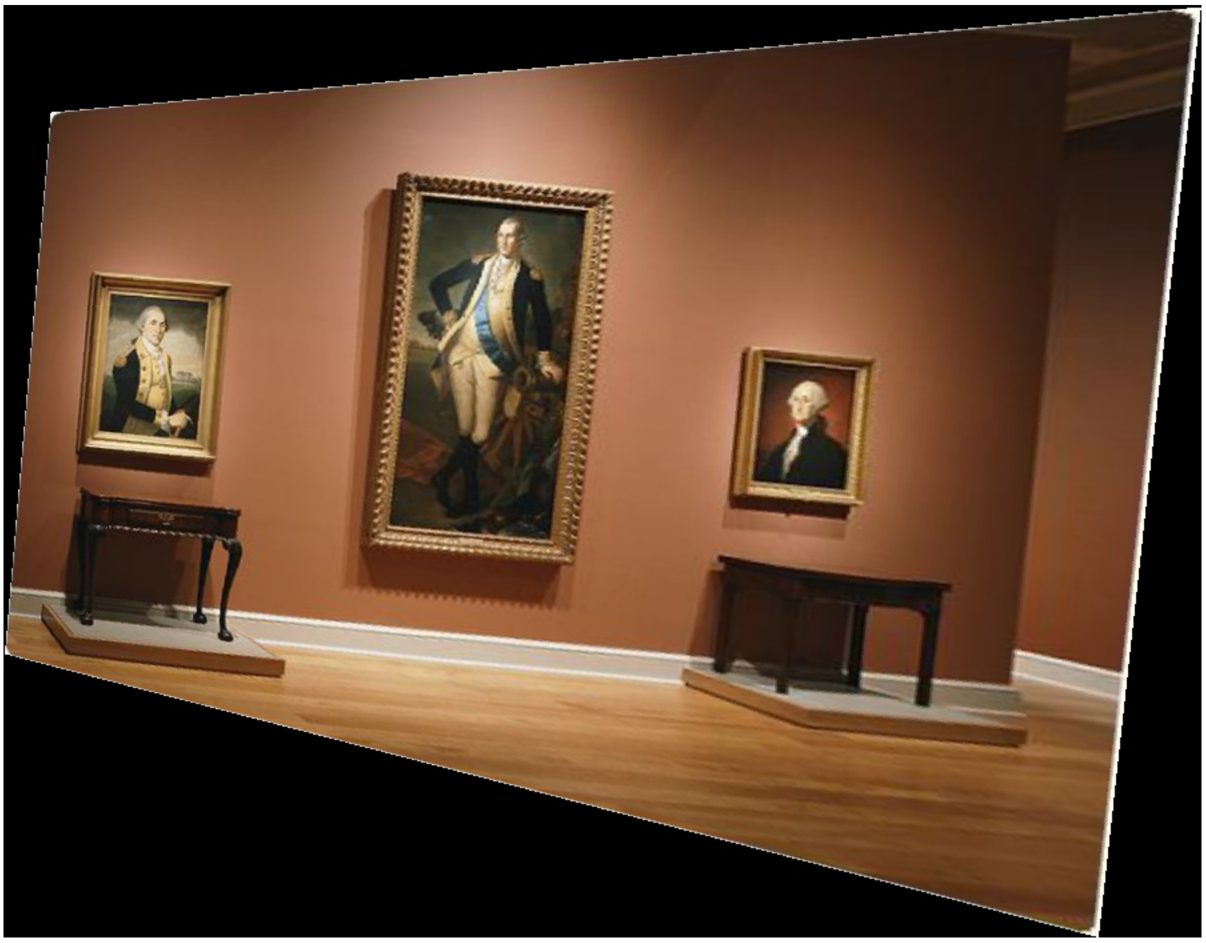

Figure 9: Results obtained using Method 2: Two Step Method :- Removing Affine Distortion as well as Projective Distortion

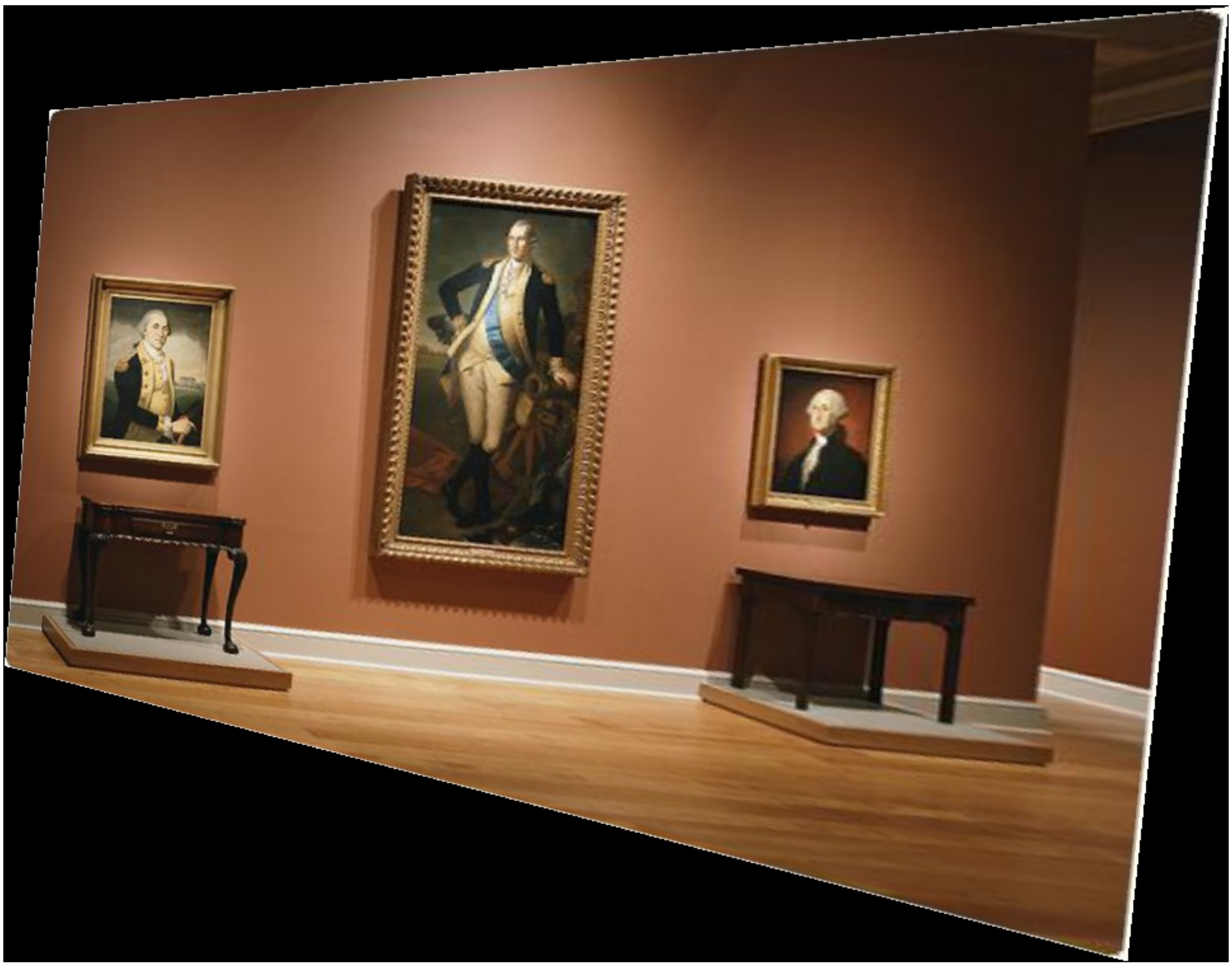

Figure 10: Results obtained using Method 3: One Step Method :- Removing Affine Distortion as well as Projective Distortion

