Australian National University

Computer Lab 3 Report

ENGN**4528** Computer Vision

by Abhinav Pandey

(U6724645)

Task 1 : 3D-2D Camera Calibration

Introduction

Geometric camera calibration, also referred to as camera re-sectioning, estimates the parameters of a lens and image sensor of an image or video camera.[(1)] It involves finding the geometric relationship between 3D world coordinates and their 2D projected positions in the image. You can use these parameters to correct for lens distortion, measure the size of an object in world units, or determine the location of the camera in the scene.[(1)]

In this task we will be using the image labelled as **stereo2012a.jpg**, as shown in Figure 1. The image contains 3 mutually orthogonal faces. The points marked on each face form a regular grid and are all 7cm apart.

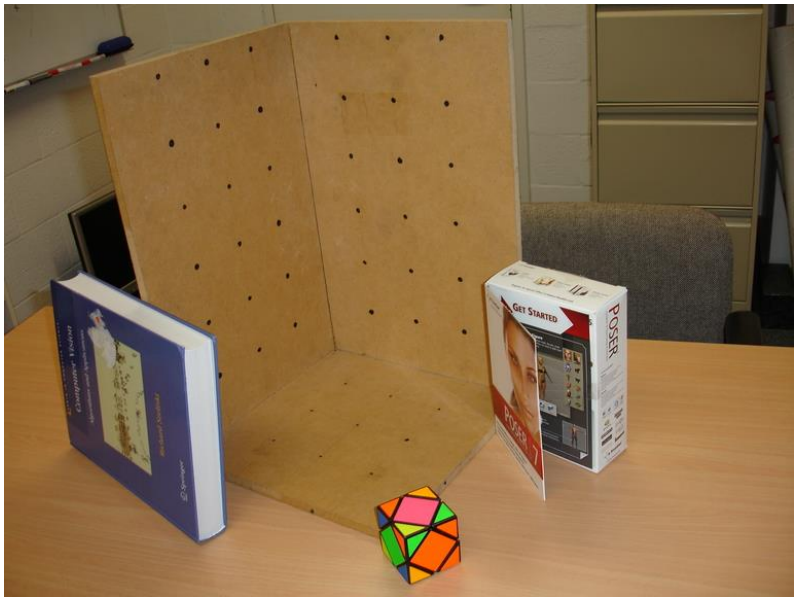

Figure 1. Target Image for Camera Calibration (stereo2012a.jpg)

Our task is to write a function that takes the following inputs and computes the 3 x 4 camera calibration matrix, C, using Direct Linear Transformation (DLT) –

- An image
- uv - N points selected on the image (2D image coordinates)
- XYZ – N real world points corresponding to the 2D image coordinates (3D Real World Coordinates)

where, N is an integer greater than or equal to 6.

Procedure

The process of camera calibration was completed in the following steps :

1. Select 6 (or more) coordinates on the target image.
2. Obtain their corresponding real-world coordinates.
3. Create the over-constrained equation matrix A and solution matrix b
4. Determine camera calibration matrix, C, using least squares estimation

Implementation

1. Select 6 (or more) coordinates on the target image (uv).

We begin by selecting 6 points on the grid of our target image (as shown in Figure 2). We can do so in Python, using the code below:

```
I = Image.open('Left.jpg')

plt.imshow(I)

uv = plt.ginput(6)
```

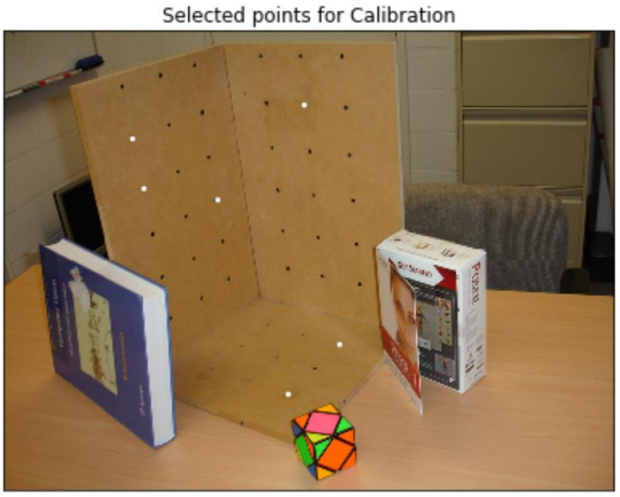

Figure 2. The white dots correspond to the points selected for camera calibration

2. Obtain their corresponding real-world coordinates (XYZ).

We prefer selecting points on the grid because it allows us to easily estimate their real-world positions once we have fixed the origin and X, Y and Z axes (as shown in Figure 3) on the target image.

