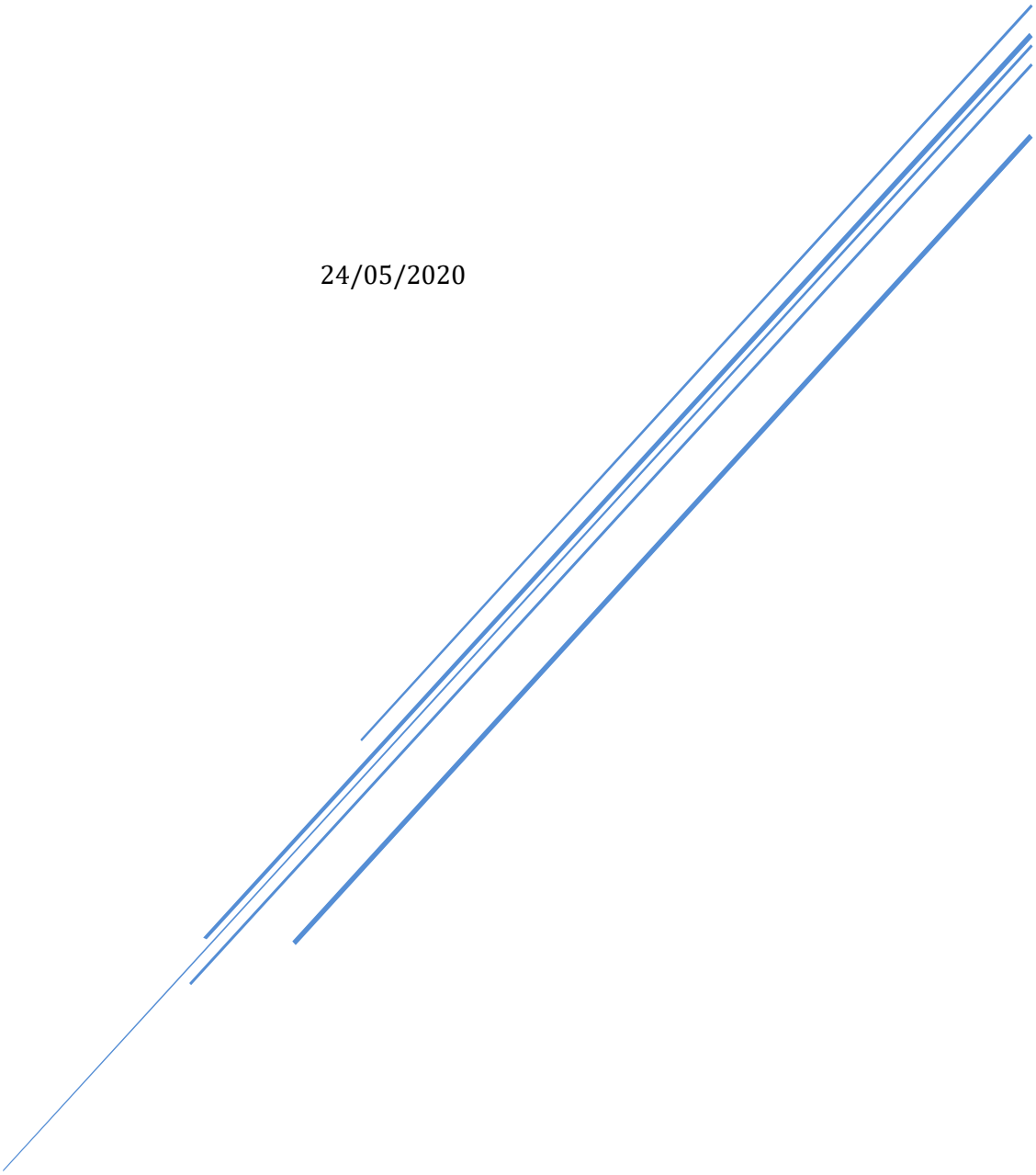
COMPUTER VISION CLAB3 REPORT

ENGN6528

24/05/2020



u7043565
Tanya Dixit

CONTENTS

1. Calibrate Function

Documentation

Following is the source code for the **calibrate** function. This function takes in X,Y,Z coordinates and corresponding image plane coordinates (got them from ginput), and we find out the calibration matrix using solution to least squares problem ||Aq – b||^2(after finding A and b).

```
def calibrate(im, XYZ, uv):
    X = XYZ[:, 0] #get X, Y, and Z
    Y = XYZ[:, 1]
    Z = XYZ[:, 2]

    u = [x[0] for x in uv] #Get u and v separately from tuples
    v = [x[1] for x in uv]

    num_points = XYZ.shape[0] #get the number of points marked

    #declare matrices A and b
    A = np.zeros((num_points*2, 11))
    b = np.zeros((num_points*2, 1))

    j=0
    for i in range(0, num_points*2, 2):
        #DLT algorithm from lectures
        A[i] = [X[j], Y[j], Z[j], 1, 0, 0, 0, 0, -u[j]*X[j], -u[j]*Y[j], -u[j]*Z[j]]