Custom Image Inputs

Results for **Custom input image 1**

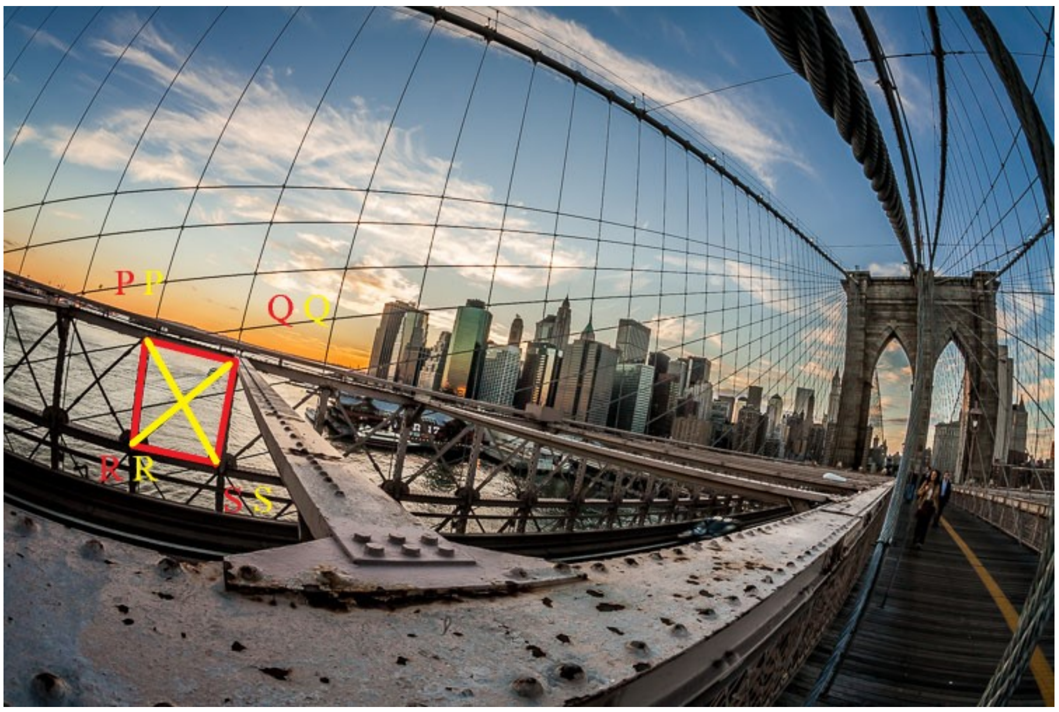

Figure 11: Input Image 1

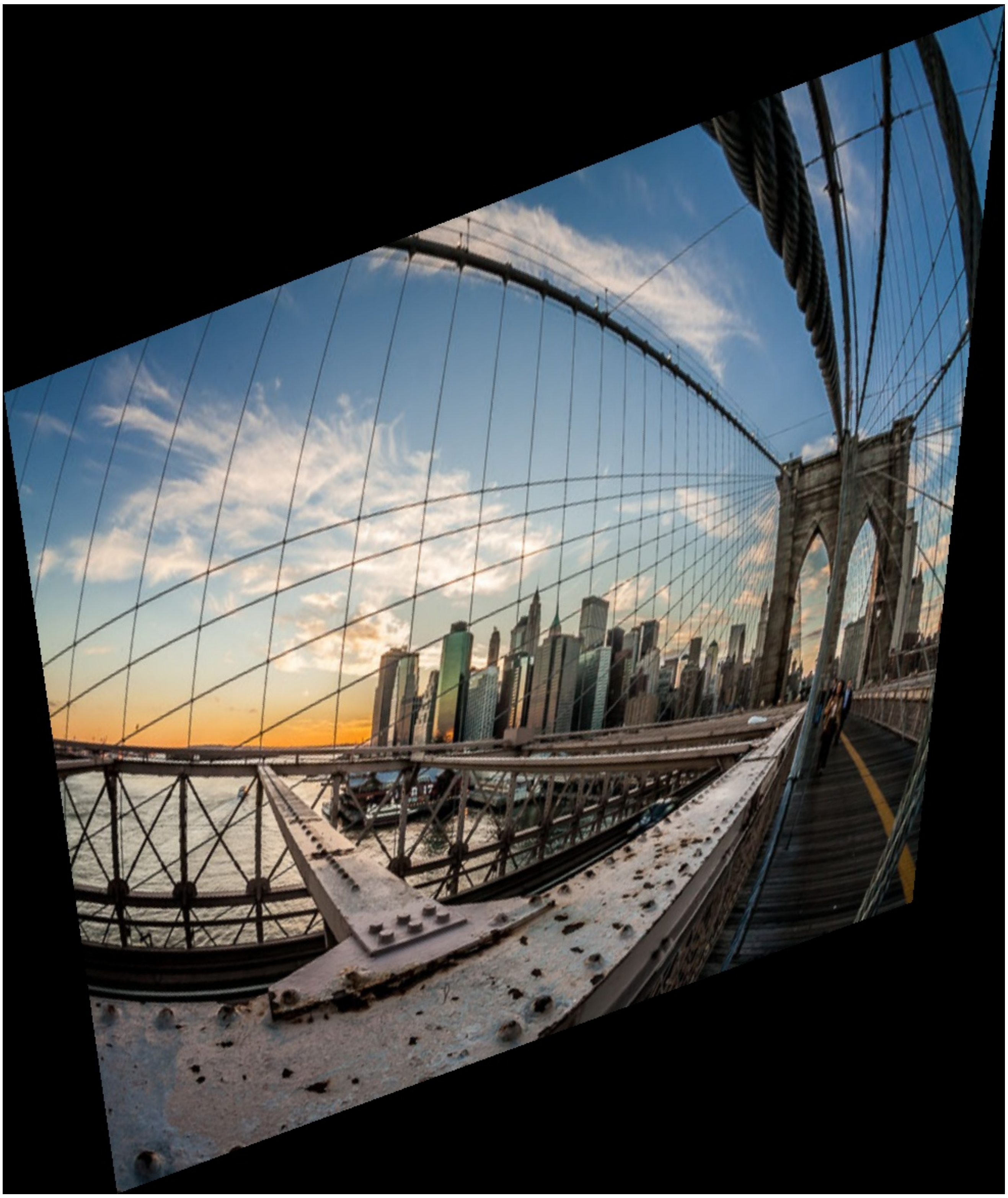

Figure 12: Result obtained using Method 1 :- Point-to-Point Correspondence Method for removing distortion

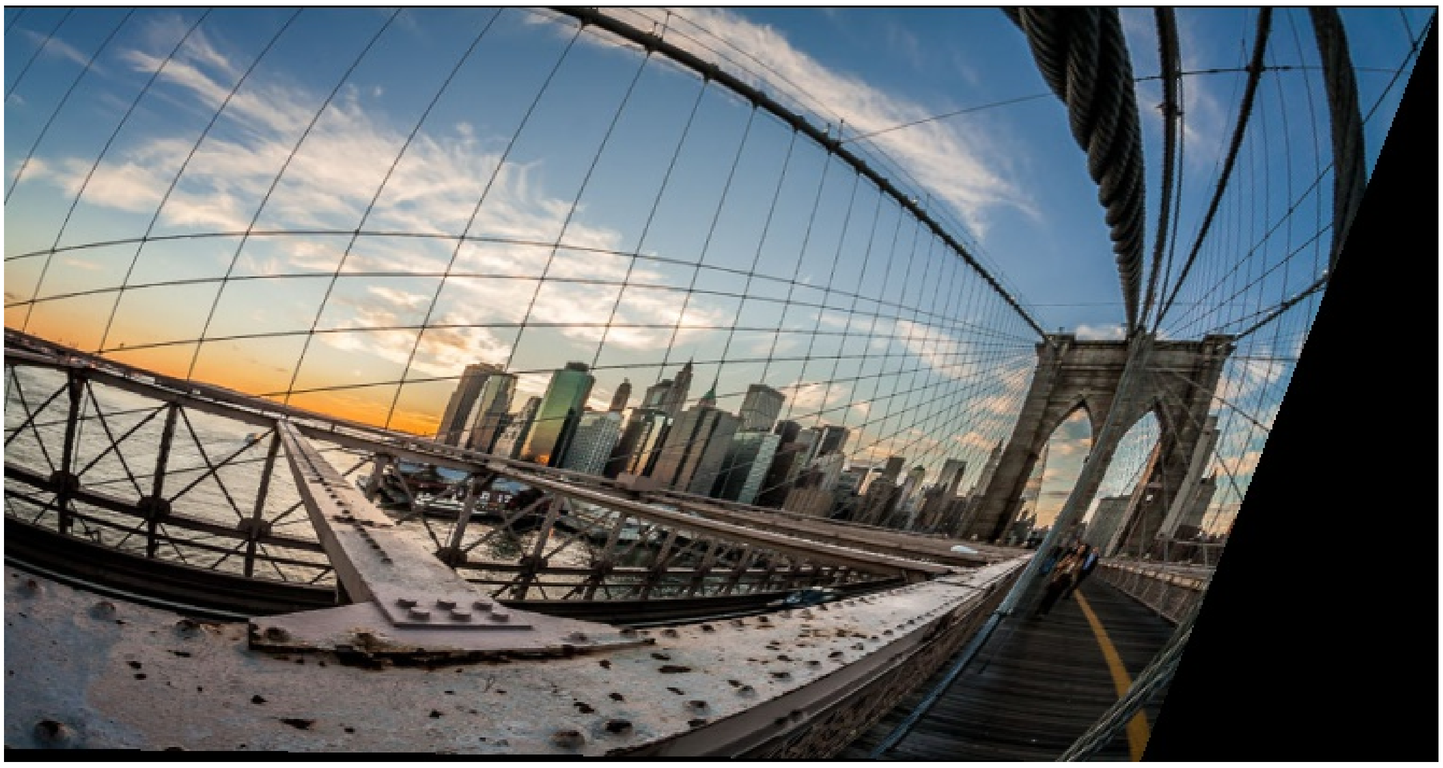

Figure 13: Results obtained using Method 2: Two Step Method :- Removing Projective Distortion