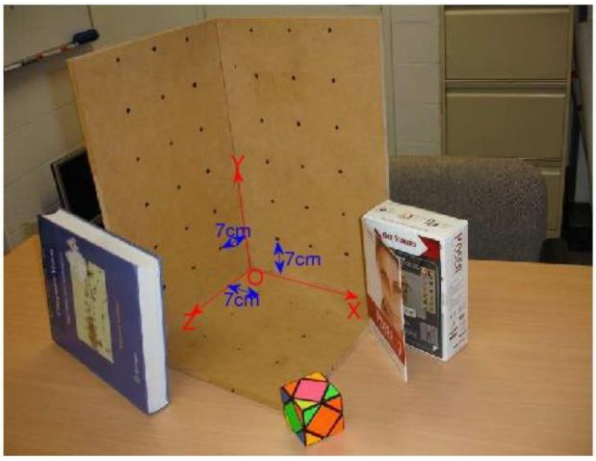

Figure 3. Origin and Reference Axes on our Target Image

The exact values of the selected image coordinates (uv) and their corresponding real-world coordinates (XYZ) can be found in Table 1 in the Appendix.

3. Create an over-constrained equation matrix, A, and solution matrix, b.

Once we have the image coordinates and their corresponding real-world coordinates, the next step to computing the 3 x 4 Camera Calibration matrix is to create an over constrained *2N* x *11* matrix **A** of the form :

$$\mathbf{A} = \begin{bmatrix} X_1 & Y_1 & Z_1 & 1 & 0 & 0 & 0 & 0 & -u_1X_1 & -u_1Y_1 & -u_1Z_1 \\ 0 & 0 & 0 & 0 & X_1 & Y_1 & Z_1 & 1 & -v_1X_1 & -v_1Y_1 & -v_1Z_1 \\ & & & & & & \vdots & & & & \\ X_n & Y_n & Z_n & 1 & 0 & 0 & 0 & 0 & -u_nX_n & -u_nY_n & -u_nZ_n \\ 0 & 0 & 0 & 0 & X_n & Y_n & Z_n & 1 & -v_nX_n & -v_nY_n & -v_nZ_n \end{bmatrix}$$

, where each pair of consecutive rows corresponds to an equation using a single pixel's coordinates (u, v) and its corresponding real-world coordinates (X, Y, Z).

Also, we create a solution matrix, **b**, of shape *2N*. This matrix contains the selected u, v coordinates and is of the form:

$$\mathbf{b} = \begin{bmatrix} u_1 \\ v_1 \\ \vdots \\ u_n \\ v_n \end{bmatrix}$$

This can be done in Python using the code below:

```
# Initialise lists to store the equation and result matrix
A = []
b = []

# Repeat for each real world point (xyz),
# where each element in xyz is of the form [x, y, z, 1]
for i, ele in enumerate(xyz):

    u = uv[i, 0]  # x-coordinate of pixel corresponding to the real world point (ele)
    v = uv[i, 1]  # y-coordinate of pixel corresponding to the real world point (ele)

    eq1 = np.concatenate((ele, [0,0,0,0], -ele*u))[:-1]  # Store the first equation
    eq2 = np.concatenate(([0,0,0,0], ele, -ele*v))[:-1]  # Store the second equation

    # Add the equations to the overconstrained equation matrix, A
    A.append(eq1)
    A.append(eq2)

    # Add pixel coordinates to the solution matrix, b
    b.append(u)
    b.append(v)
```

4. Determine camera calibration matrix, C, using least squares estimation

The final step of calculating our camera calibration matrix is to use the matrices **A** and **b** and estimate the elements of our camera matrix using least squares. Since, there are 11 columns in A, the results of our least-squares estimation will also have 11 elements.

```
# Solve the equations in 2N x 11 matrix, A using solutions from 2N matrix, by using least
 squares method to obtain the first 11 elements of 3 x 4 camera matrix
C = np.linalg.lstsq(A, b, rcond=None)[0]
```

To complete the list of our camera matrix elements we will add 1 as the last element. This is because the camera calibration matrix has 11 degrees of freedom i.e. it has 11 unique elements, and the last element is always 1. Following this, we can reshape our 1-dimensional array into our *3* x *4* camera calibration matrix.

$$\begin{bmatrix} m_{11} \\ m_{12} \\ m_{13} \\ m_{14} \\ m_{21} \\ m_{22} \\ m_{23} \\ m_{24} \\ m_{31} \\ m_{32} \\ m_{33} \\ 1 \end{bmatrix} \longrightarrow \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & 1 \end{bmatrix} = \mathbf{C}$$

We can do this reshaping simply with the following code :

```
# The last element of C is 1
C = np.concatenate((C, [1]))

# Reshape matrix C to dimensions 3 x 4
C = C.reshape((3,4))
```

Ans 3.
Below is the resulting Camera Calibration matrix for our target image :

31.27309	-14.024	-43.0879	323.2746
1.387306	-52.0039	9.243383	334.5393
-0.02424	-0.02297	-0.04074	1

*Table 1. Camera Calibration Matrix, **C***

Now, to evaluate our camera matrix **C**, we will use our camera matrix to project our 3D real-world coordinates onto the image plane and visually check the results and also compute the mean squared distance (MSE) of our projections from the actual 2D coordinates.