        A[i+1] = [0, 0, 0, 0, X[j], Y[j], Z[j], 1, -v[j]*X[j], -v[j]*Y[j], -v[j]*Z[j]]
        b[i] = u[j]
        b[i+1] = v[j]
        j += 1

    #The matrix is the solution to a linear least squares problem
    C = np.linalg.lstsq(A, b, rcond=None)[0]

    #these two should be equal, verify by printing
    # print(A@C)
    # print(uv)

    newrow = [1]
    C = np.vstack([C, newrow]) #append 1 (last entry) so that it can be reshaped to 3x4
    C = C.reshape((3,4))
    return C
```

The rest is display code and reconstruction code and can be referred from **calibrate.py**
Also, the points xyz can be seen in **calibrate.py** as well.

Observations and Results

2. **Image used for experiments**

In Fig. 1.1, the image used for calibration is displayed “stereo2012a.jpg”

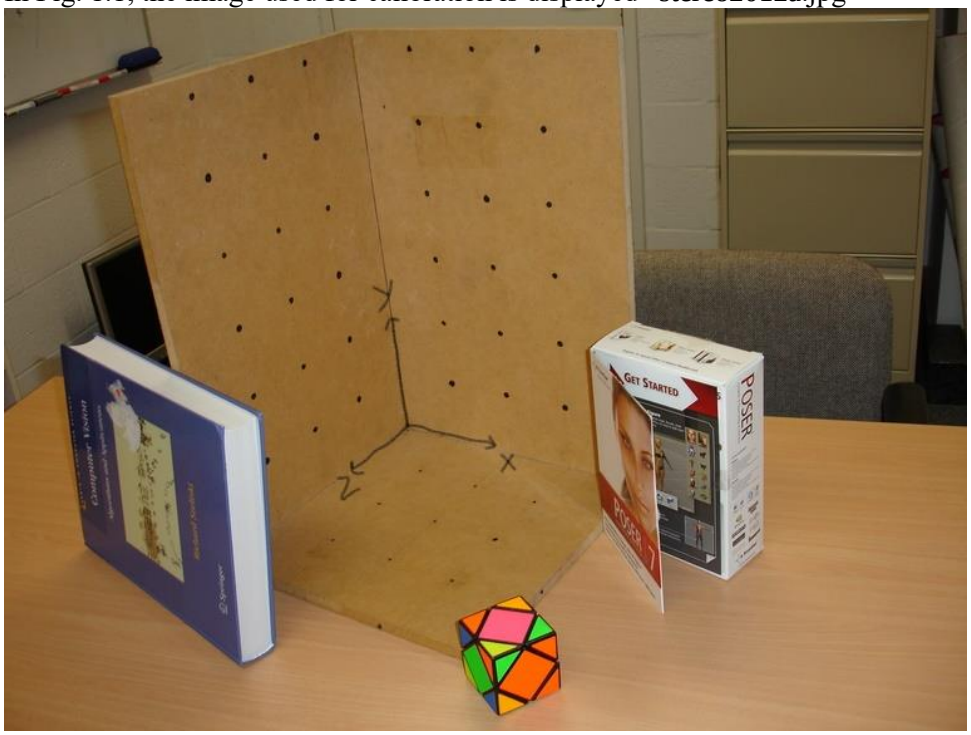


FIGURE 1.1 IMAGE STEREO2012A.JPG USED FOR CALIBRATION

In Fig. 1.2, we display the points in world coordinates used for calibrating. These are marked while taking ginput.