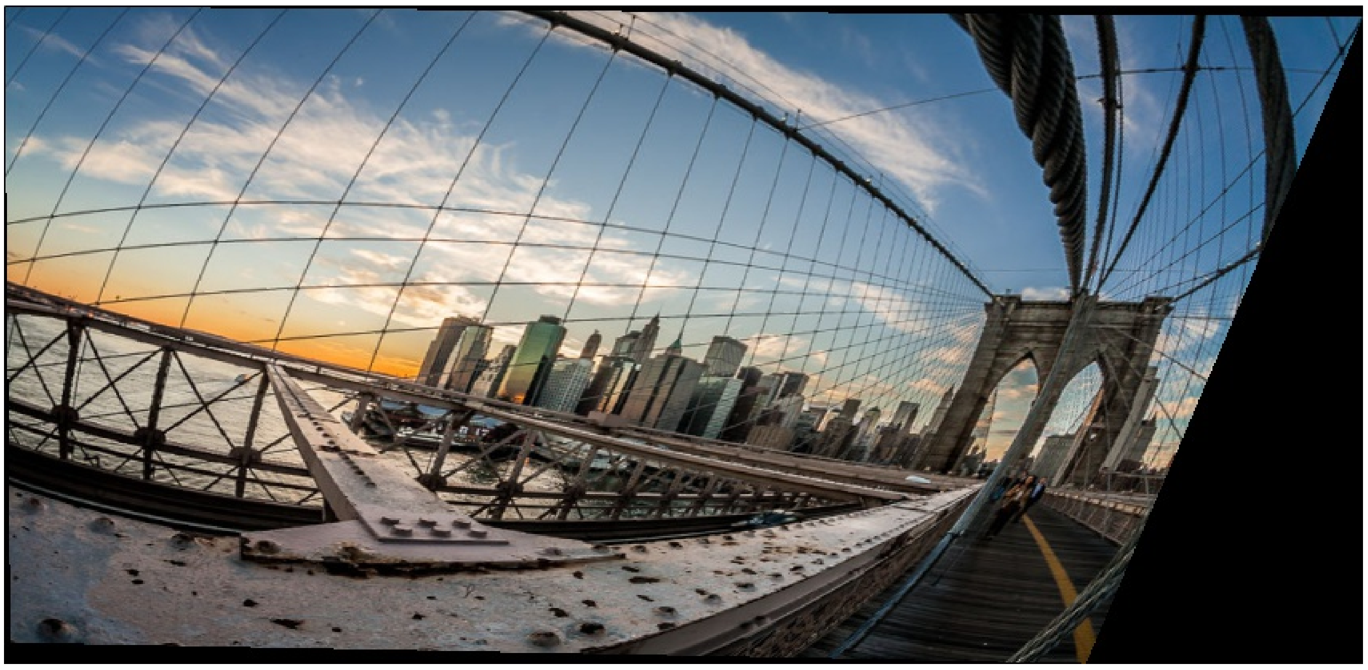

Figure 14: Results obtained using Method 2: Two Step Method :- Removing Affine Distortion as well as Projective Distortion

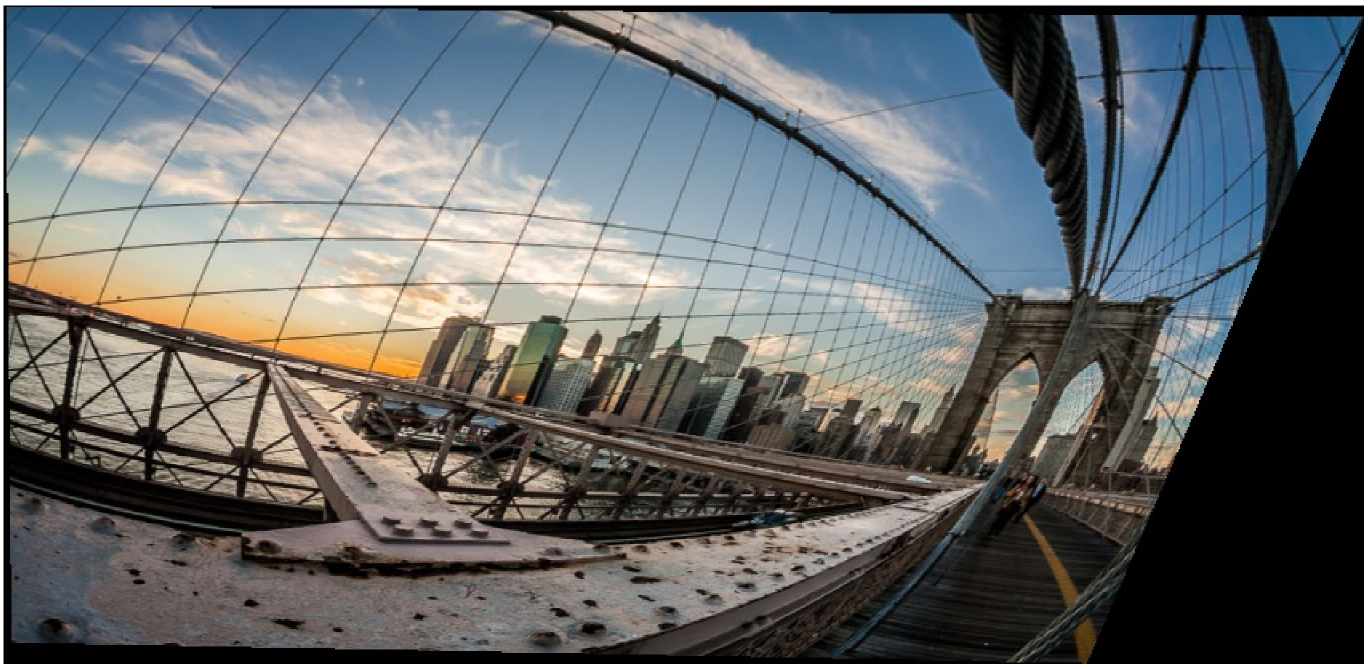

Figure 15: Results obtained using Method 3: One Step Method :- Removing Affine Distortion as well as Projective Distortion