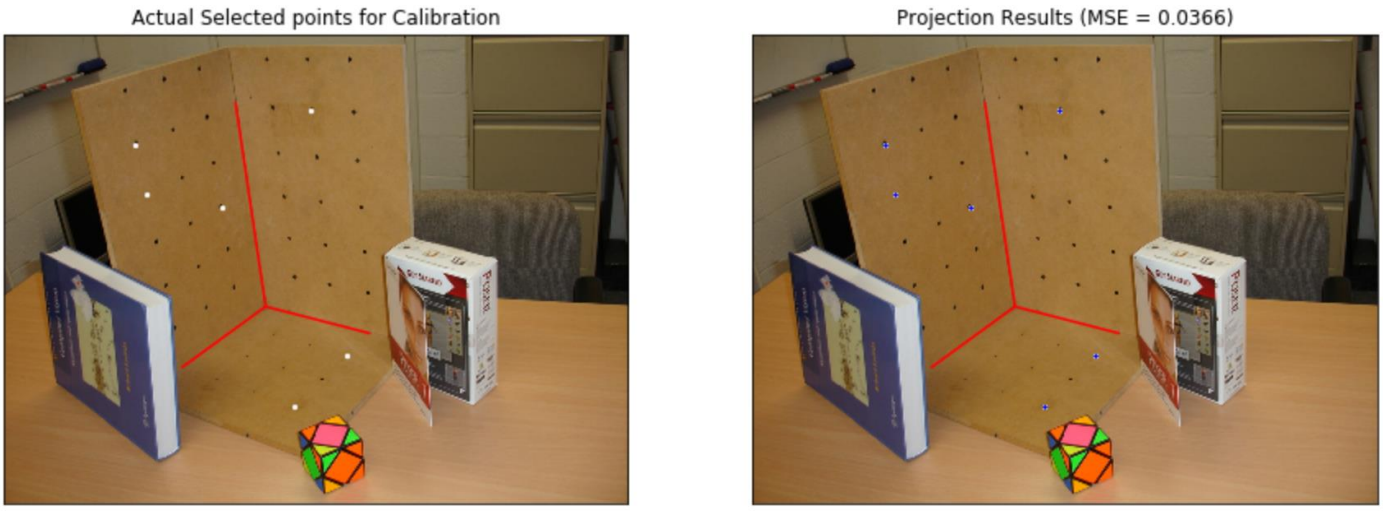

*Figure 4. Target image with selected points (in white) on the left, and the projection results (in blue) using the real-world coordinates and the camera calibration matrix (**C**) on the right. Lines from the origin to the vanishing points in X, Y and Z direction are overlaid on the image (in red). The Mean Squared Error was found to be **0.0366**.*

A visual check confirms that the projected points lie remarkably close to the actual 2D coordinates (u, v).

The Mean Squared Error between the actual coordinates and the projected coordinates is found to be *0.0366*, which further confirms that our camera matrix is a *good approximation* of the **real camera matrix**.

Ans 4.
Since we have accurately determined the camera calibration matrix, we can further decompose it to obtain the intrinsic (**K**) and extrinsic (rotation (**R**) and translation (**t**)) parameters of the camera. The code for the decomposition can be found in the Appendix.

K

930.7032	0.586586	475.467
0	962.3615	282.5658
0	0	1

*Table 2. Intrinsic Parameter Matrix (**K**)*

R

0.872833	-0.06274	-0.48397
0.162481	-0.89777	0.409411
-0.46018	-0.43598	-0.77341

t

119.8925
24.98627
86.38786

*Table 3. And Table 4. Extrinsic Parameter Matrices (**R** and **t**) respectively*

Ans 5.

Using these extrinsic and intrinsic parameters of the camera, we can even determine the focal length of the camera lens and rotation angles (such as the pitch angle which is the angle between the camera's optical axis and the ground-plane). Let us see how we can do that.

a. Determining the focal length of the camera :

The intrinsic parameter matrix **K**, is of the form –

$$\begin{bmatrix} f_x & 0 & 0 \\ s & f_y & 0 \\ c_x & c_y & 1 \end{bmatrix}$$

, where f_x *and* f_y represent the focal length of the camera (measured in pixels) [(1)] in the x and y directions.