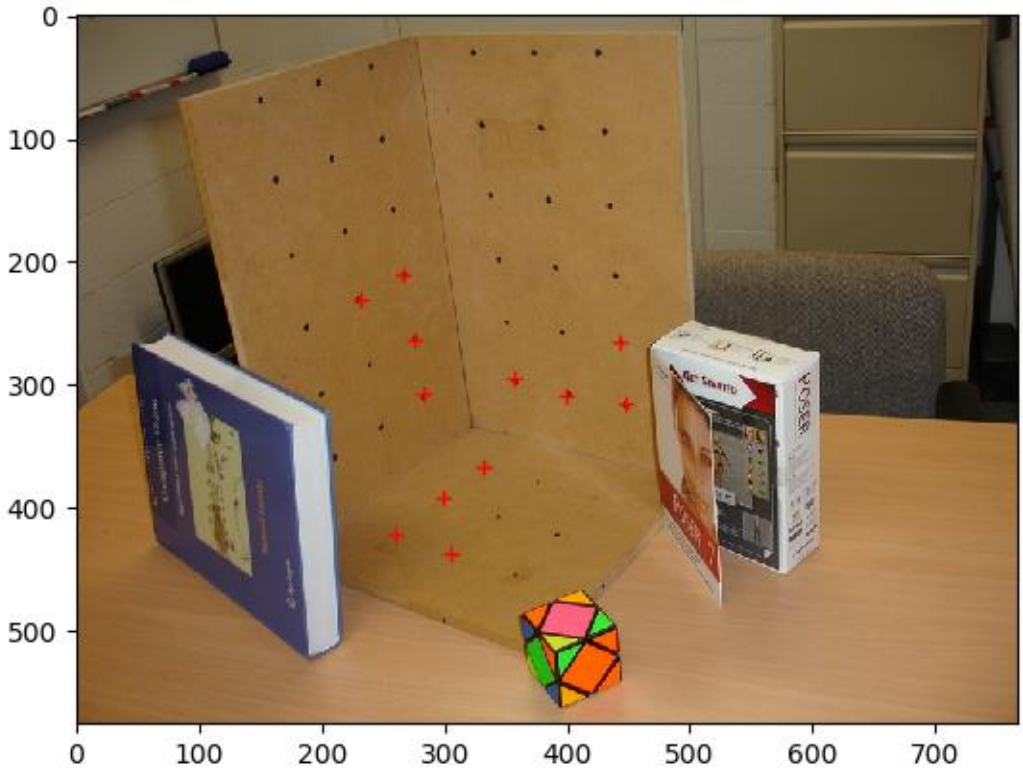


FIGURE 1.2 THE XYZ COORDINATES MARKED IN RED CROSSES – 12 POINTS USED FOR CALIBRATING

In Fig. 1.3, we display the original XYZ points in red and the reconstructed points in blue. Although the red points aren't completely visible, comparing the reconstructed points in Fig. 1.3 with the red crosses in Fig. 1.2 (actual XYZ coordinates), seems like the calibration was done okay.

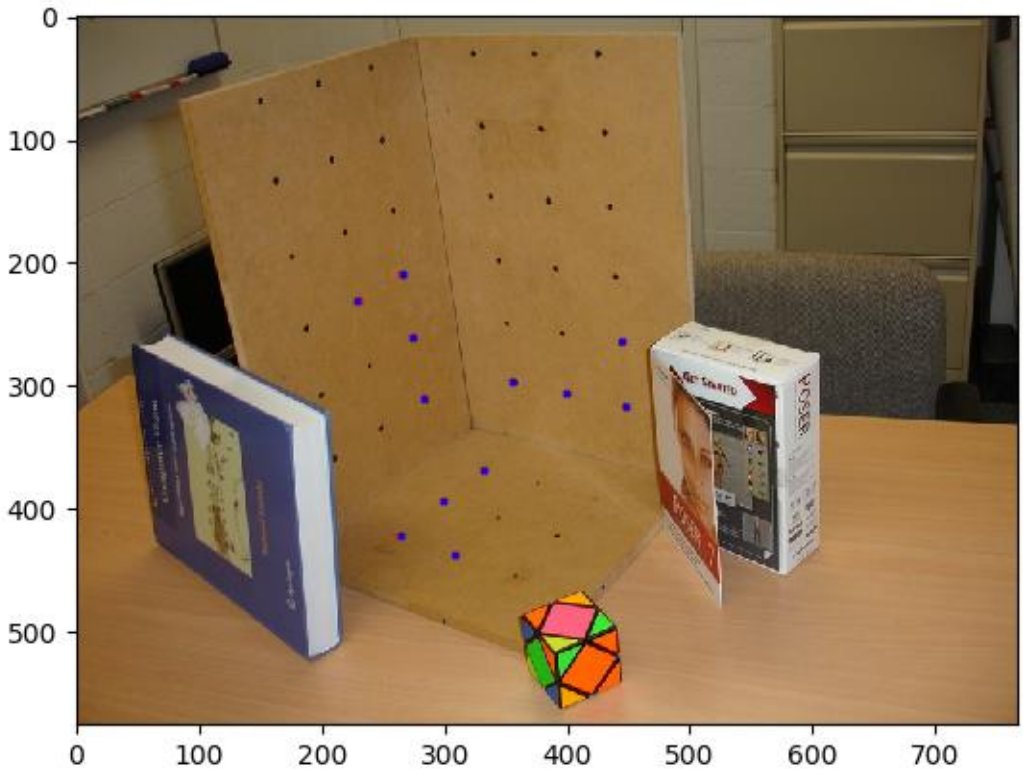


FIGURE 1.3 RECONSTRUCTED POINTS IN BLUE JUXTAPOSED WITH XYZ COORDINATES

3. Get Calibration Matrix

Displayed below is a 3x4 calibration matrix obtained from the function **calibrate** in **calibrate.py**

3.880364	-2.140653	-5.618045	321.172915
-0.299588	-7.571841	1.736675	333.202531
-0.005188	-0.003672	-0.004355	1.000000

