ECE661: Homework 3

Rahul Deshmukh
deshmuk5@purdue.edu
PUID: 0030004932

September 19, 2022

1 Logic and Methodology

For this Homework we have to remove projective and affine distortion using three different approaches:

a) Using Point-to-Point Correspondences

b) Using Two-step Method

c) Using One-step Method

1.1 Point-to-Point Correspondences

For this approach we need to estimate the Homography using set of points on the image plane along with their world plane coordinates.
We have the equation $H * x = x'$; where H is the homography, x is the homogeneous coordinate of a point on world plane and x' is the homogeneous coordinate of a point in image plane.

Now, we need to find the elements of the H,$3x3$ matrix. Also, as any other homography $k*H$ is equivalent to H and thus the information given by H is only in the ratios of the elements. Therefore, We can assume the $(3,3)$ index of H as 1, and will only need to find 8 other elements of H namely $h_{11}, h_{12}, h_{13}, h_{21}, h_{22}, h_{23}, h_{31}$ & h_{32}.

For any point $x = [x_1, y_1, 1]^T$ we have the equation:

$$H * x = x'$$

$$\Rightarrow \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \end{bmatrix}$$

Thus we get:

$$h_{11}x_1 + h_{12}y_1 + h_{13} = x'_1$$
$$h_{21}x_1 + h_{22}y_1 + h_{23} = x'_2$$
$$h_{31}x_1 + h_{32}y_1 + 1 = x'_3$$

Now in the above equation we have the information of only the $\mathbb{R}^2$ coordinates. That is $x' = x'_1/x'_3$ and $y' = x'_2/x'_3$. Therefore, in order to make the right hand side as a vector of known quantities we need to divide the matrix equation by the scalar x'_3

Thus we get:

$$\frac{1}{(h_{31}x_1 + h_{32}y_1 + 1)} * \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} x_1' \\ y_1' \\ 1 \end{bmatrix}$$

Writing individual equation we get:

$$x_1' = \frac{(h_{11}x_1 + h_{12}y_1 + h_{13})}{(h_{31}x_1 + h_{32}y_1 + 1)}$$

$$\Rightarrow h_{11}x_1 + h_{12}y_1 + h_{13} - h_{31}x_1x_1' - h_{32}y_1x_1' = x_1'$$

$$y_1' = \frac{(h_{21}x_1 + h_{22}y_1 + h_{23})}{(h_{31}x_1 + h_{32}y_1 + 1)}$$

$$\Rightarrow h_{21}x_1 + h_{22}y_1 + h_{23} - h_{31}x_1y_1' - h_{32}y_1y_1' = y_1'$$

Note that, in the above equations we need to solve h_{ij}, therefore we will need at-least 4 more points in order to get at-least 8 equations to solve the 8 unknowns.

We can write out the system of equations in the form of $A\vec{x} = \vec{b}$. Also, if we order our vectors $\vec{x}$ and $\vec{b}$ in a particular order then we get a block Matrix for A, which will be suited in the future when we will have more than 4 points.

Let the four points in the World Plane be denoted by $P = [P_1, P_2]^T$,$Q = [Q_1, Q_2]^T$,$R = [R_1, R_2]^T$ & $S = [S_1, S_2]^T$
and those in Image Plane be denoted by $P' = [P_1', P_2']^T$,$Q' = [Q_1', Q_2']^T$,$R' = [R_1', R_2']^T$ & $S' = [S_1', S_2']^T$

Taking $\vec{x} = [h_{11}, h_{12}, h_{13}, h_{21}, h_{22}, h_{23}, h_{31}, h_{32}]^T$ and $\vec{b} = [P_1, Q_1, R_1, S_1, P_2, Q_2, R_2, S_2]^T$ we get the Matrix A as:

$$A = \begin{bmatrix} P_1 & P_2 & 1 & 0 & 0 & 0 & -P_1'P_1 & -P_1'P_2 \\ Q_1 & Q_2 & 1 & 0 & 0 & 0 & -Q_1'Q_1 & -Q_1'Q_2 \\ R_1 & R_2 & 1 & 0 & 0 & 0 & -R_1'R_1 & -R_1'R_2 \\ S_1 & S_2 & 1 & 0 & 0 & 0 & -S_1'S_1 & -S_1'S_2 \\ 0 & 0 & 0 & P_1 & P_2 & 1 & -P_2'P_1 & -P_2'P_2 \\ 0 & 0 & 0 & Q_1 & Q_2 & 1 & -Q_2'Q_1 & -Q_2'Q_2 \\ 0 & 0 & 0 & R_1 & R_2 & 1 & -R_2'R_1 & -R_2'R_2 \\ 0 & 0 & 0 & S_1 & S_2 & 1 & -S_2'S_1 & -S_2'S_2 \end{bmatrix}$$

Clearly the above matrix can be constructed using basic vectorization and if we have more points then those can be accommodated easily.