Ans 5(a)

Therefore, the focal lengths (f_x and f_y) of our camera are 930.7032 and 962.3615, respectively.

Ideally, f_x and f_y should have equal values, which would represent the true focal length (measured in pixels) of the camera. However, that is not the case here. One explanation for this could be the noise introduced in the coordinates during the selection step.

Since, our focal lengths have different values, it would be fair to assume that the *true* focal length of the camera (measured in pixels) lies in the range [930.7032, 962.3615].

b. Determine the pitch angle of the camera with respect to the X-Z plane in the world coordinate system :

The pitch angle, β, can be determined using the formula below[(2)] –

$$\beta = \tan^{-1}\left(- r_{31} / \sqrt{r_{32}^2 + r_{33}^2} \right)$$

, where r_{31}, r_{32} and r_{33} are elements of the last row of **R** (Table 3) viz. -0.46018, -0.43598 and -0.77341 respectively.

Ans 5(b)

Pitch Angle (in radians) = 0.4782 radians

Pitch Angle (in degrees) = 27.39°

My Calibrate function (in Python)

```
def calibrate(im, XYZ, uv):

    # Concatenate 1 to every real world coordinate : [X Y Z] --> [X Y Z 1]
    XYZ = np.concatenate((XYZ, np.ones((len(XYZ), 1))), axis=1)

    # Initialise lists to store the equation and result matrix
    A = []
    b = []

    # Repeat for each real world point (XYZ)
    # where each element of XYZ is of the form [X Y Z 1]
    for i, ele in enumerate(XYZ):

        u = uv[i, 0]  # Store x-coordinate of pixel corresponding to the real world point (xyz)
        v = uv[i, 1]  # Store y-coordinate of pixel corresponding to the real world point (xyz)

        eq1 = np.concatenate((ele, [0,0,0,0], -ele*u))[:-1]  # Store the first equation
        eq2 = np.concatenate(([0,0,0,0], ele, -ele*v))[:-1]  # Store the second equation

        # Add the equations to the equation matrix, A
        A.append(eq1)
        A.append(eq2)

        # Add pixel coordinates to the solution matrix, b
        b.append(u)
        b.append(v)

    A, b = np.array(A), np.array(b)

    # Solve the equations in 2N x 11 matrix, A using solutions from 2N matrix, by using least
    # squares method to approximately determine the Camera Calibration matrix,
    # C (of length 11)
    C, residuals = np.linalg.lstsq(A, b, rcond=None)[:2]

    # Display the error in satisfying camera calibration matrix constraints
    print("Error in satisfying the camera calibration matrix constraints =
{}".format(residuals[0]))

    # The last element of C is 1
    C = np.concatenate((C, [1]))

    # Reshape matrix C to dimensions 3 x 4
    C = C.reshape((3,4))
```

```
    # Project XYZ onto the 2D image plane
    uv_preds = (C @ XYZ[:,:,None]).reshape((-1,3))
    uv_preds = uv_preds[:,:2] / uv_preds[:,-1, None] # Normalise the coordinates using the last element (u = uw/w)

    # Compute projection error, mean squared error
    mse = np.mean((uv_preds - uv)**2)

    # Load Vanishing Lines and Origin coordinates
    origin_and_vanishing_pts = np.load("Vanishing XYZ.npy")

    # Plotting Code
    plt.imshow(im)
    plt.scatter(uv[:,0], uv[:,1], s=10, marker='o', c='w')
    plt.scatter(uv_preds[:,0], uv_preds[:,1], s=10, marker='x', c='b')
    plt.title("Projection Error (MSE) = %.5f"%mse)

    # Visualize lines from the origin to the vanishing points in the X, Y and Z direction
    color = 'red' # Line color
    plt.plot(origin_and_vanishing_pts[:2,0], origin_and_vanishing_pts[:2,1], c=color)
    plt.plot(origin_and_vanishing_pts[1:3,0], origin_and_vanishing_pts[1:3,1], c=color)
    plt.plot(origin_and_vanishing_pts[[1,3],0], origin_and_vanishing_pts[[1,3],1], c=color)

    plt.xticks([])
    plt.yticks([])
    plt.show()

    return C
```

Task 2 : Two-View DLT based Homography Estimation

A transformation from the projective space P^3 to itself is called homography. A homography is represented by a 3x3 matrix with 8 degree of freedom (scale invariant).