Error Calculated: The error between actual u,v and reconstructed u,v from calibration matrix is: 0.267446

4. Get K, R, t matrix

K Matrix

```
[[905.59582986   14.55273388  205.47119936]
 [  0.          939.87965274  367.09114911]
 [  0.            0.            1.        ]]
```

R Matrix

[[0.70534851 -0.18486569 -0.68433045]
[0.22161255 -0.85949518 0.46060386]
[-0.67332857 -0.47654246 -0.56527508]]

t matrix

[76.67957913 60.90808597 86.92249222]

5. **Focal Length and Pitch Angle**

The focal length (scaled) of the camera seems is 922.73 (taking average assuming m_x and m_y are equal)

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

Pitch angle: For a rotation matrix R given as: , the pitch angle is given as

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

Therefore, pitch angle = 42.32 degrees or 0.738 radians

Analysis

The calibration matrix depends a lot on the choice of points used to calibrate. It is important to take points that cover all planes. Also, we can't really know the focal length in cm, only in pixels from this. If we want to know it in cms, we will have to break open the camera and look inside it to know the pixels per cm (m_x, m_y).

We see that we are able to calculate the calibration matrix using the DLT algorithm and reconstruct the points using the calibration matrix. Essentially what we did was solve a linear equation between points in the world and points in the image plane. And using that, we are able to find out the characteristics of the camera we use.

We observe that in order to get an accurate calibration, we need to make sure the points marked using ginput are accurate. If it's even slightly inaccurate, the C matrix changes drastically and can lead to even 100-200 pixel variation in focal length, and 5-10 degrees variation in pitch angle.

TASK 2 – TWO-VIEW DLT BASED HOMOGRAPHY ESTIMATION

2.1 Code snippets with comments

Documentation

The snippet below in function **homography** shows code for calculating the Homography matrix. The function **test_homography** tests if the reconstructed points using the H matrix correspond to the actual points on the image. Please find the rest of the display code in **calibrate.py**

We solve a linear equation (using SVD) to get the mapping between two images.

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

    num_points = len(u2Trans)
    A = np.zeros((num_points*2, 9))

    j=0
    for i in range(0, num_points*2, 2):
        print(i)
        A[i] = [u2Trans[j], v2Trans[j], 1, 0, 0, 0, -u2Trans[j]*uBase[j], -uBase[j]*v2Trans[j], -uBase[j]]
        A[i+1] = [0, 0, 0, u2Trans[j], v2Trans[j], 1, -u2Trans[j]*vBase[j], -v2Trans[j]*vBase[j], -vBase[j]]
        j += 1

    u, s, vh = np.linalg.svd(A, full_matrices=True)

    H = vh[-1, :]/vh[-1,-1]
    return H

H_matrix = homography(u_circ, v_circ, u_base, v_base)

def test_homography(H, base, circ):

    newrow = np.repeat([1],circ.shape[1])
    circ = np.vstack([circ, newrow])

    #H = H/H[-1]
    print(H)
    x = H.reshape((3,3))@circ

    r_u = x[0, :]/x[2, :]
    r_v = x[1, :]/x[2, :]
    reconstructed_base = np.asarray([r_u, r_v])
    print(reconstructed_base)
    print(base)

circ = np.asarray([u_circ, v_circ])
print(circ.shape)
base = np.asarray([u_base, v_base])
test_homography(H_matrix, base, circ)
```

Warping function:

The code snippet given below is the warping function used to warp left image using the H matrix.

```python
#Function to warp left image
def warp_img(img, H):
    #since it's a square image
    dst = np.zeros(img.shape)
    h, w = img.shape
    # print(h)
    # print(w)

    for x in range(h):
        for y in range(w):
            newrow = np.repeat([1],1)
            init_coords = np.vstack([x, y, newrow])
            u, v, s = np.linalg.inv(H.reshape(3,3))@init_coords
            u = int(u/s)
            v = int(v/s) #no interpolation technique applied here, just converted to int