Results for **Custom input image 2**

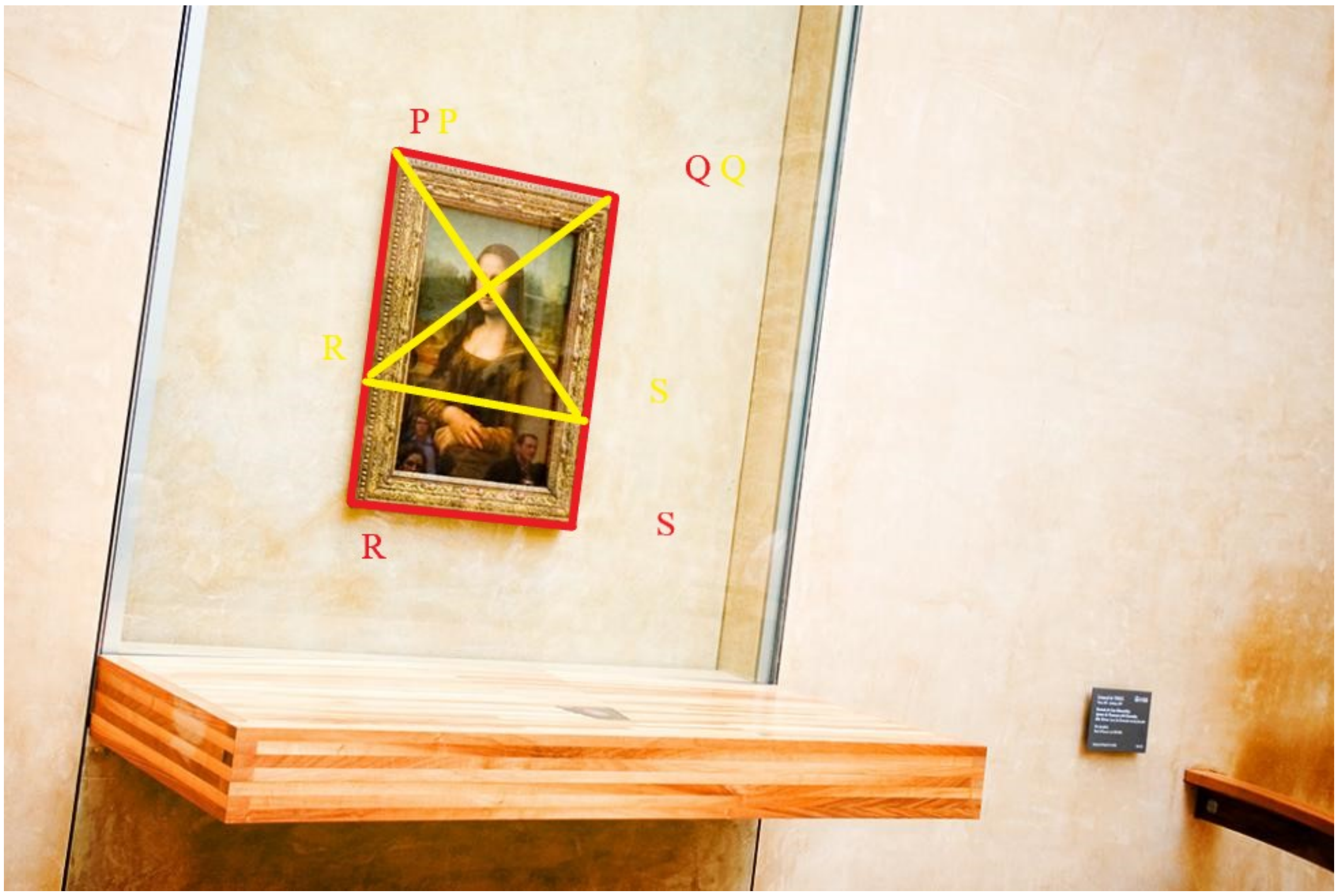

Figure 16: Input Image 2

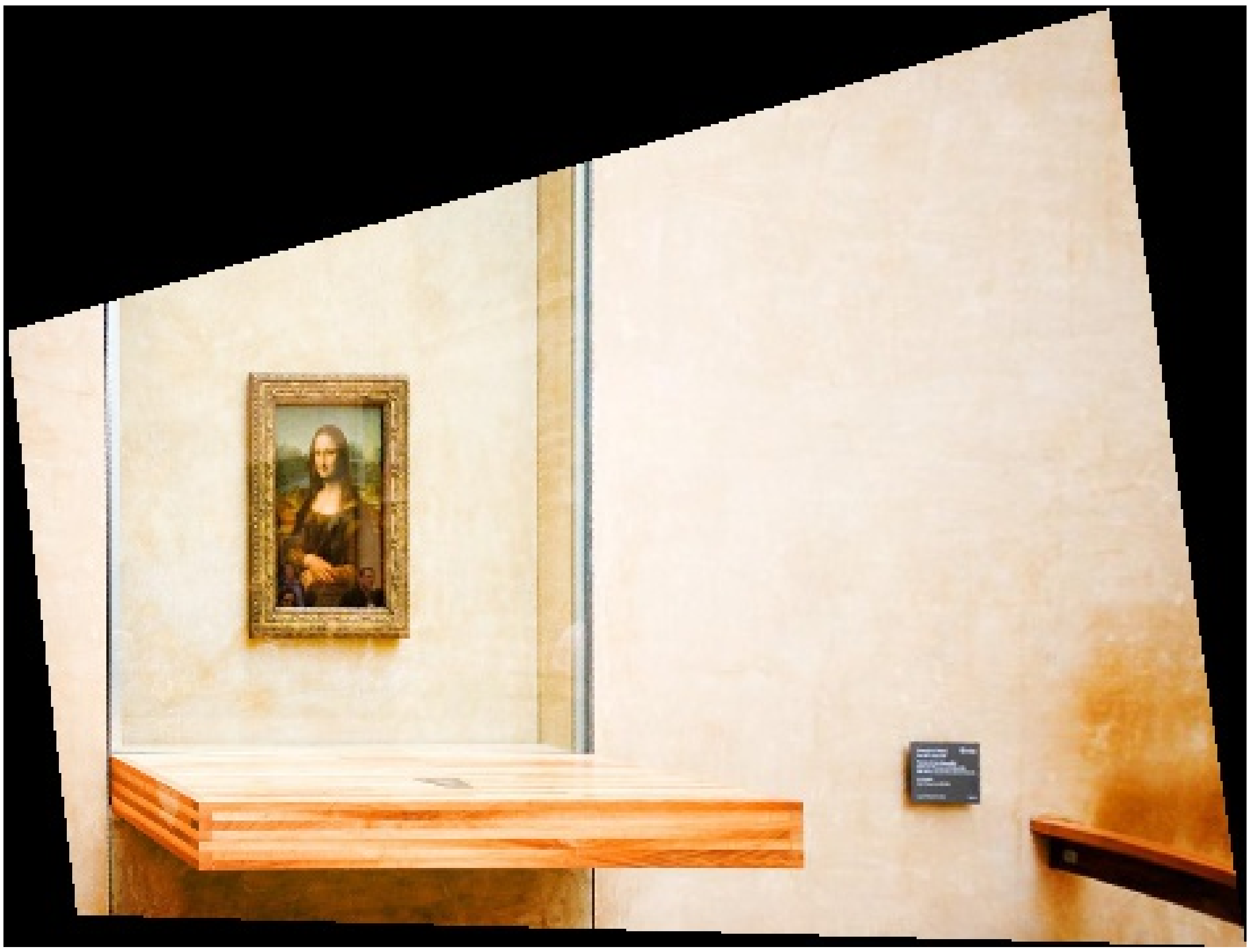

Figure 17: Result obtained using Method 1 :- Point-to-Point Correspondence Method for removing distortion

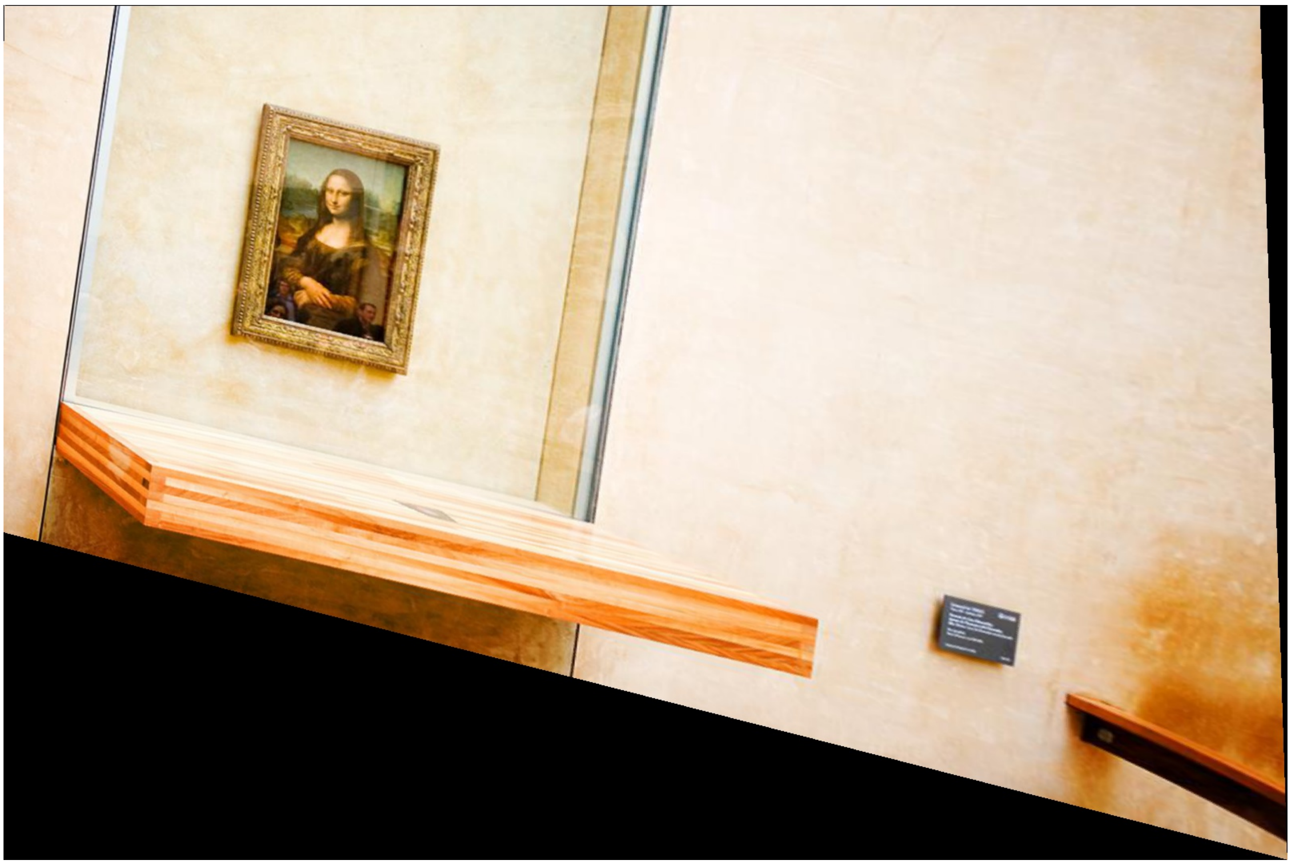

Figure 18: Results obtained using Method 2: Two Step Method :- Removing Projective Distortion