The homography matrix is of the form –

$$\begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix}$$

In this task we will be estimating a 3 x 3 homography matrix, to rectify a large perspective distortion as seen in Figure 5. In simpler words, we will attempt to transform the image on the left to produce an image like the one on the right.

Figure 5. Perspective Distorted Image (left) and Target Perspective Image (right)

Our task is to write a function that takes the following inputs and computes the 3 x 3 homography matrix, **H**, using Direct Linear Transformation (DLT) –

- U2Trans : x-coordinates of the selected points on the distorted image (left image)
- V2Trans : y-coordinates of the selected points on the distorted image (left image)
- UBase : x-coordinates of the selected points on the target image (right image)
- VBase : y-coordinates of the selected points on the target image (right image)

The number of selected points must be greater than 4 for estimating the homography matrix.

Procedure

The process of homography estimation was completed in the following steps :

1. Select 4 (or more) coordinates on the distorted image and their corresponding coordinates on the target image.
2. Create an over-constrained matrix A.
3. Perform Singular Value Decomposition (SVD) on matrix A.
4. Obtain the elements of homography matrix, **H** , using the last column of matrix V^T (obtained from SVD).
5. Divide the entire matrix, **H** by the last diagonal element of **H.**

Implementation

1. Select 4 (or more) coordinates on the distorted image.

We will begin by selecting 6 points (representing the position in the real world) on both, left and right images.

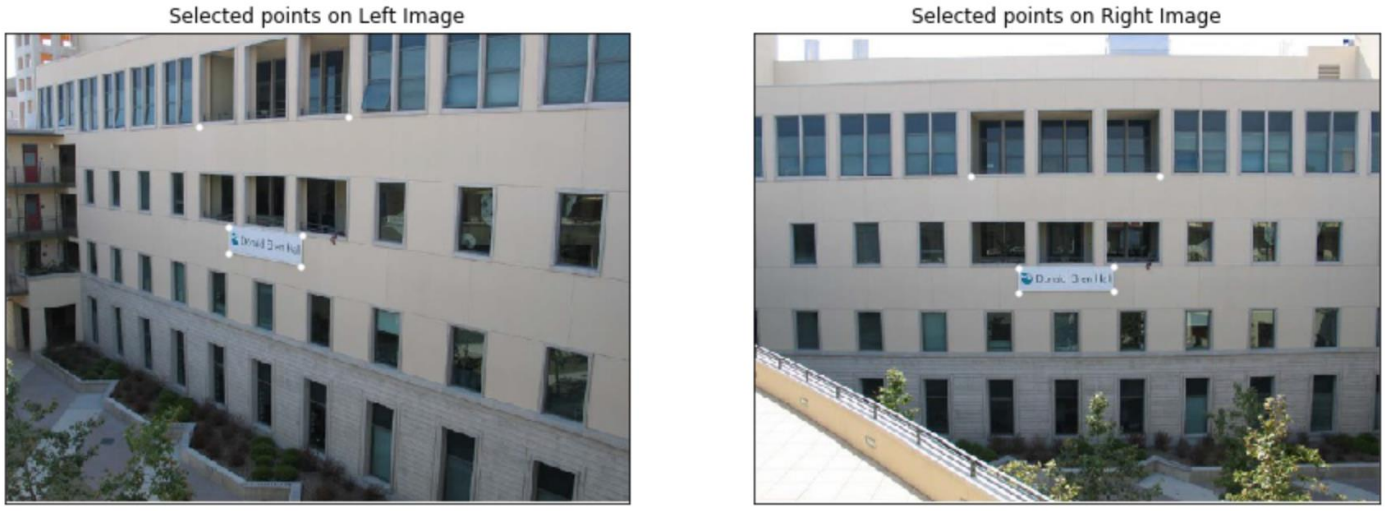

Figure 6. The selected coordinates for Homography estimation (as white dots)

2. Create the over-constrained equation matrix A.

Once we have selected our coordinates, the next step is to compile an over-constrained system of equations in our equation matrix, **A**.

The left and right image coordinates must be used, and the following two equations should be added to **A** for each coordinate pair -

$$A = \begin{pmatrix} -x & -y & -1 & 0 & 0 & 0 & ux & uy & u \\ 0 & 0 & 0 & -x & -y & -1 & vx & vy & v \end{pmatrix}$$

, where x, y, u, and v correspond to elements of u2Trans, v2Trans, uBase and vBase, respectively.

