CLab 3 Report
COMP4528

Kirat Alreja
u7119530

23/05/2022

Task 1 Reference : [6] [7] [10]

1. List `calibrate` function in your PDF file

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

    '''
    %% TASK 1: CALIBRATE
    %
    % Function to perform camera calibration
    %
    % Usage:   calibrate(image, XYZ, uv)
    %          return C
    %   Where:   image - is the image of the calibration target.
    %            XYZ - is a N x 3 array of  XYZ coordinates
    %                  of the calibration target points.
    %            uv  - is a N x 2 array of the image coordinates
    %                  of the calibration target points.
    %            K   - is the 3 x 4 camera calibration matrix.
    %  The variable N should be an integer greater than or equal to 6.
    %
    %  This function plots the uv coordinates onto the image of the calibration
    %  target.
    %
    %  It also projects the XYZ coordinates back into image coordinates using
    %  the calibration matrix and plots these points too as
    %  a visual check on the accuracy of the calibration process.
    %
    %  Lines from the origin to the vanishing points in the X, Y and Z
    %  directions are overlaid on the image.
    %
    %  The mean squared error between the positions of the uv coordinates
    %  and the projected XYZ coordinates is also reported.
    %
    %  The function should also report the error in satisfying the
    %  camera calibration matrix constraints.
    %
    % Kirat Alreja, 23/05/2022
    '''
```

```
    img_dim = im.shape[2]
    world = np.asarray(XYZ)
    dim_world = world.shape[1]
    image = np.asarray(uv)
    dim_image = image.shape[1]

    #normalise both world & image points
    world_norm_matrix, world_normalised =      normalise_data(world,np.mean(world,0),np.std(world),dim_world)
    image_norm_matrix, image_normalised = normalise_data(image,np.mean(image,0),np.std(image),dim_image)

    #calculate the A matrix equation for calibration
    A = []
    for i in range(world.shape[0]):
        x, y, z = world_normalised[i, 0], world_normalised[i, 1],   world_normalised[i, 2]
        u, v = image_normalised[i, 0], image_normalised[i, 1]
        A.append( [x, y, z, 1, 0, 0, 0, 0, -u * x, -u * y, -u * z, -u] )
        A.append( [0, 0, 0, 0, x, y, z, 1, -v * x, -v * y, -v * z, -v] )
    A = np.asarray(A)

    #perform SVD on A
    U, S, V = np.linalg.svd(A)

    #solution is in the last column of V, and normalise it
    C = V[-1, :] / V[-1, -1]

    #reshape it to 3x4 matrix
    C = C.reshape((3,img_dim+1))

    return C
```

Along with the `calibrate` function, I used two helper functions to complete the task. The first one is `normalise_data` which is used to normalise the world & image points appropriately for the calibration operaion. The second one is `get_points_and_error` which does the same operations as calibrate, but to calculate the projected points using calibration matrix & DLT error.

I used the following matrix for normalisation operation

T = [[standard_deviation,0,0,mean[0]]
[[0,standard_deviation,0,mean[1]]
[[0,0,standard_deviation,mean[2]]
[[0, 0, 0, 1]]

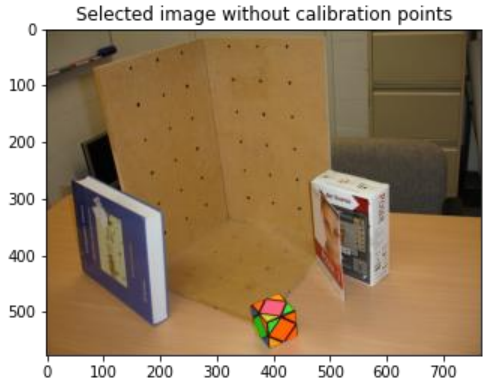
T = inverse(T)

```
def normalise_data(x,mean,std,dimensions):

    #normalising the data using a transformation

    #define the normalisation matrix for two or three dimensions, as needed
    if dimensions == 2:
        Norm_Matrix = [[std,0,mean[0]],[0,std,mean[1]],[0,0,1]]
    else:
        Norm_Matrix = [[std,0,0,mean[0]],[0,std,0,mean[1]],[0,0,std,mean[2]],[0,0,0,1]]



    Norm_Matrix = np.asarray(Norm_Matrix)
    Norm_Matrix_Inverse = np.linalg.inv(Norm_Matrix)

    #add a row of ones to the data for dot operation
    form_for_normalisation = np.concatenate((x.T,np.ones((1,x.shape[0]))))