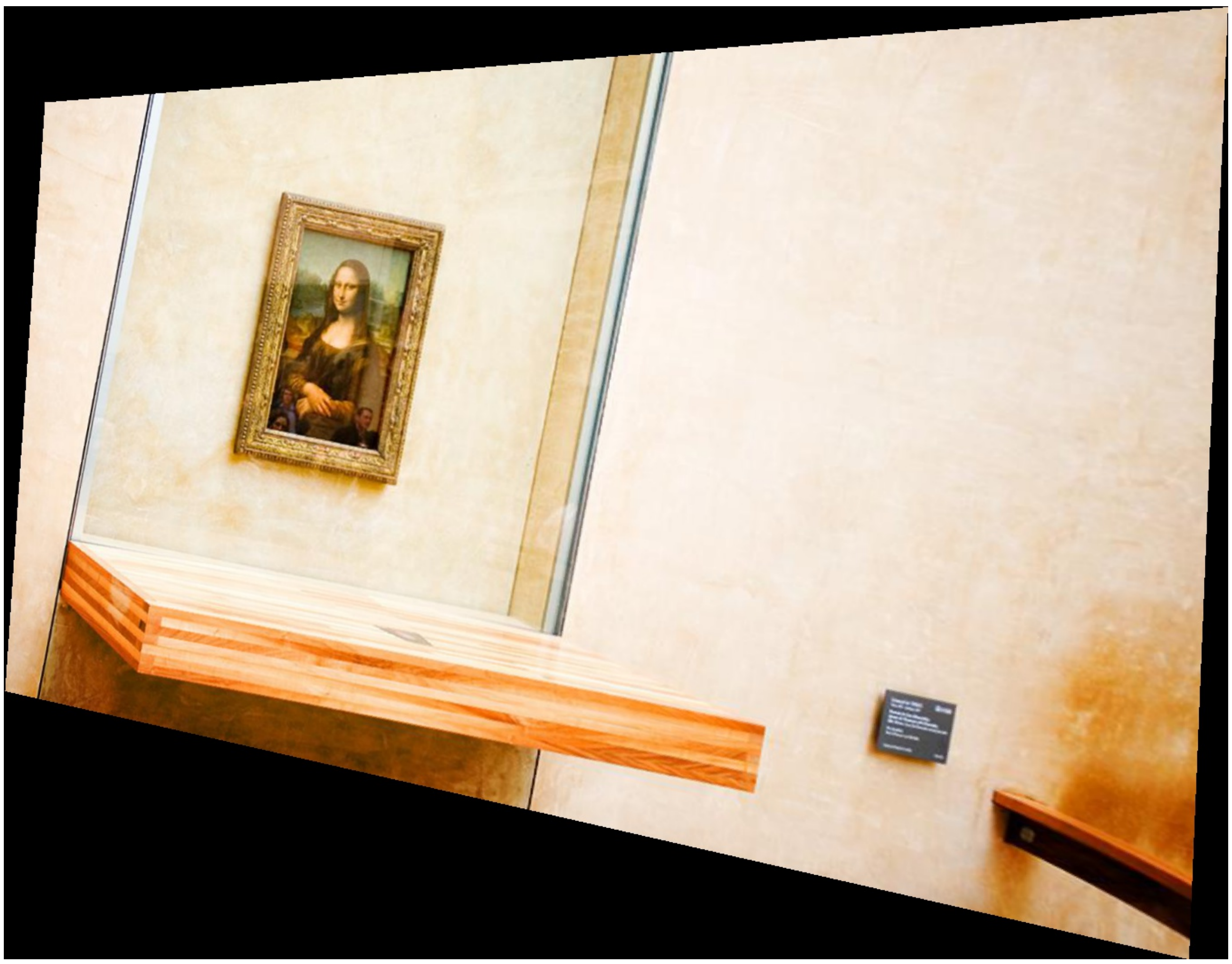

Figure 19: Results obtained using Method 2: Two Step Method :- Removing Affine Distortion as well as Projective Distortion

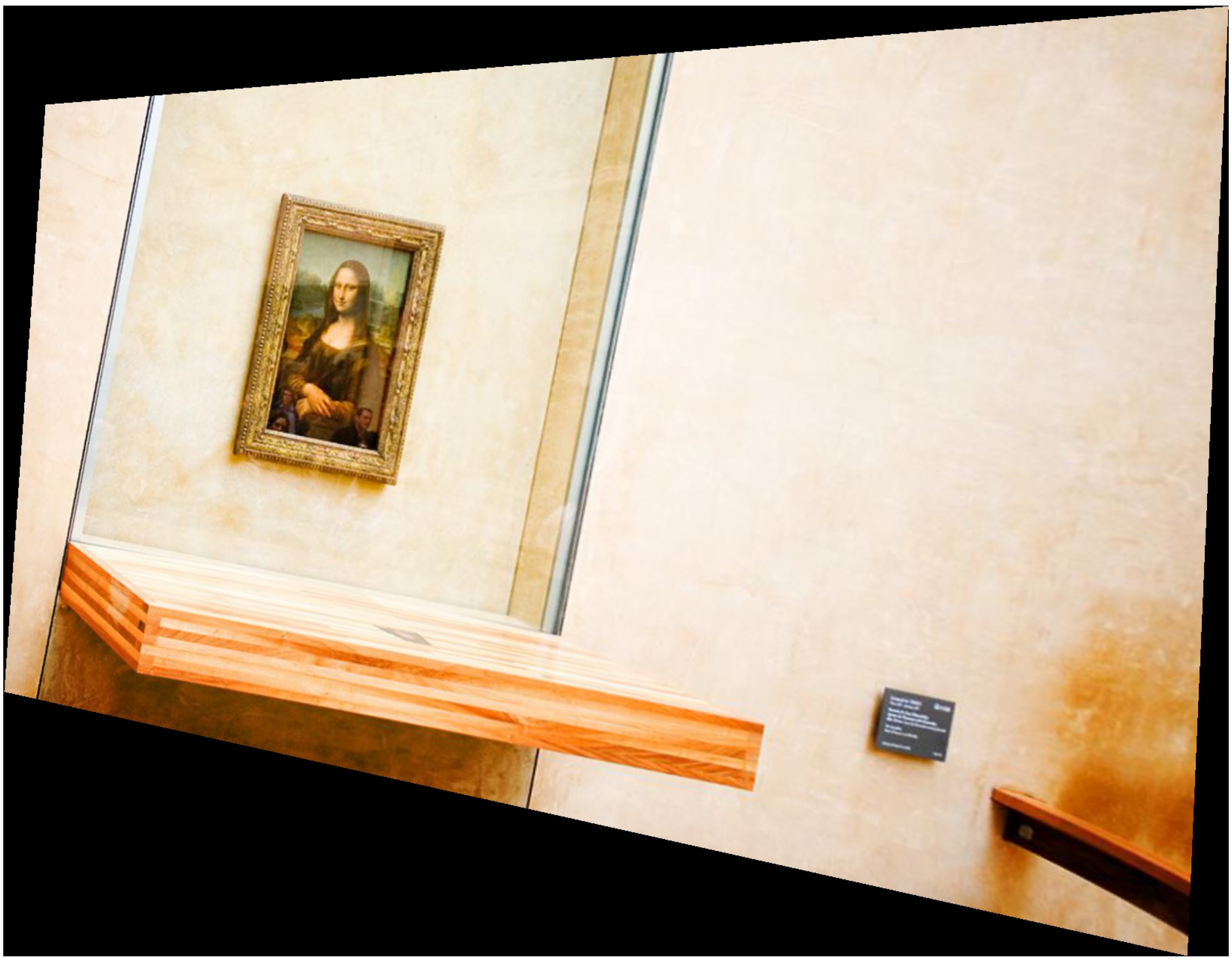

Figure 20: Results obtained using Method 3: One Step Method :- Removing Affine Distortion as well as Projective Distortion

4 Observations and Comments

In general, while experimenting with different images I have observed that :- In the single step method, the orthogonal lines required for estimating H, should be chosen in a consistent and good manner. If even 1 set of orthogonal lines is not good, or if all the lines are chosen poorly, such that together they cannot accurately or robustly capture the H matrix which is required for mapping, then we get poor results. This method is sensitive to the points chosen for computing the orthogonal lines.

The two-step method however, is more robust, and is not as sensitive to the points we initially pick for computing the orthogonal and parallel lines when compared to the one-step method. Though it has two steps, Both the steps are pretty simple, More Stable, and they run faster than the one-step method.

The images for which I have shown results above, Both the One-Step and Two-Step work equally good, for a similar set of points chosen. I would generally go ahead and use the two-step method, Since it is more Robust, Practical to understand as well as runs faster than the one-step method.