This can be done in python using the following code :

```
# Initialise A as an empty list
A = []

# Repeat for each selected coordinate pair of the perspective distorted image,
# where each element in the coordinate list is of the form [x y 1]
for i, ele in enumerate(trans_uv):

    # Corresponding x, y coordinates on target image
    u = uBase[i]
    v = vBase[i]

    eq1 = np.concatenate(([0,0,0], -ele, ele * v))    # First equation
    eq2 = np.concatenate(( -ele, [0,0,0], ele * u))   # Second equation

    # Add equations to A
    A.append(eq1)
    A.append(eq2)
```

3. Perform Singular Value Decomposition (SVD) on matrix A.

The next step in estimating the homography matrix, **H** using Direct Linear Transformation (DLT) is to perform Singular Value Decomposition on the over-constrained system of equations, **A**.

```
# Decompose A using SVD and store only the last orthogonal component, V
_ , _ , V = np.linalg.svd(A)
```

Note : The Singular Value Decomposition of a matrix yields a total of 3 component matrices. However, we require only the last component matrix, **V** for homography estimation.

4. Obtain the elements of homography matrix, H.

The next step of our homography estimation process is to use the matrix, **V**, we computed in the previous step to obtain the elements of our Homography matrix, **H**.

According to the DLT algorithm the last column of **V** contains the elements of **H**, in the form –

$$\begin{pmatrix} h_1 & h_2 & h_3 & h_4 & h_5 & h_6 & h_7 & h_8 & h_9 \end{pmatrix}$$

which we can reshape into a 3 x 3 homography matrix.

Let us do this in our code –

```
# Reshape last column of V to get homography matrix
# Use the transpose of V since np.linalg.svd returns transpose of results
H = V.T[:,-1].reshape((3,3))
```

5. Divide the entire matrix H by the last diagonal element of H.

After the previous step, our homography matrix, **H**, is in the form –

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

The final step of our homography estimation process is to divide the entire matrix by h_{33}.

```
# Divide matrix H with the last diagonal element
H = H / H[2,2]
```

Now, our homography estimation process is complete and **H**, is in the form –

$$\begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix}$$

Ans 2.

Below is the resulting Homography Matrix for our selected images :

3.484001	0.018529	-250.282
0.578407	1.572033	-17.1378
0.004358	-8.41e-05	1

*Table 5. Homography Matrix, **H***

Ans 3.

We can even use the homography matrix to transform our perspective distorted image, using the following code snippet –

```
# Perspective transform the distorted image using H
im_warped = cv2.warpPerspective(img_left, H, img_left.shape[:2])
```

Ans 3(a).

The aim of this task was to rectify the large perspective distortion in the left image, such that it matched the target (right) image. Let us now visualize this transformation.

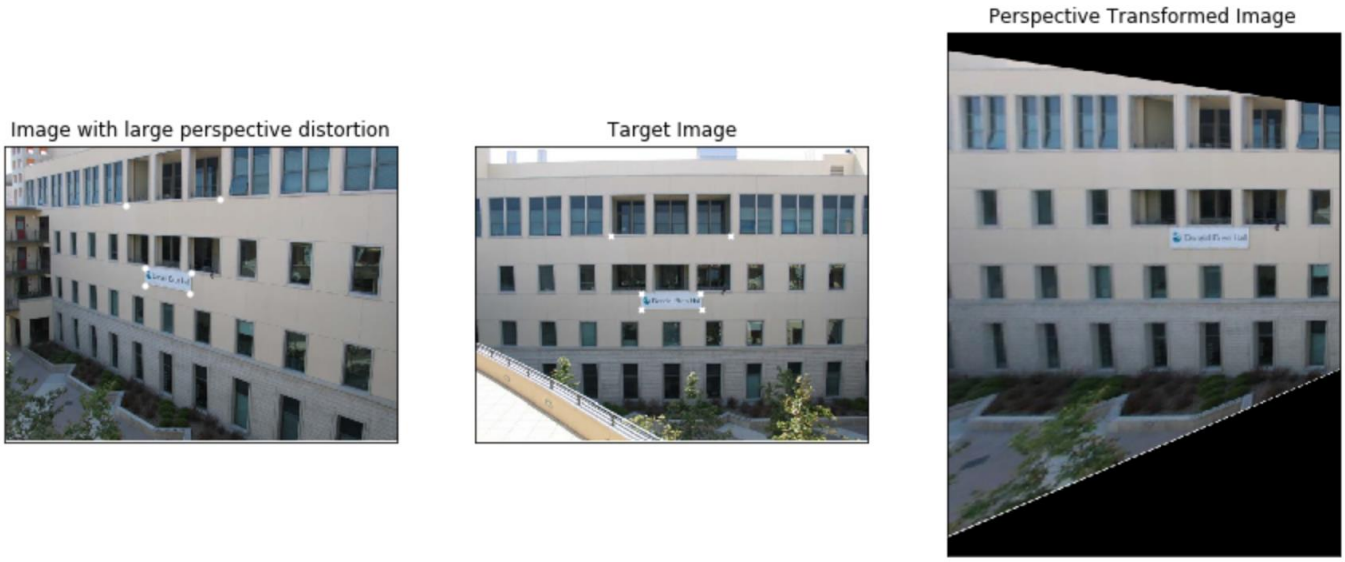

*Figure 7. Perspective transformation of the distorted image using the homography matrix, **H***

The rightmost image is the result of applying perspective transformation to the distorted image. We can clearly observe the similarities between this transformed image and the target image, immediately.