The function written out in Python for this task is:

```
# Homograhpy using points
def HMat_Pts(world,img):
    """
    Returns a homography mat such that H*world=img and H[2,2]=1 ie push forward
    img is on destination Image
    world is on source Image
    img,world=np.array([[x1,y1],[x2,y2],[...],[...]])
    """
    n=np.shape(world)[0]
    m=np.shape(world)[1]
    A=np.zeros((2*n,8))
    b=np.reshape(img.T,n*m)

    block1=np.zeros((n,m+1))
    block1[:,-1]=np.ones((1,n))
    block1[:,:2]=world[:,:]
    A[:n,:3]=block1
    A[n:2*n,3:6]=block1
```

```
    block2=np.multiply(world.T,-1*img[:,0])
    block3=np.multiply(world.T,-1*img[:,1])

    A[:n,6:8]=block2.T
    A[n:2*n,6:8]=block3.T
    #solve for h using psuedo inverse
    h=(np.linalg.inv((A.T)@A)@A.T)@b
    h=np.concatenate([h,[1]])
    H=np.reshape(h,(3,3))
    return(H)
```

1.2 Two Step method

As the name suggests, in this method we correct the distortion in the image in two steps:

i) Pure Projective correction using vanishing line

ii) Affine correction using perpendicular lines

1.2.1 Step-1: Projective Correction using Vanishing Line

In this step we estimate the vanishing line using a set of parallel lines and then use the parameters of the vanishing line to construct the homography.

Given, a line l_1 & m_1 are parallel in the world plane. Then in the Image plane the transformed lines will intersect at a finite point, i.e. will no longer remain parallel. This Point of intersection of the lines is called as the vanishing point. We then estimate another vanishing point using another set of parallel lines l_2 & m_2.

Taking the cross product of the two vanishing points we get the parameters of the line crossing these two points. This line is called as the Vanishing line in the image plane. The same line in the world plane is called as the l_∞. The Homography that would correct the pure projective distortion is given as:

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

where l_1,l_2 & l_3 are the Homogeneous coordinates of the the vanishing line.

Note: The direction of the above homography is such it pulls back the projective distortion point to pure affine distortion domain.

The Python function which estimates the above homography is as follows:

```
#Homography for Projective Correction using vanishing Line vectorized
def Hmat_ProjectiveCorrection(imPts):
    """
    Input:
        imPts=[[X,Y],[X,Y],..] min 8 points in the order PQRSTUVW in the image plane
        sets of 4 points give 2 parallel lines in world plane
    Output: Returns the H for Projective Correction using concept of vanishing line
    such that H*im_pt=src_pt or (Inv(H)^T).VL=L_inf
    """
    N=int(np.shape(imPts)[0]/4) #number of vanishing point
    #convert imPts to HC
    im_pts_hc=np.ones((np.shape(imPts)[0],np.shape(imPts)[1]+1))
    im_pts_hc[:,:-1]=imPts
    #find vanishing points
```

```
L1=np.zeros((N,3))#line p1p2
L2=np.zeros((N,3))#line p3p4
VP=np.zeros((N,3))#intersection l1 l2
for i in range(N):
    L1[i,:]=np.cross(im_pts_hc[4*i,:],im_pts_hc[4*i+1,:])
    L2[i,:]=np.cross(im_pts_hc[4*i+2,:],im_pts_hc[4*i+3,:])
    VP[i,:]=np.cross(L1[i,:],L2[i,:])
#hc vector of vanishing line
VL=np.cross(VP[0,:],VP[1,:])
#normalizing VL
VL=VL/VL[-1]
print('The Vanishing Line is: '+str(VL))
H=np.identity(3)
H[-1,:]=VL
return(H)
```