Extra Credit for the Two-Step Method :- Calculating different VL's

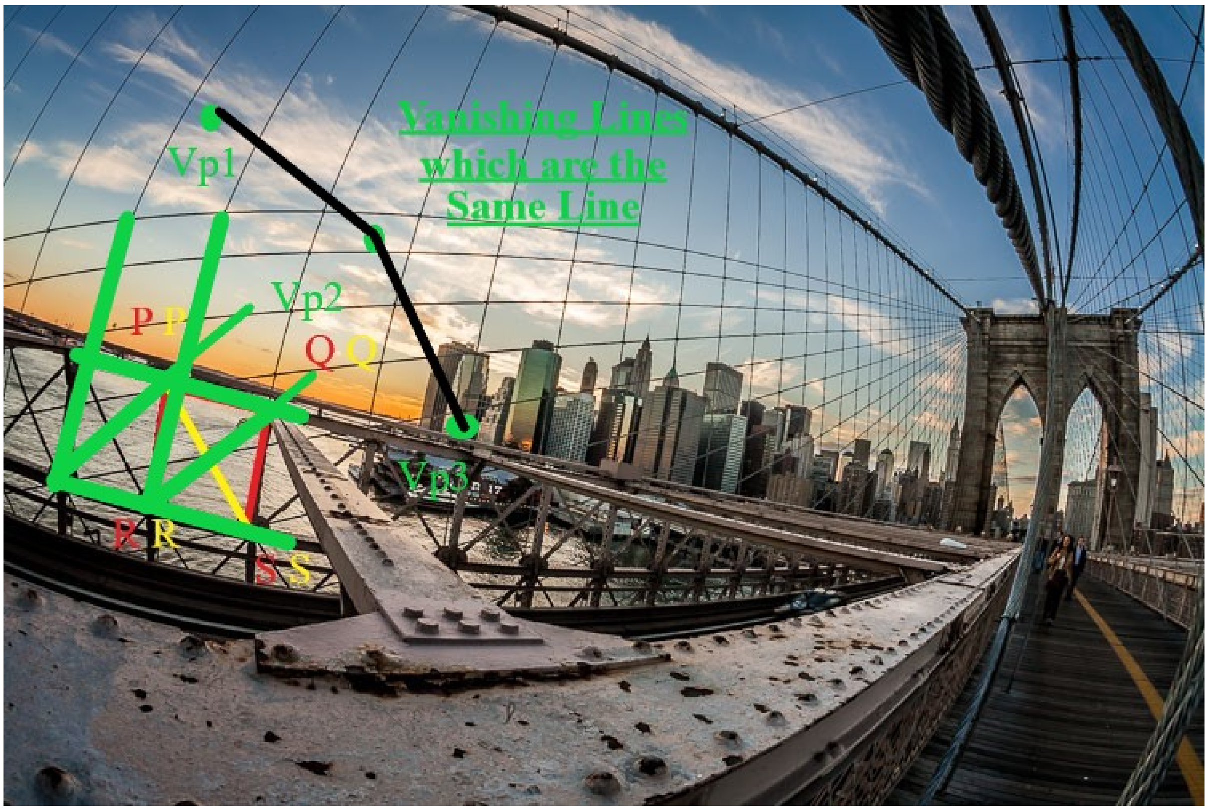

Figure 21: Extra Credit

l1=np.cross(np.array([31.0,296.0,1.0]),np.array([53.0,219.0,1.0]))
m1=np.cross(np.array([90.0,313.0,1.0]),np.array([110.0,235.0,1.0]))
l2=np.cross(np.array([31.0,296.0,1.0]),np.array([90.0,313.0,1.0]))
m2=np.cross(np.array([53.0,219.0,1.0]),np.array([110.0,235.0,1.0]))
l3=np.cross(np.array([31.0,296.0,1.0]),np.array([110.0,235.0,1.0]))
m3=np.cross(np.array([90.0,313.0,1.0]),np.array([167.0,252.0,1.0]))

Normalizing the lines, to find the vanishing points and normalizing them, we get :-
vp1 = array([6.48750000e+02, -1.86612500e+03, 1.00000000e+00])
vp2 = array([-1.11577600e+04, -2.92788000e+03, 1.00000000e+00])
vp3 = array([3.20913115e+03, -2.15800000e+03, 1.00000000e+00])

Finding the Vanishing Lines, and normalizing them, we get :-
vl1 = array([-4.67296300e-05, 5.19624437e-04, 1.00000000e+00])
vl2 = array([-2.29990700e-05, 4.29190439e-04, 1.00000000e+00])

This implies that approximately:-
vl1=[0,0,1]
vl2=[0,0,1]

We get the line at infinity in this case, because, we have taken points which are close together on a very small square, which are parallel, even in the distorted image. Theoretically this can be extended to any image with distortions and we would get the same vanishing line, within numerical error and precision limits. For this example, I chose these points and lines, since its easier to illustrate that, multiple vanishing lines are actually the same within numerical limits It just so happens the vanishing lines in this case are the same as l_∞ , But the results hold true generally for other points and lines chosen as well.

5 Python Code

5.1 Method 1

```
import numpy as np
import cv2
import math

# Importing the Neccessary Libraries required

def HomographyMatrix(src_img_pts,world_plane_pts):

    tb=[]          # Lists used for creating Matrices A,b
    ta=[]

    for i in range(0,len(src_img_pts)):
        # MATHEMATICAL EQUATIONS which are used for filling up entries of A, b
        tmp=[world_plane_pts[i][0],world_plane_pts[i][1],1,0,0,0,
        -src_img_pts[i][0]*world_plane_pts[i][0],
        -src_img_pts[i][0]*world_plane_pts[i][1]]
        ta.append(tmp)
        tmp=[0,0,0,world_plane_pts[i][0],world_plane_pts[i][1],1,
        -src_img_pts[i][1]*world_plane_pts[i][0],
        -src_img_pts[i][1]*world_plane_pts[i][1]]
        ta.append(tmp)
        A=np.asarray(ta)          # Computing the Matrix A for estimating the Homography H

        tmp=[src_img_pts[i][0],src_img_pts[i][1]]
        tb.append(tmp)
        tmp=np.asarray(tb)
        b=tmp.reshape(-1)         # Computing the Matrix b for estimating the Homography H


    tmp=np.dot(np.linalg.pinv(A),b.transpose())    # Computing Pesudo Inverse
    H=np.zeros((3,3))
    H[0]=tmp[0:3]
    H[1]=tmp[3:6]
    H[2][0:2]=tmp[6:8]
    H[2][2]=1
                                  # The Homography Matrix H
    return H

def WeightedAverageRGBPixelValue(pt, img):

    x1=int(math.floor(pt[0]))           # Used in Weighted Average Pixel Computation
    x2=int(math.ceil(pt[0]))
    y1=int(math.floor(pt[1]))
    y2=int(math.ceil(pt[1]))

    Wp=1/np.linalg.norm(np.array([pt[0]-x1,pt[1]-y1]))  # Weights
    Wq=1/np.linalg.norm(np.array([pt[0]-x1,pt[1]-y2]))
    Wr=1/np.linalg.norm(np.array([pt[0]-x2,pt[1]-y1]))
    Ws=1/np.linalg.norm(np.array([pt[0]-x2,pt[1]-y2]))

    pixel_value = (Wp*img[y1][x1] + Wq*img[y2][x1] + Wr*img[y1][x2] + Ws*img[y2][x2])
    /(Wp+Wq+Wr+Ws)

    return pixel_value        # Return Pixel Value

def ProjectionImage(H,world_plane_img):

    ImgP=np.asarray([0.0,0.0,1.0])      # Image Plane Coordinates
    ImgQ=np.asarray([float(np.shape(world_plane_img)[1])-1.0,0.0,1.0])
    ImgR=np.asarray([0.0,float(np.shape(world_plane_img)[0])-1.0,1.0])
    ImgS=np.asarray([float(np.shape(world_plane_img)[1]-1.0),
    float(np.shape(world_plane_img)[0])-1.0,1.0])


    WorldA=np.dot(H,ImgP)             # Computing World Coordinates from Image Plane
    WorldA=WorldA/WorldA[2]           # Using the Homography Matrix H
    WorldB=np.dot(H,ImgQ)
    # Finding corner points in image and mapping to Real World
    WorldB=WorldB/WorldB[2]
    WorldC=np.dot(H,ImgR)
    WorldC=WorldC/WorldC[2]
    WorldD=np.dot(H,ImgS)
    WorldD=WorldD/WorldD[2]
                                # Image Boundaries
    xmin = int(math.floor(min([WorldA[0],WorldB[0],WorldC[0],WorldD[0]])))
    xmax = int(math.ceil(max([WorldA[0],WorldB[0],WorldC[0],WorldD[0]])))
    ymin = int(math.floor(min([WorldA[1],WorldB[1],WorldC[1],WorldD[1]])))
    ymax = int(math.ceil(max([WorldA[1],WorldB[1],WorldC[1],WorldD[1]])))

    yLength=ymax-ymin        # Computing the Shape/Size of the output images
    xLength=xmax-xmin

    src_img=np.zeros((yLength,xLength,3))
    Hn=np.linalg.pinv(H)
    Hn=Hn/Hn[2][2]
                               # Applying Projection from Image to World
    for i in range(0,yLength):
        for j in range(0,xLength):
            tmp=np.array([j+xmin,i+ymin,1.0])
            xp=np.array(np.dot(Hn,tmp))        # Fitting the Image to World
            xp=xp/xp[2]
            if((xp[0]>0)and(xp[0]<world_plane_img.shape[1]-1)
            and(xp[1]>0)and(xp[1]<world_plane_img.shape[0]-1)):
                src_img[i][j]=WeightedAverageRGBPixelValue(xp,world_plane_img)

            # Returning the Image


    output_img = src_img
    return output_img

# Second Image
image2=cv2.imread("2.jpg")

t2=[]
tI=[]

t2.append([290.0,109.0,1.0])        # Pixel Coordinates for Mapping in the Image Plane
t2.append([449.0,143.0,1.0])        # Image 2 Input, P,Q,R,S
t2.append([254.0,367.0,1.0])
t2.append([417.0,383.0,1.0])
```

```
pts_image2=np.asarray(t2)

tI.append([0.0,0.0,1.0])              # Actual Sizes in the World Plane
tI.append([60.0,0.0,1.0])             # P,Q,R,S
tI.append([0,100.0,1.0])
tI.append([60.0,100.0,1.0])

pts_imageI=np.asarray(tI)

H=HomographyMatrix(pts_imageI,pts_image2)         # Computing H Matrix

output=ProjectionImage(H,image2)           # Finding the Output projection onto World Plane
cv2.imwrite('Out2.jpg',output)


# First Image

image1=cv2.imread("1.jpg")

t1=[]
tJ=[]

t1.append([104.0,238.0,1.0])
t1.append([163.0,254.0,1.0])           # Image 1 Input P,Q,R,S
t1.append([93.0,313.0,1.0])
t1.append([154.0,330.0,1.0])

pts_image1=np.asarray(t1)

tJ.append([0.0,0.0,1.0])
tJ.append([70.0,0.0,1.0])             # P,Q,R,S
tJ.append([0,120.0,1.0])
tJ.append([70.0,120.0,1.0])

pts_imageJ=np.asarray(tJ)

H=HomographyMatrix(pts_imageJ,pts_image1)

output=ProjectionImage(H,image1)
cv2.imwrite('Out1.jpg',output)
```