Ans 3(b).

We can further inspect factors such as *the distance between the selected points*, and the effects they have on the transformation results and the mean squared projection error.

For this purpose, we selected 4 distinct sets of coordinates with varying intra-coordinate distances. The figure below (Figure 8) shows the selected points along with the perspective transformed images using the homography matrix computed using these points.

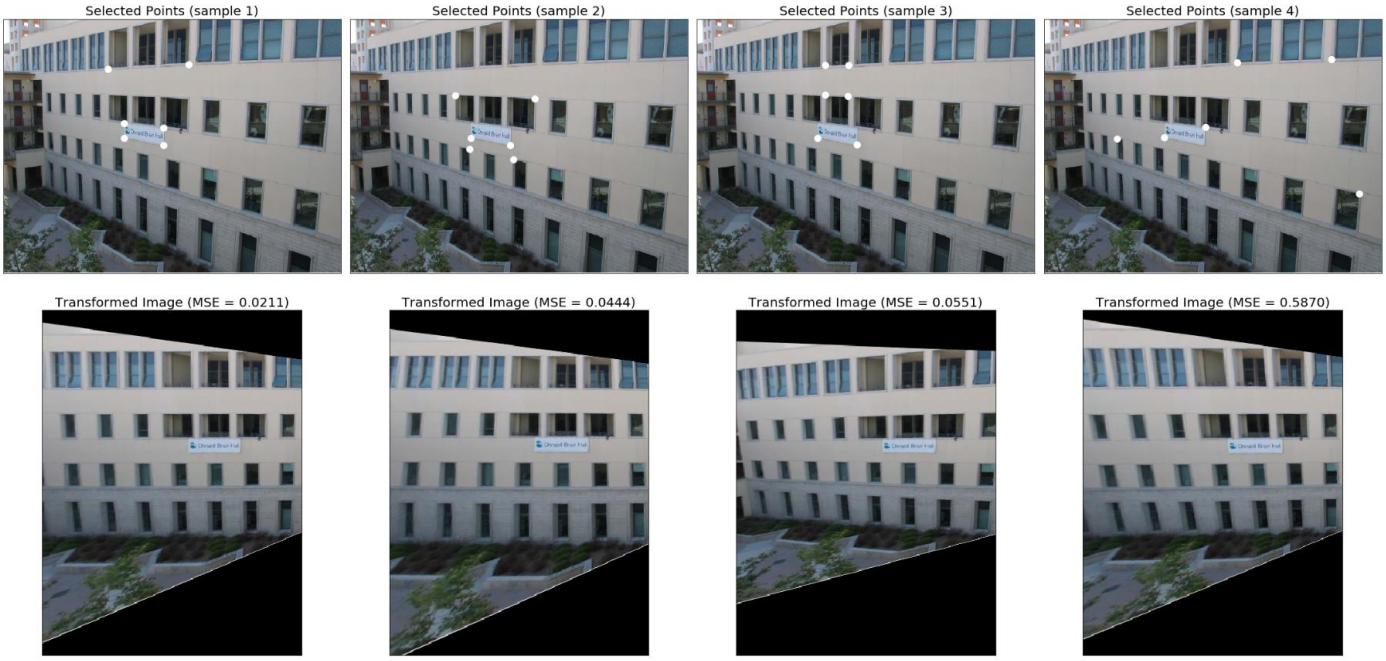

Figure 8. Selected points on the distorted image (top row) and the Transformed Image (bottom row) using the corresponding Homography Matrices (along with the mean squared projection error)

On careful analysis we can see that the best perspective transformation results are obtained by the first 2 sets of coordinates. It is important to note that these sets of coordinates although distinct, have somewhat similar orientations, with the coordinates in sample 2 being closer to each other

The transformation obtained by the second set (sample 2 coordinates) seems to provide a flatter view of the building, as opposed to the transformation obtained by sample 1 coordinates, which is slightly more skewed in comparison. The projection error, however, is greater for sample 2 coordinates. But we note that this difference is marginal.