1.2.2 Step-2: Affine correction using perpendicular lines

After removing the pure projective distortion we are left with pure affine distortion. In order to estimate this homography we make use of the concept of angle between two lines. For any line pair l'_1 and m'_1 in the image plane, the angle (θ) formed by these lines in is given as:

$$cos(\theta) = \frac{{l'_1}^T {C'_\infty}^* m'_1}{({l'_1}^T {C'_\infty}^* l'_1) * ({m'_1}^T {C'_\infty}^* m'_1)} = \frac{l_1^T C^*_\infty m_1}{(l_1^T C^*_\infty l_1) * (m_1^T C^*_\infty m_1)}$$

where the lines l_1 & m_1 are the corresponding lines in the World Plane and ${C'_\infty}^*$,C^*_∞ are given as:

$${C'_\infty}^* = \begin{bmatrix} a & b/2 & d/2 \\ b/2 & c & e/2 \\ d/2 & e/2 & f \end{bmatrix}, C^*_\infty = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

If we pick perpendicular line pairs l_1 & m_1 in the world plane, then the $\theta = 90$ and $cos(\theta) = 0$. Thus we get the equation:

$${l'_1}^T {C'_\infty}^* m'_1 = 0 \tag{1}$$

Also,

$${C'_\infty}^* = H C^*_\infty H^T$$

where H is the forward mapping which pushes forward any point in the world plane to affine domain and is given as:

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

$$\Rightarrow {C'_\infty}^* = \begin{bmatrix} A & t \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} I & 0 \\ 0^T & 0 \end{bmatrix} \begin{bmatrix} A^T & 0 \\ t^T & 1 \end{bmatrix}$$

$$\Rightarrow {C'_\infty}^* = \begin{bmatrix} AA^T & 0 \\ 0^T & 0 \end{bmatrix}$$

Using the above expression in equation (1) we get:

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

Note: When writing out individual equations for the above equation we will loose the third homogeneous coordinate information and in doing so we will not obtain the correct set of equations. Therefore we scale the two line vectors with their third coordinate.

$$\Rightarrow \begin{bmatrix} l'_1/l'_3 & l'_2/l'_3 \end{bmatrix} AA^T \begin{bmatrix} m'_1/m'_3 \\ m'_2/m'_3 \end{bmatrix} = 0 \tag{2}$$

In the above equation the matrix $S = AA^T$ is a symmetric matrix and can be written out as:

$$S = \begin{bmatrix} s_{11} & s_{12} \\ s_{12} & s_{22} \end{bmatrix}$$

Also, as the information in S is in the ratios only, therefore we can assume $s_{22} = 1$ and then we need to solve for only two unknowns s_{11} & s_{12}. Thus we need two sets of **different** perpendicular line pairs.

Note: While solving for s_{11} & s_{12} we make use of only $[l'_1/l'_3, l'_2/l'_3]$ which is essentially the direction vector of the line l'. Therefore the sets of lines chosen for solving S should have different direction vectors. For instance, we cannot choose the pairs from edges of a rectangle. However if we can identify a square in the image then the two sets can consist of one edge pair and another as diagonal pair.

Now writing out the equation obtained from equation (2):

Let $l_1 = l'_1/l'_3$ and $l_2 = l'_2/l'_3$ and similarly for m

$$\begin{bmatrix} l_1 m_1 & (l_1 m_2 + l_2 m_1) \end{bmatrix} \begin{bmatrix} s_{11} \\ s_{12} \end{bmatrix} = \begin{bmatrix} -l_2 m_2 \end{bmatrix}$$

We get the above equation for one line pair and can obtain a similar equation for another line pair, thus obtaining the system $As = b$ where $s = [s_{11}, s_{12}]$. This system can be easily vectorized so as to include more than 2 perpendicular line pairs.

We can then use pseudo inverse of A to solve for s:

$$s = (A^T A)^{-1} A^T b$$

Now that we have estimated S, we need to estimate 'A part' of the homography using the information that $S = AA^t$. We make use of Singular Value Decomposition which is same as eigen-value decomposition for this case as S is a square matrix. We get the following:

$$S = U \Sigma V$$

$$A = V \sqrt{(\Sigma)} V^T$$

Assuming the Translation component of the homography as the **0** vector we can obtain the homography for Step-2.

Note: When using the Two-Step method the order of homography is important. We first correct the pure projective distortion and then correct the affine distortion.