            if (u >= 0 and u < h) and (v >=0 and v < w):
                dst[u, v] = img[x,y]
    return dst
```

FIGURE 2.1 LEFT IMAGE AND POINTS (RED) USED FOR CALCULATING THE H MATRIX MENTIONED IN PART 2

FIGURE 2.2 RIGHT IMAGE AND POINTS(RED) USED FOR CALCULATING THE H MATRIX MENTIONED IN PART 2

2. **The Homography Matrix:**

[3.3279 -0.1859 -221.204]

[0.5925 1.2273 7.269]

[0.0043 -0.0008 1.0]

Observations and Results

3. **Factors that influence the H matrix and consequent warped image**

1. **Base Image**: Below in Fig 2.3 is the image resulting from warping the left image using the H matrix written above. Let's take this as our base image and compare to other warped images using other factors.

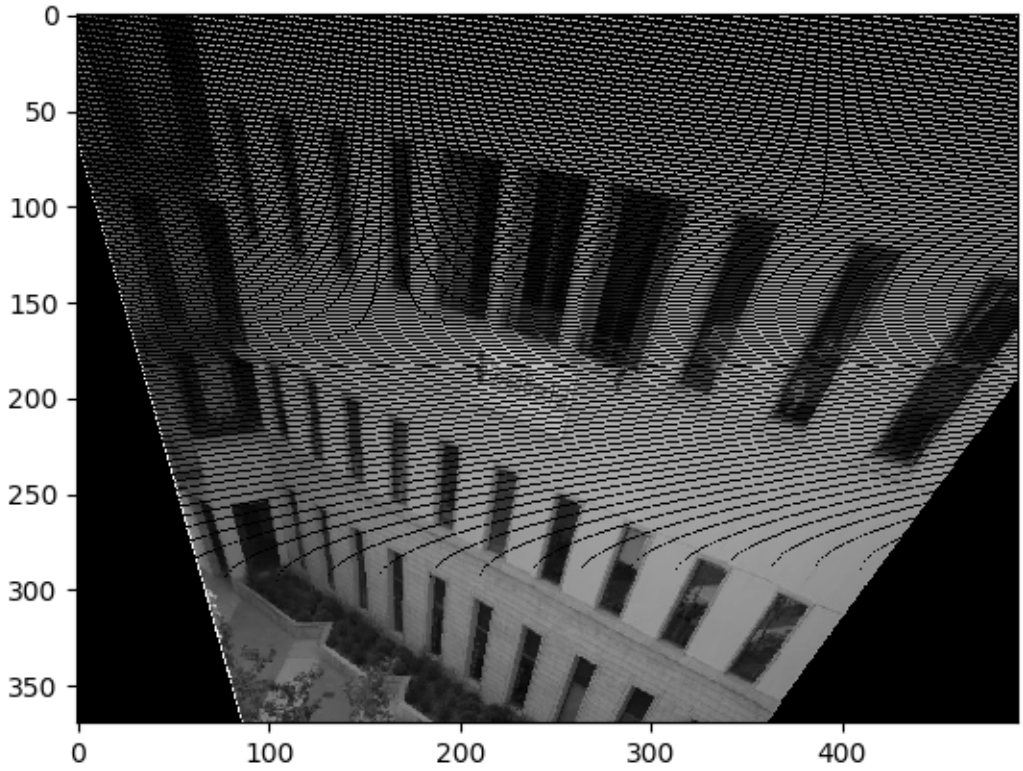


FIGURE 2.3 WARPED IMAGE USING H MATRIX IN PART 2

2. More distance between points

We increased the distance between points and compare it to our base image. In Fig 2.4 we show the right image with the points used marked in Red. Note that the same points were marked on the left image but aren't shown here to keep the report concise.

FIGURE 2.4 INCREASE THE DISTANCE BETWEEN POINTS (RED) TO CALCULATE A NEW H MATRIX