The transformed image in the case of sample 4 (rightmost) coordinates is not similar to the target (right) image. Also, the highest projection error is observed in this case.

In conclusion, we may deduce that there might exist a correlation between the proximity of selected points and the effectiveness of the Homography matrix computed using these points, with closer points resulting in better homography results.

My Homography Estimation Function (in Python)

```
def homography(u2Trans, v2Trans, uBase, vBase):

    # Reshape arrays to support concatenation
    u2Trans, v2Trans = u2Trans.reshape((-1,1)), v2Trans.reshape((-1,1))

    # Concatenate the distorted image coordinates : [u', v'] ---> [u', v', 1]
    size = len(u2Trans)
    trans_uv = np.concatenate((u2Trans, v2Trans, np.ones((size, 1))), axis=1)

    # Initialise empty list
    A = []

    # Repeat for all coordinates pairs in perpspective distorted image,
    # each element is of the form [u', v', 1]
    for i, ele in enumerate(trans_uv):

        # Coordinates of the target image
        u = uBase[i]
        v = vBase[i]

        # Equations over-constrained equation matrix, A
        eq1 = np.concatenate(([0,0,0], -ele, ele * v))
        eq2 = np.concatenate(( -ele, [0,0,0], ele * u))

        # Add equations to the list
        A.append(eq1)
        A.append(eq2)

    # Convert list to 2N x 9 matrix of of equations
    A = np.array(A)

    # Decompose matrix A using SVD
    _, _, V = np.linalg.svd(A)

    # Reshape last column of V to get homography matrix
    H = V.T[:,-1].reshape((3,3))  # Use the transpose of V since np.linalg.svd returns transpose of
results

    # Divide matrix H with the last diagonal element
    H = H / H[2,2]

    return H
```

References

1. Au.mathworks.com. 2020. *What Is Camera Calibration?- MATLAB & Simulink- Mathworks Australia*. [online] Available at: <https://au.mathworks.com/help/vision/ug/camera-calibration.html#:~:text=Geometric%20camera%20calibration%2C%20also%20referred,the%20camera%20in%20the%20scene.> [Accessed 5 June 2020].
2. Planning.cs.uiuc.edu. 2020. *Determining Yaw, Pitch, And Roll From A Rotation Matrix*. [online] Available at: <http://planning.cs.uiuc.edu/node103.html> [Accessed 6 June 2020].

APPENDIX

1. Inputs coordinates for camera Calibration

	U	V	X	Y	Z
1.	377.336	92.3107	14	35	0
2.	267.603	210.916	0	21	7
3.	162.54	134.336	0	35	21
4.	357.724	455.13	21	0	21
5.	421.696	392.559	21	0	7
6.	175.147	195.973	0	28	21

Table 3. Image coordinates (U,V) and corresponding Real-World coordinates (X, Y, Z) for Camera Calibration

2. Code for obtaining intrinsic and extrinsic parameters from C

```
%VGG_KR_FROM_P Extract K, R from camera matrix.
%
%    [K,R,t] = VGG_KR_FROM_P(P [,noscale]) finds K, R, t such that P = K*R*[eye(3) -t].
%    It is det(R)==1.
%    K is scaled so that K(3,3)==1 and K(1,1)>0. Optional parameter noscale prevents this.
%
%    Works also generally for any P of size N-by-(N+1).
%    Works also for P of size N-by-N, then t is not computed.

% original Author: Andrew Fitzgibbon <awf@robots.ox.ac.uk> and awf
% Date: 15 May 98

% Modified by Shu.
% Date: 8 May 20
'''
import numpy as np

def vgg_rq(S):
    S = S.T
    [Q,U] = np.linalg.qr(S[::-1,::-1], mode='complete')

    Q = Q.T
    Q = Q[::-1, ::-1]
    U = U.T
    U = U[::-1, ::-1]
    if np.linalg.det(Q)<0:
        U[:,0] = -U[:,0]
        Q[0,:] = -Q[0,:]
    return U,Q


def vgg_KR_from_P(P, noscale = True):
    N = P.shape[0]
    H = P[:,0:N]
    print(N,'|', H)
    [K,R] = vgg_rq(H)
    if noscale:
        K = K / K[N-1,N-1]
        if K[0,0] < 0:
            D = np.diag([-1, -1, np.ones([1,N-2])]);
            K = K @ D
            R = D @ R

            test = K*R;
            assert (test/test[0,0] - H/H[0,0]).all() <= 1e-07

    t = np.linalg.inv(-P[:,0:N]) @ P[:,-1]
    return K, R, t

K, R, t = vgg_KR_from_P(C)
```