The Python function which estimates the above homography is as follows:

```
#Homography for Affine Correction using perpendicular pairs
def Hmat_AffineCorrection(imPts,H_PC):
    """
    Input:
        imPts=[[X,Y],[X,Y],..] min 8 points in the order PQRSTUVW in the image plane
        Note: imPts should have projective distortion as well
```

```
    H_PC = 3x3 Homography that corrects Projective distortion st H_PC.im_pt=affine_pt ie
        pull-back
    Note: perpendicular line pair chosen are (PQ,RS) (TU,VW) and the pairs should have
        different directions
Output:purely affine H =[[A,0],[0,1]] : A is 2x2
such that H*src_pts=im_pt
"""
N=int(np.shape(imPts)[0]/4)# number of line pairs
#convert imPts to HC
im_pts_hc=np.ones((np.shape(imPts)[0],np.shape(imPts)[1]+1))
im_pts_hc[:,:-1]=imPts
#remove Projective distortion
im_pts_hc=H_PC@im_pts_hc.T
im_pts_hc=im_pts_hc.T
im_pts_hc=np.linalg.inv(np.diag(im_pts_hc[:,-1]))@im_pts_hc
#find perpendicular lines
L=np.zeros((N,3));M=np.zeros((N,3))
for i in range(N):
    L[i,:]=np.cross(im_pts_hc[4*i,:],im_pts_hc[4*i+1,:])
    M[i,:]=np.cross(im_pts_hc[4*i+2,:],im_pts_hc[4*i+3,:])
#normalize lines using third hc coord: imp as info of l1 and l2 is only taken for
    further step
L=np.linalg.inv(np.diag(L[:,-1]))@L
M=np.linalg.inv(np.diag(M[:,-1]))@M
L=L[:,:-1]
M=M[:,:-1]
M_reshape=np.reshape(M,(N*2))
#fill A and b
A=np.zeros((N,2)) # coeff matrix
b=np.zeros((N,1)) # RHS
temp=np.multiply(L,M)
A[:,0]=temp[:,0]
b=-1*temp[:,-1]
# permutation Matrix
P=[[0,1],[1,0]]
A[:,1]=L@np.kron(np.ones(N),P)@M_reshape
#solve for S using pseudo inverse
S=(np.linalg.inv((A.T)@A)@A.T)@b
S=np.array([[S[0],S[1]],
            [S[1],1]])
#find A using SVD
u,d,v= np.linalg.svd(S)
A=v@np.diag(np.sqrt(d))@v.T
# construct H, assuming t=0
H=np.identity(3)
H[:2,:2]=A
return(H)
```

1.3 One Step method

In this method we directly estimate the matrix ${C'_\infty}^*$ using perpendicular line pairs.
We have equation (1) and we have to estimate the coefficients of ${C'_\infty}^*$:

$$l_1'^T {C'_\infty}^* m_1' = 0$$

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

As ${C'_\infty}^*$ is a symmetric matrix of size 3x3, thus we only need to estimate 6 unknowns. Also, as the information stored in ${C'_\infty}^*$ is only in the ratios of the elements, Therefore we can assume $f = 1$, and need to solve for

only 5 unknowns $x = [a, c, b/2, d/2, e/2]$. thus we would need at-least 5 pairs of perpendicular lines to solve x.

Writing out the system of equation for one line pair l & m (here both l & m are in image plane, for ease of reading we have dropped the ' in superscripts) we get:

$$\begin{bmatrix} l_1m_1 & l_2m_2 & (l_1m_2 + l_2m_1) & (l_1m_3 + l_3m1) & (l_2m_3 + l_3m_2) \end{bmatrix} \begin{bmatrix} a \\ c \\ b/2 \\ d/2 \\ e/2 \end{bmatrix} = \begin{bmatrix} -l_3m_3 \end{bmatrix}$$

Similarly we can obtain the equation for other line pairs. We can easily vectorize the above system of equation as $Ax = b$, by using permutation matrices and element-wise multiplication, and can use the methodology for more than 5 pairs of perpendicular lines. Thus solving for x using the pseudo inverse of A.

$$x = (A^T A)^{-1} A^T b$$

Now, we can construct ${C'_\infty}^*$ using x. We now need to estimate the Homography H. We know that ${C'_\infty}^* = H C^*_\infty H^T$. Thus if we take the SVD of ${C'_\infty}^*$ we should obtain ${C'_\infty}^* = U C^*_\infty U^T$ and U is our solution, i.e. $H = U$.