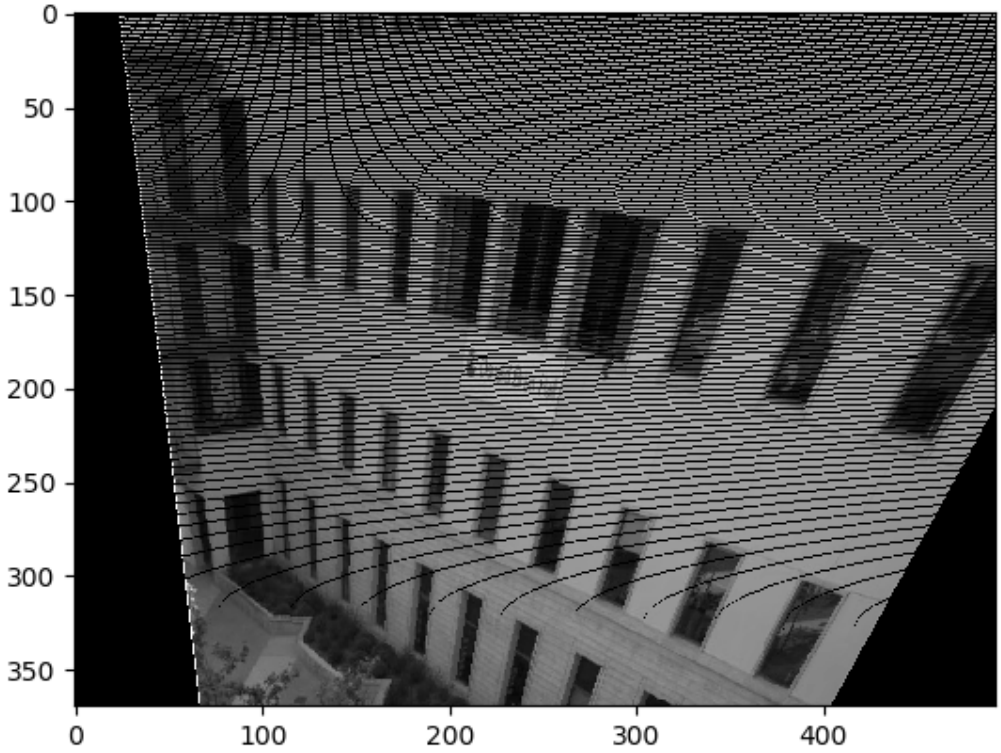


FIGURE 2.5 THE WARPED IMAGE FORMED FROM LEFT IMAGE AND THE NEW H MATRIX WHEN WE INCREASE THE DISTANCE BETWEEN POINTS

3. What if we don't take corner points?

Let's try taking points that are not on corners or edges. In Fig 2.6, we show the right image with the points used marked in Red. Note that the same points were marked on the left image but aren't shown here to keep the report concise.

FIGURE 2.6 TAKE POINTS (RED) THAT ARE NOT EDGES OR CORNERS TO CALCULATE A NEW H MATRIX

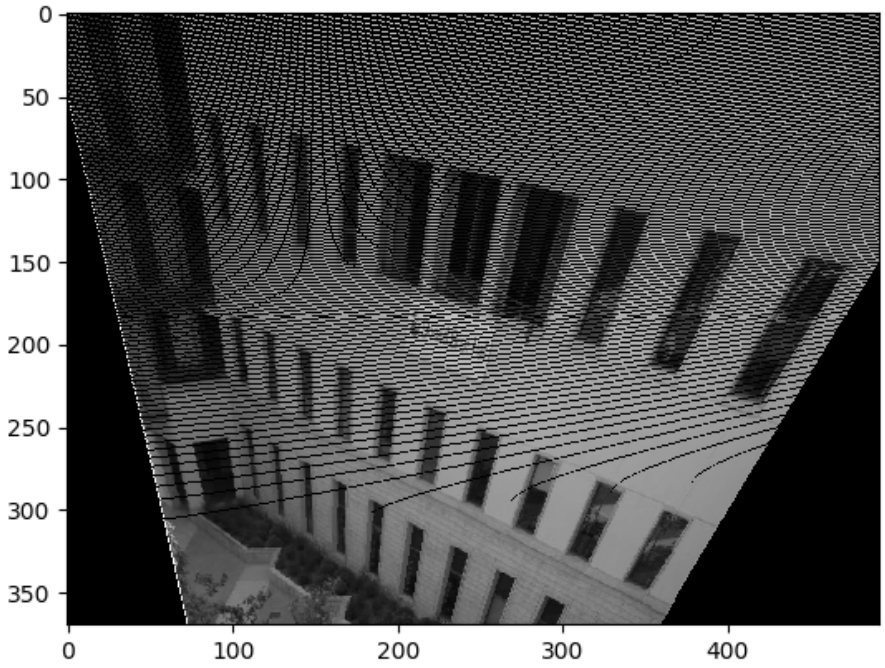


FIGURE 2.7 THE WARPED IMAGE FORMED FROM LEFT IMAGE AND THE NEW H MATRIX WHEN WE USE NO CORNER OR EDGE POINTS

4. Taking points that are too close

Let's try taking points that very close and compact. In Fig 2.8, we show the right image with the points used marked in Red. Note that the same points were marked on the left image but aren't shown here to keep the report concise.

FIGURE 2.8 TAKE POINTS (RED) THAT VERY COMPACT TO CALCULATE A NEW H MATRIX

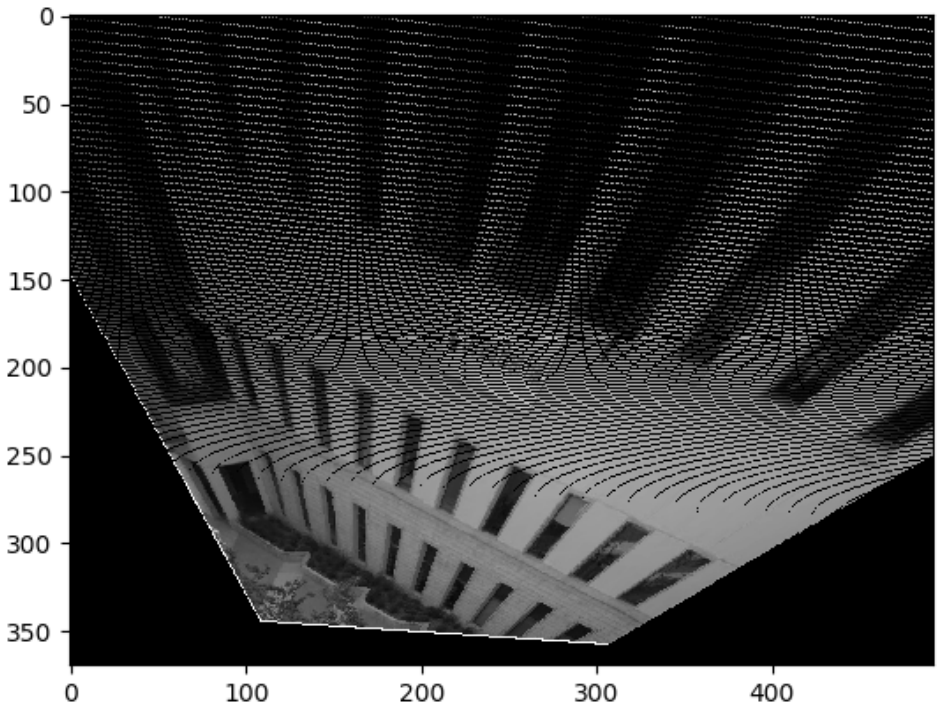


FIGURE 2.9 THE WARPED IMAGE FORMED FROM LEFT IMAGE AND THE NEW H MATRIX WHEN WE USE COMPACT POINTS