5.2 Method 2

```
import numpy as np
import cv2
import math

# Importing the Neccessary Libraries required

# Removing Projective Distortion

def VLMatrix(array):
                                    #Finding the Vanishing Line
    l1=np.cross(array[0],array[1])
    l2=np.cross(array[2],array[3])

    Vp1=np.cross(l1,l2)
    Vp1=Vp1/Vp1[2]                     # Vanishing Point 1

    l3=np.cross(array[0],array[2])
    l4=np.cross(array[1],array[3])

    Vp2=np.cross(l3,l4)
    Vp2=Vp2/Vp2[2]                    # Vanishing Point 2
    VL=np.cross(Vp1,Vp2)

    VL=VL/VL[2]                         # Vanishing Line

    H=np.zeros((3,3))
    H[0][0]=1
    H[1][1]=1
    H[2]=VL                             # Computing the Homography H

    return H

# Removing Affine Distortion

def AffineMatrix(arrays1,arrays2):

    ta=[]                                   # Using the Cos theta Method
    tb=[]

    l1=np.cross(arrays1[0],arrays1[1])
    m1=np.cross(arrays1[0],arrays1[2])
    l1=l1/l1[2]                             # 1 pair of Lines perpendicular
    m1=m1/m1[2]

    l2=np.cross(arrays1[0],arrays1[3])
    m2=np.cross(arrays1[1],arrays1[2])
    l2=l2/l2[2]                             # 2 pair of lines perpendicular
    m2=m2/m2[2]

    ta.append([l1[0]*m1[0],l1[0]*m1[1]+l1[1]*m1[0]])
    tb.append([-l1[1]*m1[1]])        # Based on Theory and Mathematrical Equations

    ta.append([l2[0]*m2[0],l2[0]*m2[1]+l2[1]*m2[0]])
    tb.append([-l2[1]*m2[1]])

    A=np.asarray(ta)
    b=np.asarray(tb)

    tmp=np.dot(np.linalg.pinv(A),b)         # Finding the Matrix S

    S=np.zeros((2,2))
    S[0][0]=tmp[0]
    S[0][1]=tmp[1]
    S[1][0]=tmp[1]
    S[1][1]=1

    u,s,vh=np.linalg.svd(S)                 # Computing SVD of S

    s1=np.diag(s)

    D=np.sqrt(s1)
    K=np.dot(np.dot(u,D),u.transpose())         # Computing matrix A

    H=np.zeros((3,3))
    H[0][0]=K[0][0]                     # Computing Matrix H
    H[0][1]=K[0][1]
    H[1][0]=K[1][0]
    H[1][1]=K[1][1]
    H[2][2]=1


    return H

def WeightedAverageRGBPixelValue(pt, img):          # Applying the Weighted Average Method
```

```
    x1=int(math.floor(pt[0]))                    # Used in Weighted Average Pixel Computation
    x2=int(math.ceil(pt[0]))
    y1=int(math.floor(pt[1]))
    y2=int(math.ceil(pt[1]))

    Wp=1/np.linalg.norm(np.array([pt[0]-x1,pt[1]-y1]))
    Wq=1/np.linalg.norm(np.array([pt[0]-x1,pt[1]-y2]))     # Weights
    Wr=1/np.linalg.norm(np.array([pt[0]-x2,pt[1]-y1]))
    Ws=1/np.linalg.norm(np.array([pt[0]-x2,pt[1]-y2]))

    pixel_value = (Wp*img[y1][x1] + Wq*img[y2][x1] + Wr*img[y1][x2] + Ws*img[y2][x2])
    /(Wp+Wq+Wr+Ws)

    return pixel_value                      # Return Pixel Value

def ProjectionImage(H,world_plane_img):

    ImgP=np.asarray([0.0,0.0,1.0])       # Image Plane Coordinates
    ImgQ=np.asarray([float(np.shape(world_plane_img)[1])-1.0,0.0,1.0])
    ImgR=np.asarray([0.0,float(np.shape(world_plane_img)[0])-1.0,1.0])
    ImgS=np.asarray([float(np.shape(world_plane_img)[1]-1.0),
    float(np.shape(world_plane_img)[0])-1.0,1.0])


    WorldA=np.dot(H,ImgP)         # Computing World Coordinates from Image Plane
    WorldA=WorldA/WorldA[2]           # Using the Homography Matrix H
    WorldB=np.dot(H,ImgQ)         # Finding corner points in image and mapping to Real World
    WorldB=WorldB/WorldB[2]
    WorldC=np.dot(H,ImgR)
    WorldC=WorldC/WorldC[2]
    WorldD=np.dot(H,ImgS)
    WorldD=WorldD/WorldD[2]
                              # Image Boundaries
    xmin = int(math.floor(min([WorldA[0],WorldB[0],WorldC[0],WorldD[0]])))
    xmax = int(math.ceil(max([WorldA[0],WorldB[0],WorldC[0],WorldD[0]])))
    ymin = int(math.floor(min([WorldA[1],WorldB[1],WorldC[1],WorldD[1]])))
    ymax = int(math.ceil(max([WorldA[1],WorldB[1],WorldC[1],WorldD[1]])))

    yLength=ymax-ymin           # Computing the Shape/Size of the output images
    xLength=xmax-xmin
                              # Applying Projection from Image to World
    src_img=np.zeros((yLength,xLength,3))
    Hn=np.linalg.pinv(H)
    Hn=Hn/Hn[2][2]

    for i in range(0,yLength):
        for j in range(0,xLength):
            tmp=np.array([j+xmin,i+ymin,1.0])
            xp=np.array(np.dot(Hn,tmp))
            xp=xp/xp[2]                         # Fitting the Image to World
            if((xp[0]>0)and(xp[0]<world_plane_img.shape[1]-1)
            and(xp[1]>0)and(xp[1]<world_plane_img.shape[0]-1)):
                src_img[i][j]=WeightedAverageRGBPixelValue(xp,world_plane_img)


    output_img = src_img          # Returning Output Image
    return output_img



image1=cv2.imread("1.jpg")              # Reading Image Input
image2=cv2.imread("2.jpg")

t1=[]
```

```
t2=[]

t1.append([104.0,238.0,1.0])        # Pixel Coordinates for Mapping in the Image Plane
t1.append([163.0,254.0,1.0])        # Rectangular Perpendicular Lines
t1.append([93.0,313.0,1.0])         # IMAGE 1  P,Q,R,S input
t1.append([154.0,330.0,1.0])

pts_image1=np.asarray(t1)

t2.append([290.0,109.0,1.0])       # Pixel Coordinates for Mapping in the Image Plane
t2.append([449.0,143.0,1.0])       # Rectangular Perpendicualr Lines
t2.append([254.0,367.0,1.0])       # Image 2 P,Q,R,S input
t2.append([417.0,383.0,1.0])

pts_image2=np.asarray(t2)


H1=VLMatrix(pts_image1)                     # Computing H Matrix
output=ProjectionImage(H1,image1)           # Finding the Output projection onto World Plane
cv2.imwrite('1_2stepP.jpg',output)

H2=VLMatrix(pts_image2)
output=ProjectionImage(H2,image2)
cv2.imwrite('2_2stepP.jpg',output)

# Projective Distortion Removed, Images written above.

# Removing Affine Distortion

tc=[]
tc.append([104.0,238.0,1.0])   # Pixel Coordinates for Mapping in the Image Plane
tc.append([163.0,254.0,1.0])   # Sqare Diagonal Lines
tc.append([93.0,313.0,1.0])    # Image 1, SQUARE P,Q,R,S Input
tc.append([154.0,330.0,1.0])

pts=np.asarray(tc)

pts[0]=np.dot(H1,pts[0])         # Finding the Points in image result after 1 step is applied
pts[0]=pts[0]/pts[0][2]          # After the Projective Distortion is removed
pts[1]=np.dot(H1,pts[1])         # These points used for remvoing affine Distortion
pts[1]=pts[1]/pts[1][2]
pts[2]=np.dot(H1,pts[2])
pts[2]=pts[2]/pts[2][2]
pts[3]=np.dot(H1,pts[3])
pts[3]=pts[3]/pts[3][2]

Hv=AffineMatrix(pts,pts_image1)
output=ProjectionImage(np.dot(np.linalg.pinv(Hv),H1),image1)
cv2.imwrite('1_2stepA.jpg',output)

td=[]
td.append([290.0,109.0,1.0])      # Pixel Coordinates for Mapping in the Image Plane
td.append([449.0,143.0,1.0])      # Sqare Diagonal Lines
td.append([267.0,267.0,1.0])      # Image 2, SQUARE P,Q,R,S Input
td.append([429.0,294.0,1.0])

pt=np.asarray(td)

pt[0]=np.dot(H2,pt[0])
pt[0]=pt[0]/pt[0][2]       # Finding the Points in image result after 1 step is applied
pt[1]=np.dot(H2,pt[1])     # After the Projective Distortion is removed
pt[1]=pt[1]/pt[1][2]       # These points used for removing affine Distortion
pt[2]=np.dot(H2,pt[2])
pt[2]=pt[2]/pt[2][2]
```

```
pt[3]=np.dot(H2,pt[3])
pt[3]=pt[3]/pt[3][2]

Hu=AffineMatrix(pt,pts_image2)
output=ProjectionImage(np.dot(np.linalg.pinv(Hu),H2),image2)
cv2.imwrite('2_2stepA.jpg',output)
```