The Python function which estimates the homography in one step is as follows:

```
# H using one step
def HMat_OneStep(imPts):
    """
    Input:
        imPts: 20x2 [P1,P2,P3,P4,...] array with information of 20 points in image plane
        can be more than 20 also but always in multiples of 4
        pts ordered such that line P1P2 and line P3P4 is perpendicular
        therefore we have at-least 5 sets of perpendicular lines
    Output:
        H: 3x3 homography matrix which will correct both
        affine and projective distortion in one step
        H*world_pt = image_pt : push fwd
    """
    N=int(np.shape(imPts)[0]/4)# number of line pairs
    #convert imPts to HC
    im_pts_hc=np.ones((np.shape(imPts)[0],np.shape(imPts)[1]+1))
    im_pts_hc[:,:-1]=imPts
    # find lines
    L=np.zeros((N,3));M=np.zeros((N,3))
    for i in range(N):
        L[i,:]=np.cross(im_pts_hc[4*i,:],im_pts_hc[4*i+1,:])
        M[i,:]=np.cross(im_pts_hc[4*i+2,:],im_pts_hc[4*i+3,:])
    #normalize lines: not really needed  and experimentation confirmed it: also it increases
        the condition number of A
#     L=np.linalg.inv(np.diag(L[:,-1]))@L
#     M=np.linalg.inv(np.diag(M[:,-1]))@M
    M_reshape=np.reshape(M,(N*3))
    # fill out A and b
    A=np.zeros((N,5))
    b=np.zeros((N,1))
    temp=np.multiply(L,M)
    A[:,:2]=temp[:,:-1]  #first two columns with l1m1, l2m2
    b=-1*temp[:,-1] #RHS = -l3m3
    #permutation matrices
    P1=[[0,1,0],[1,0,0],[0,0,0]]
    P2=[[0,0,1],[0,0,0],[1,0,0]]
    P3=[[0,0,0],[0,0,1],[0,1,0]]
    A[:,2]= L@np.kron(np.ones(N),P1)@M_reshape  #l1m2+l2m1
```

```
    A[:,3]= L@np.kron(np.ones(N),P2)@M_reshape  #l1m3+l3m1
    A[:,4]= L@np.kron(np.ones(N),P3)@M_reshape  #l2m3+l3m2
    #check condition number of A
    ua,da,va=np.linalg.svd(A)
    print('d for A is\n')
    print(da)
    print('The condition number of A is\n')
    print(np.linalg.norm(da[0])/np.linalg.norm(da[-1]))
    #solve for coeffs using psuedo inverse
    C=(np.linalg.inv((A.T)@A)@A.T)@b  #C=[a,c,b/2,d/2,e/2]
#     C=C/max(np.abs(C)) #not needed
    C=np.array([[C[0],C[2],C[3]],
        [C[2],C[1],C[4]],
        [C[3],C[4],1]])
#     #find A and v
#     S=C[:2,:2]
#     U,D,V=np.linalg.svd(S)
#     A=V@np.diag(np.sqrt(D))@V.T
#     v=np.linalg.inv(A)@C[:2,-1]
#     # construct H
#     H=np.identity(3)
#     H[:2,:2]=A
#     H[-1,:-1]=v  #assuming t=0
#     find H from C using SVD
    u,d,v=np.linalg.svd(C)
    H=u
    return(H)
```

1.4 Mapping of Image

Now that we have calculated the homography H, we now need to map the image to a canvas on world plane. This means that we need to assign a pixel value to any point on the canvas and the assigned pixel value will be obtained form the image. Also we need to fit the mapped image to a custom canvas size so that we can see the entire mapped image.

The task can be divided into two parts:

a) Transformation of coordinates

b) Interpolation of Pixels

1.4.1 Transformation of coordinates

This task can be best explained using figure 1.

Step-1 We are trying to identify the pixel values on the green canvas in the world plane. For this, we first start out with bringing back the corners of the image from the image plane to the world plane using H^{-1}. The corners of the image in the world plane might get rotated and might not represent a rectangular window, also the image might have translation associated to it. We then construct a rectangular window across the transformed image using max and min of coordinate values of the four corners.