5. **Taking all points in one line**
   Let's try taking points that are collinear. In Fig 2.10, we show the right image with the points used marked in Red. Note that the same points were marked on the left image but aren't shown here to keep the report concise.

FIGURE 2.10 TAKE POINTS (RED) THAT ARE COLLINEAR TO CALCULATE A NEW H MATRIX

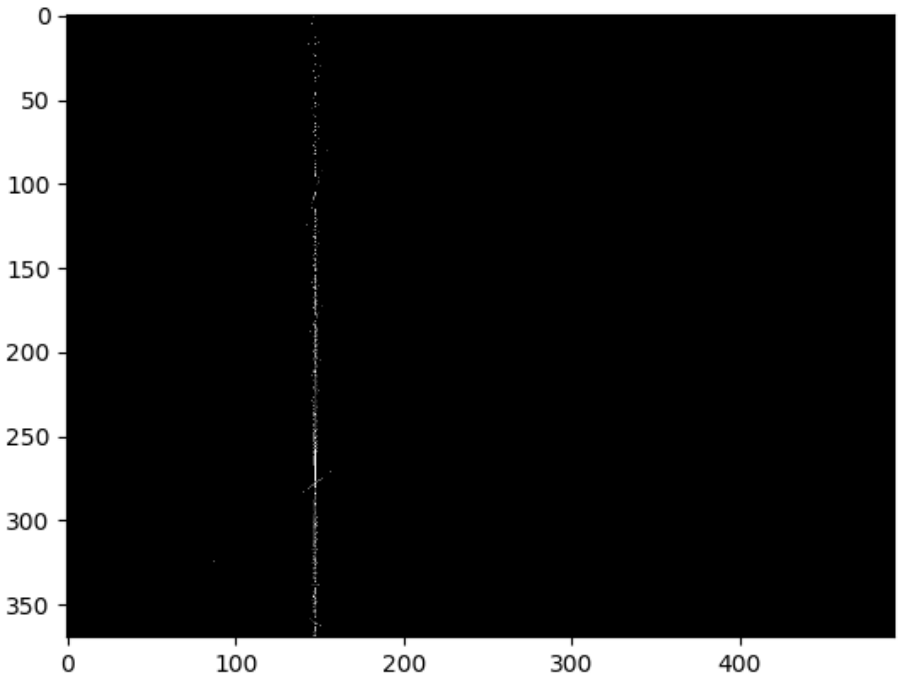


FIGURE 2.11 THE WARPED IMAGE FORMED FROM LEFT IMAGE AND THE NEW H MATRIX WHEN WE USE COLLINEAR POINTS

6. Taking 4 points

Let's try taking less number of points. In Fig 2.8, we show the right image with the points used marked in Red. Note that the same points were marked on the left image but aren't shown here to keep the report concise.

FIGURE 2.12 TAKE 4 POINTS (RED) INSTEAD OF 6 TO CALCULATE A NEW H MATRIX

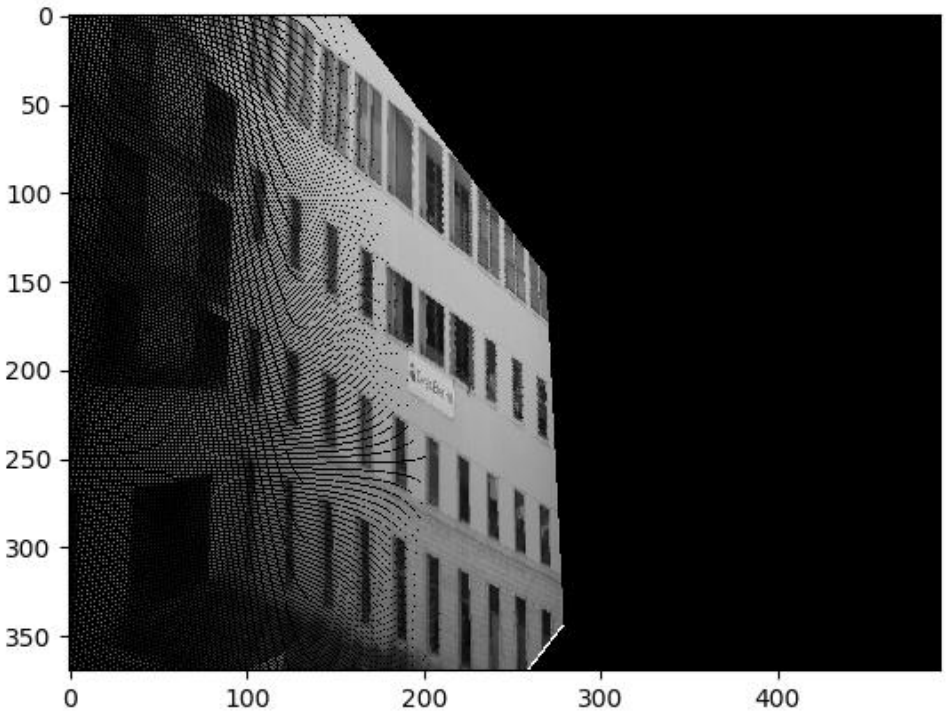


FIGURE 2.13 THE WARPED IMAGE FORMED FROM LEFT IMAGE AND THE NEW H MATRIX WHEN WE USE 4 POINTS

7. Taking 12 points

Let's try taking more number of points than before. In Fig 2.8, we show the right image with the points used marked in Red. Note that the same points were marked on the left image but aren't shown here to keep the report concise