5.3 Method 3

```
import numpy as np
import cv2
import math

# Importing the Neccessary Libraries required

# One Step Method for Removing Affine and Projective Distortion

def onestep(arrays1,arrays2):

    l1=np.cross(arrays2[0],arrays2[1])    # Choosing 5 sets of orthogonal lines
    m1=np.cross(arrays2[1],arrays2[3])
    l1=l1/max(l1)
    m1=m1/max(m1)

    l2=np.cross(arrays2[1],arrays2[3])    # For Removing Projective distortion
    m2=np.cross(arrays2[3],arrays2[2])
    l2=l2/max(l2)
    m2=m2/max(m2)

    l3=np.cross(arrays2[3],arrays2[2])
    m3=np.cross(arrays2[2],arrays2[0])
    l3=l3/max(l3)
    m3=m3/max(m3)

    l4=np.cross(arrays2[2],arrays2[0])
    m4=np.cross(arrays2[0],arrays2[1])
    l4=l4/max(l4)
    m4=m4/max(m4)

    l5=np.cross(arrays1[0],arrays1[3])     # For Removing Affine distortion
    m5=np.cross(arrays1[1],arrays1[2])     # Diagonal Lines
    l5=l5/max(l5)
    m5=m5/max(m5)

    ta=[]
    tb=[]         # From the Theory and Mathematical Equations for filling up Matrices

    ta.append([l1[0]*m1[0],(l1[0]*m1[1]+l1[1]*m1[0])/2,l1[1]*m1[1],
    (l1[0]*m1[2]+l1[2]*m1[0])/2,(l1[1]*m1[2]+l1[2]*m1[1])/2])
    tb.append([-l1[2]*m1[2]])

    ta.append([l2[0]*m2[0],(l2[0]*m2[1]+l2[1]*m2[0])/2,l2[1]*m2[1],
    (l2[0]*m2[2]+l2[2]*m2[0])/2,(l2[1]*m2[2]+l2[2]*m2[1])/2])
    tb.append([-l2[2]*m2[2]])

    ta.append([l3[0]*m3[0],(l3[0]*m3[1]+l3[1]*m3[0])/2,l3[1]*m3[1],
    (l3[0]*m3[2]+l3[2]*m3[0])/2,(l3[1]*m3[2]+l3[2]*m3[1])/2])
    tb.append([-l3[2]*m3[2]])

    ta.append([l4[0]*m4[0],(l4[0]*m4[1]+l4[1]*m4[0])/2,l4[1]*m4[1],
    (l4[0]*m4[2]+l4[2]*m4[0])/2,(l4[1]*m4[2]+l4[2]*m4[1])/2])
    tb.append([-l4[2]*m4[2]])

    ta.append([l5[0]*m5[0],(l5[0]*m5[1]+l5[1]*m5[0])/2,l5[1]*m5[1],
    (l5[0]*m5[2]+l5[2]*m5[0])/2,(l5[1]*m5[2]+l5[2]*m5[1])/2])
    tb.append([-l5[2]*m5[2]])

    A=np.asarray(ta)
    b=np.asarray(tb)

    tmp=np.dot(np.linalg.pinv(A),b)  # Computing Matrix for C inf'
    tmp=tmp/np.max(tmp)              # Normalize Coefficients - Important

    S=np.zeros((2,2))         # Computing the S Matrix with reference to Theory
    S[0][0]=tmp[0]
    S[0][1]=tmp[1]/2
    S[1][0]=tmp[1]/2
    S[1][1]=tmp[2]

    u,s,vh=np.linalg.svd(S)          # SVD of the S Matrix

    s1=np.diag(s)

    D=np.sqrt(s1)
    K=np.dot(np.dot(u,D),u.transpose())     # Computing Matrix A

    tmp1=np.array([tmp[3]/2,tmp[4]/2])

    v=np.dot(np.linalg.pinv(K),tmp1)        # Computing Vector V

    H=np.zeros((3,3))
    H[2][2]=1
    H[0][0]=K[0][0]              # Computing Matrix H
    H[0][1]=K[0][1]
    H[1][0]=K[1][0]
    H[1][1]=K[1][1]
    H[2][0]=v[0]
    H[2][1]=v[1]


    return H

def WeightedAverageRGBPixelValue(pt, img):  # Applying the Weighted Average Method

    x1=int(math.floor(pt[0]))
    x2=int(math.ceil(pt[0]))    # Used in Weighted Average Pixel Computation
    y1=int(math.floor(pt[1]))
    y2=int(math.ceil(pt[1]))

    Wp=1/np.linalg.norm(np.array([pt[0]-x1,pt[1]-y1]))
    Wq=1/np.linalg.norm(np.array([pt[0]-x1,pt[1]-y2]))    # Weights
    Wr=1/np.linalg.norm(np.array([pt[0]-x2,pt[1]-y1]))
    Ws=1/np.linalg.norm(np.array([pt[0]-x2,pt[1]-y2]))

    pixel_value = (Wp*img[y1][x1] + Wq*img[y2][x1] + Wr*img[y1][x2] + Ws*img[y2][x2])
    /(Wp+Wq+Wr+Ws)

    return pixel_value     # Return Pixel Value

def ProjectionImage(H,world_plane_img):

    ImgP=np.asarray([0.0,0.0,1.0])      # Image Plane Coordinates
    ImgQ=np.asarray([float(np.shape(world_plane_img)[1])-1.0,0.0,1.0])
    ImgR=np.asarray([0.0,float(np.shape(world_plane_img)[0])-1.0,1.0])
    ImgS=np.asarray([float(np.shape(world_plane_img)[1]-1.0),
    float(np.shape(world_plane_img)[0])-1.0,1.0])


    WorldA=np.dot(H,ImgP)
    WorldA=WorldA/WorldA[2]    # Computing World Coordinates from Image Plane
    WorldB=np.dot(H,ImgQ)     # Using the Homography Matrix H
    WorldB=WorldB/WorldB[2]     # Finding corner points in image and mapping to Real World
    WorldC=np.dot(H,ImgR)
    WorldC=WorldC/WorldC[2]
    WorldD=np.dot(H,ImgS)
    WorldD=WorldD/WorldD[2]
                                # Image Boundaries
    xmin = int(math.floor(min([WorldA[0],WorldB[0],WorldC[0],WorldD[0]])))
    xmax = int(math.ceil(max([WorldA[0],WorldB[0],WorldC[0],WorldD[0]])))
    ymin = int(math.floor(min([WorldA[1],WorldB[1],WorldC[1],WorldD[1]])))
    ymax = int(math.ceil(max([WorldA[1],WorldB[1],WorldC[1],WorldD[1]])))

    yLength=ymax-ymin           # Computing the Shape/Size of the output images
    xLength=xmax-xmin
                                # Applying Projection from Image to World

    src_img=np.zeros((yLength,xLength,3))
    Hn=np.linalg.pinv(H)
    Hn=Hn/Hn[2][2]


    for i in range(0,yLength):
        for j in range(0,xLength):
            tmp=np.array([j+xmin,i+ymin,1.0])
            xp=np.array(np.dot(Hn,tmp))         # Fitting the Image to World
            xp=xp/xp[2]
            if((xp[0]>0)and(xp[0]<world_plane_img.shape[1]-1)
            and(xp[1]>0)and(xp[1]<world_plane_img.shape[0]-1)):
                src_img[i][j]=WeightedAverageRGBPixelValue(xp,world_plane_img)

                                # Returning Output Image
    output_img = src_img
    return output_img



image1=cv2.imread("1.jpg")      # Reading Image Input
image2=cv2.imread("2.jpg")

t1=[]
t2=[]

t1.append([104.0,238.0,1.0])
t1.append([163.0,254.0,1.0])   # Pixel Coordinates for Mapping in the Image Plane
t1.append([93.0,313.0,1.0])    # Rectangular Perpendicualr Lines
t1.append([154.0,330.0,1.0])   # Image 1 Input P,Q,R,S


pts_image1=np.asarray(t1)

t2.append([290.0,109.0,1.0])
t2.append([449.0,143.0,1.0])    # Pixel Coordinates for Mapping in the Image Plane
t2.append([254.0,367.0,1.0])    # Rectangular Perpendicualr Lines
t2.append([417.0,383.0,1.0])    # Image 2 Input P,Q,R,S

pts_image2=np.asarray(t2)

tc=[]
tc.append([104.0,238.0,1.0])   # Pixel Coordinates for Mapping in the Image Plane
tc.append([163.0,254.0,1.0])   # Sqare Diagonal Lines
tc.append([93.0,313.0,1.0])    # Image 1 Square Input P,Q,R,S
tc.append([154.0,330.0,1.0])
pts=np.asarray(tc)
```

```
td=[]
td.append([290.0,109.0,1.0])    # Pixel Coordinates for Mapping in the Image Plane
td.append([449.0,143.0,1.0])    # Sqare Diagonal Lines
td.append([267.0,267.0,1.0])    # Image 2 Square Input P,Q,R,S
td.append([429.0,294.0,1.0])

pt=np.asarray(td)


H1=onestep(pts,pts_image1)    # Computing H Matrix
output=ProjectionImage(np.linalg.pinv(H1),image1)
# Finding the Output projection onto World Plane
cv2.imwrite('1_1step.jpg',output)

H2=onestep(pt,pts_image2)
output=ProjectionImage(np.linalg.pinv(H2),image2)
cv2.imwrite('2_1step.jpg',output)
```

THE END