Step-2 As we know the coordinates of points on the canvas$(\hat{x}, \hat{y})$, we need to transform these coordinates to the black window (x, y) within the world plane. This is done using a simple transformation as shown in figure 2 for both the axes.

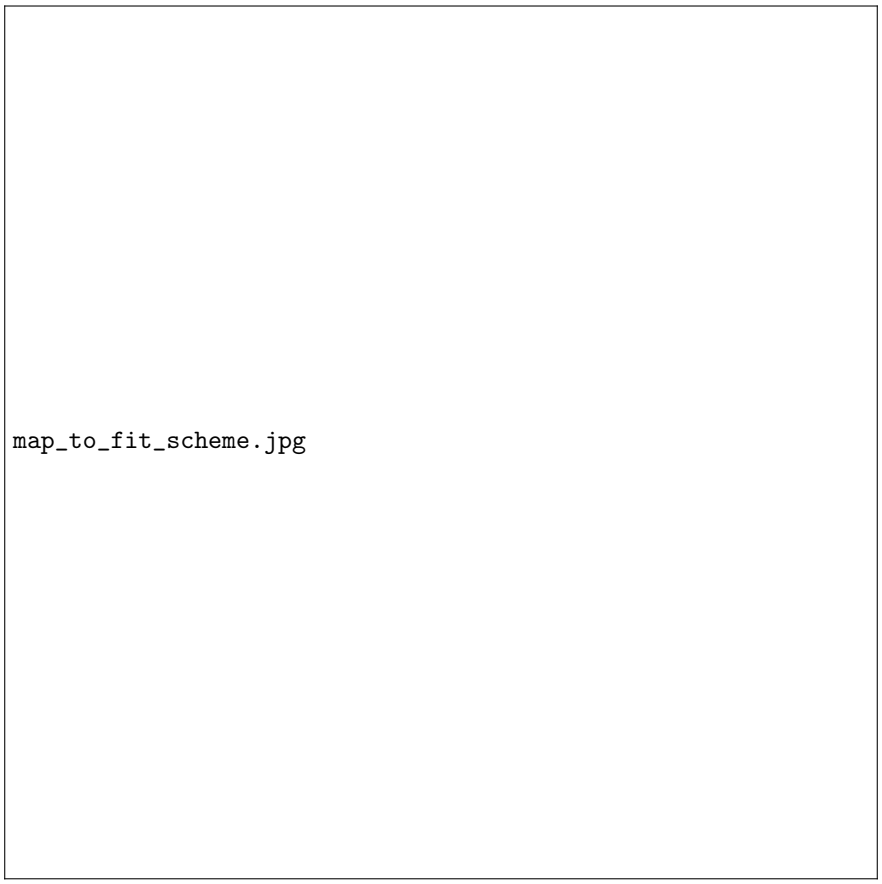

Figure 1: Schematic

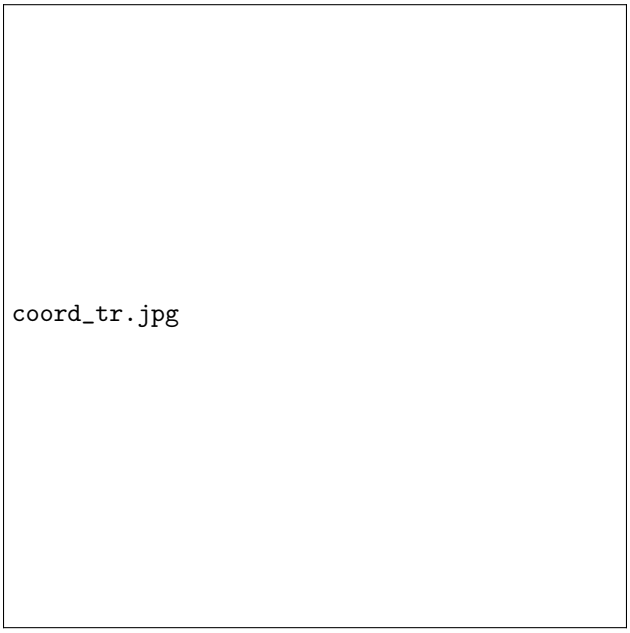

Figure 2: Transformation of coordinates

from the transform shown figure**??** we obtain:

$$\frac{\hat{x} - 0}{x - xmin} = \frac{size(x) - 0}{xmax - xmin}$$
$$\Rightarrow x = \hat{x} * \frac{\Delta_x}{size(x)} + xmin$$

where $\Delta_x = xmax - xmin$, Similarly we can obtain a transform for the y coordinate.