FIGURE 2.14 TAKE 12 POINTS (RED) INSTEAD OF 6 TO CALCULATE A NEW H MATRIX

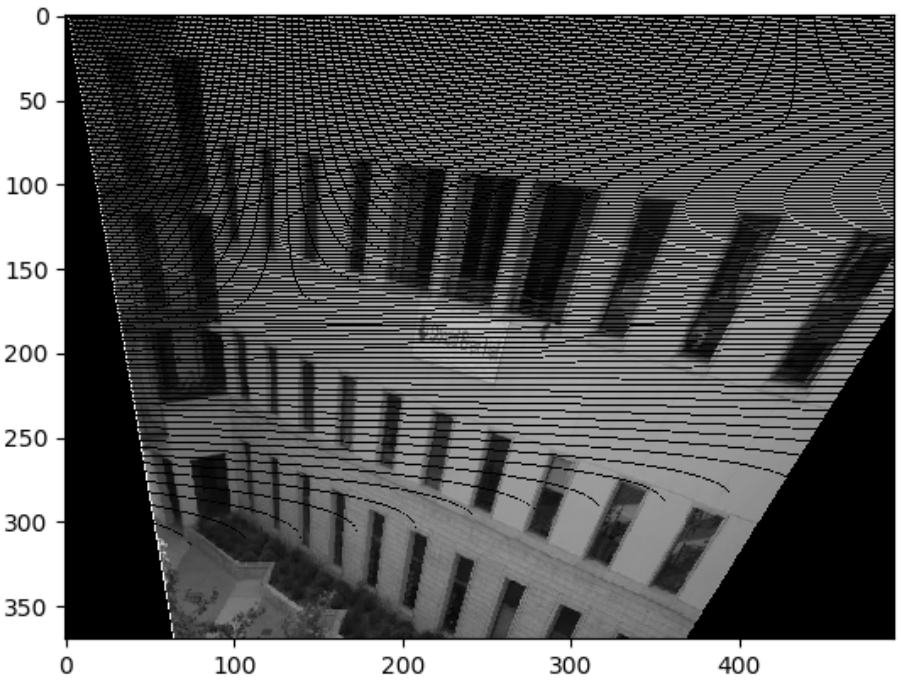


FIGURE 2.15 THE WARPED IMAGE FORMED FROM LEFT IMAGE AND THE NEW H MATRIX WHEN WE USE 12 POINTS

Analysis

From the above experiments, we can see that the number of points is a very big factor. Taking 4 points leads to a very different warping than taking 6 points or more. If all points are collinear, H matrix doesn't give any information and we get a black warped image. Basically, it maps all points to a line instead of mapping it to the 2-d image. If we take points that are too close, the warping is accentuated, while far apart points lead to less skew in warping. The reason could be that close points don't give a general overview of the entire image leading to very skewed results.

Also, taking corner points generally results in better results than taking in-between points. Generally in cv2 the warping function first identifies corner points using Harris corner and then uses H matrix to warp those points and get the warped image.

Thank you for taking the time to read this report.