Step-3 Finally, we push forward the point from the black window to the image plane using the homography H. Then we check if the point is inside the image and assign a pixel value using interpolation.

1.4.2 Interpolation of pixels

Any point x' in the image plane when transformed to the world plane, using $x = H^{-1}x'$, will not always have integer coordinates. As we have pixel values only at integer coordinates in any image, therefore we will need to interpolate the pixel value at x from its closest neighbors.

Also, the RGB pixel values will always be an integer in the range 0-255. Thus, the interpolated pixel value at x should conform to this format. Therefore, a interpolation scheme with sum of weights=1 will be best suited here.

Interpolation Scheme using L2 norm and squared L2 norm For any point x, the four closest integer points can simply be constructed using a permutations of floor and ceiling of the coordinates of x. We are using Euclidean distance (or the L2 norm) of the point x to any of the closest point normalized by the sum of distances for all the four points as our interpolation weights.

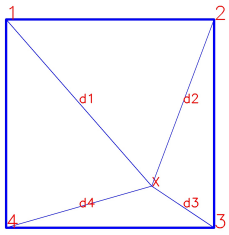

Figure 3: interpolation diagram

As shown from in Figure 3 we can define our interpolation scheme as:

$$c(x) = \frac{d_1 c(1) + d_2 c(2) + d_3 c(3) + d_4 c(4)}{d_1 + d_2 + d_3 + d_4}$$

where, $c(x)$ denotes pixel values at any point x and d_i denotes the distance of point i from the point x. Similarly for squared L2 norm we are just squaring the Euclidean distance

Interpolation Scheme using Bilinear Shape functions Another way to obtain weights is to make use of Bilinear shape functions, a technique used in Finite Elements. The shape functions are defined using a parent element with orthogonal coordinates as ξ and η in the domain $-1 \leq \xi \leq 1$ and $-1 \leq \eta \leq 1$.

The corresponding shape function,N_i, at the point (ξ_i, η_i) is given by:

$$N_i(\xi, \eta) = \frac{(1 + \xi_i \xi) * (1 + \eta_i \eta)}{4} \qquad i = 1, 2, 3, 4$$

We can use these shape functions by converting our problem into parent domain, using a simple transformation of coordinates.

Interpolation Scheme using Round Up and Round Down This method is computationally the least expensive as we are not finding any distances,which involves finding a square root, and neither involves any linear interpolation. In this method we just assign the pixel value at x as that of the point 1 in case of rounding down and point 3 in case of rounding up.(refer to Figure 3 for naming of points)

Further, The pixel value obtained from the above interpolation may have floats as value. Therefore, we need to take floor/ceiling of the values to convert them into integers.

The set of python functions which map the source image onto a destination image is:

```
def mapFitToCanvas(Img,imPts,canvas,H,distype):
    """
    function applies homogrpahy H to source image and fits the resulting matrix to a canvas
    Input:
        Img & canvas: Image matrices
        imPts =np.array([[X,Y],[X Y]...])
        H is a 3x3 matrix: H*world_pt=image_pt ie push forward
        distype= a string L2, sqL2,BiLinear,RoundDown,RoundUp
    Output: Function will return the result Image matrix
    """
    resultImg=np.zeros(np.shape(canvas))
    invH=np.linalg.inv(H)
    #convert imPts to hc
    src_hc=np.ones((3,np.shape(imPts)[0]))
    src_hc[:2,:]=imPts.T
    #find mapping of imPts on canvas
    im_src_hc=invH@src_hc#columns of im_src_hc are images of src_pts
    #divide each column by the value in the last index of that column to get physical
        coordinates
    im_src_hc=np.linalg.inv(np.diag(im_src_hc[-1,:]))@im_src_hc.T
    #now rows of im_src_hc are physical coordinates with third column all 1s
    #find parameters for coordinate transformation within canvas plane
    xmin=min(im_src_hc[:,0])
    xmax=max(im_src_hc[:,0])
    ymin=min(im_src_hc[:,1])
    ymax=max(im_src_hc[:,1])
    deltax=xmax-xmin
    deltay=ymax-ymin
    #now find x and y limits for src image coordiantes
    src_xmin=min(imPts[:,0])
```

```
    src_xmax=max(imPts[:,0])
    src_ymin=min(imPts[:,1])
    src_ymax=max(imPts[:,1])
    # iterate over points in canvas coordinates
    for i in range(np.shape(canvas)[0]):
        for j in range(np.shape(canvas)[1]):
            #x: col index & y:row index , pt on canvas is [j,i,1]
            #transform [j,i] on canvas to the block of interest
            x_tr=xmin+(deltax/np.shape(canvas)[1])*j
            y_tr=ymin+(deltay/np.shape(canvas)[0])*i
            #find corresponding point in src image using invH
            x = np.dot(H,[x_tr,y_tr,1])
            x=np.array([x[0]/x[-1],x[1]/x[-1]])
            #now we have a non-integer point in src Img
            #check if the point lies inside the window in the source image
            if x[0]>src_xmin and x[0]<src_xmax and x[1]>src_ymin and x[1]<src_ymax:
                #find new pixel value
                cx = InterpolatePixel(x,Img,distype)
                #assign new pixel value to destImg
                resultImg[i,j,:]=cx
            else:
                resultImg[i,j,:]=canvas[i,j,:]
    return(resultImg)

# Interpolation of colors from the source image
def InterpolatePixel(x,srcImg,distype):
    """
    Input:
        x= n.array([X,Y]) coordinates of a point
        srcImg=source image object a n,m,3 matrix
        distype= a string L2, sqL2,BiLinear,RoundDown,RoundUp
    Output:
        cx= size(1,3) vector of pixel values with only integer values
    The function will interpolate the pixel values of the nearest neighbours and do a
        rounding
    to get integers
    """
    #finding the nearest interger neighbours
    p1=[int(np.floor(x[0])),int(np.floor(x[1]))]
    p2=[int(np.ceil(x[0])),int(np.floor(x[1]))]
    p3=[int(np.ceil(x[0])),int(np.ceil(x[1]))]
    p4=[int(np.floor(x[0])),int(np.ceil(x[1]))]
    #storing value of pixels at these points
    #x: col index & y:row index
    c1=srcImg[p1[1],p1[0],:]
    c2=srcImg[p1[1],p1[0],:]
    c3=srcImg[p1[1],p1[0],:]
    c4=srcImg[p1[1],p1[0],:]
    C=np.array([c1,c2,c3,c4])
    #storing weights for interpolation
    w=weights(np.array([p1,p2,p3,p4]),x,distype);#should return a np array
    #interpolating
    fcx=C.T@w
    fcx=np.floor(fcx)
    cx=fcx.astype(int)
    return(cx)

def weights(pts,x,distype):
    """
    function will give a list of weights
    Input:  pts = np.array[[X Y],[X Y]...]
    x=np.array[X,Y]
    distype is a string with options L2, sqL2,BiLinear,RoundDown,RoundUp
    output:
        w=[w1,w2,w3,w4]
```

```
"""
w=[]
if distype=='L2':
    #use L2 norm for distance
    for i in range(np.shape(pts)[0]):
        w.append(np.linalg.norm(pts[i,:]-x))
    wsum=sum(w)
    w=w/wsum
    return(w)
elif distype=='sqL2':
    #use squared L2 norm for distance
    for i in range(np.shape(pts)[0]):
        w.append((np.linalg.norm(pts[i,:]-x))**2)
    wsum=sum(w)
    w=w/wsum
    return(w)
elif distype=='BiLinear':
    #using bilinear shape functions as weights
    xmin=pts[0,0]
    xmax=pts[1,0]
    ymin=pts[0,1]
    ymax=pts[3,1]
    xi=2*((x[0]-xmin)/(xmax-xmin))-1
    eta=2*((x[1]-ymin)/(ymax-ymin))-1
    w.append((1-xi)*(1-eta)/4.0)
    w.append((1+xi)*(1-eta)/4.0)
    w.append((1+xi)*(1+eta)/4.0)
    w.append((1-xi)*(1+eta)/4.0)
    return(w)
elif distype=='RoundDown':
    w=[1,0,0,0]
    return(w)
elif distype=='RoundUp':
    w=[0,0,1,0]
    return(w)
```

2 Results

2.1 Results for Image-1

2.1.1 Point-To-Point Correspondences

2.1.2 Results 1(b)

2.2 Results of Task 2

2.2.1 Results 2(a)

2.2.2 Results 2(b)