    #multiply the data with normalisation matrix
    normalised_x = np.dot(Norm_Matrix_Inverse,form_for_normalisation)

    #delete the extra row added for dot operation
    normalised_x = normalised_x[0:dimensions,:].T

    return Norm_Matrix_Inverse,normalised_x

def get_points_and_error(XYZ,uv):

    #perform similar operations to "calibrate", but for calculating
    #the projected points & DLT error
    world = np.asarray(XYZ)
    dim_world = world.shape[1]
    image = np.asarray(uv)
    dim_image = image.shape[1]

    world_norm_matrix, world_normalised = normalise_data(world,np.mean(world,0),np.std(world),dim_world)
    image_norm_matrix, image_normalised = normalise_data(image,np.mean(image,0),np.std(image),dim_image)
```

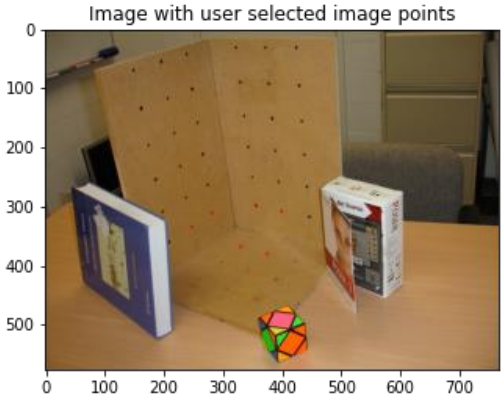
```
A = []
for i in range(world.shape[0]):
    x, y, z = world_normalised[i, 0], world_normalised[i, 1], world_normalised[i, 2]
    u, v = image_normalised[i, 0], image_normalised[i, 1]
    A.append( [x, y, z, 1, 0, 0, 0, 0, -u * x, -u * y, -u * z, -u] )
    A.append( [0, 0, 0, 0, x, y, z, 1, -v * x, -v * y, -v * z, -v] )
A = np.asarray(A)

U, S, V = np.linalg.svd(A)

C = V[-1, :] / V[-1, -1]



#H matrix is used for calculating the value of projected points
H = C.reshape(3, 4)

#Peform denormaliation
H = np.dot(np.dot( np.linalg.pinv(image_norm_matrix), H ), world_norm_matrix)

#Normalise with last column & row
H = H / H[-1, -1]

#Calculate the projected points, with dot product of H & world coordinates appended with 1
projected_points = np.dot( H, np.concatenate( (world.T, np.ones((1, world.shape[0]))) ) )

#Normalise the projected points
projected_points = projected_points / projected_points[2, :]

#mean distance between true & projected points
error = np.sqrt( np.mean(np.sum( (projected_points[0:2, :].T - uv)**2, 1)) )

#delete the appened ones column
projected_points = projected_points[0:2]

return error,projected_points
```

2. List the image you have chosen for your experiment, and display the image in your PDF file

I have chosen the image ‘stereo2012a.jpg’ for my experiment



Here is the image with my selected points for calibration operation

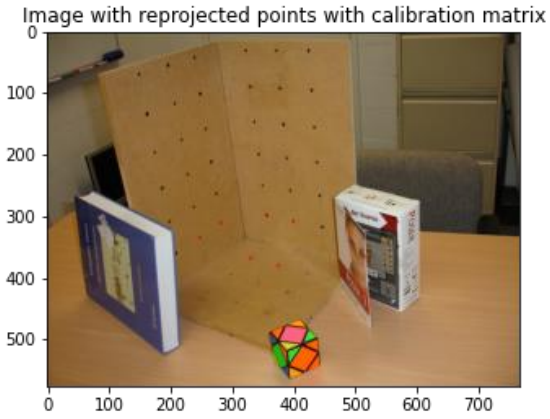


3. List the 3x4 camera calibration matrix P that you have calculated for the selected image. Please visualise the projection of the XYZ coordinates back onto image using the calibration matrix P and report the reprojection error (The mean squared error between the positions of the uv coordinates and the projected XYZ coordinates using the estimated projection matrix)

The 3x4 camera calibration matrix :

```
[[ 0.70762418, -0.05287326, -0.46887587,  0.03752043],
 [ 0.17781128, -0.74564872,  0.40374439, -0.00829428],
 [-0.00708621, -0.01915903, -0.04322884,  1.        ]]
```

Projection of XYZ coordinates back onto the image :



Reprojection Error : 0.605

4. Decompose the P matrix into K, R, t, such that P = K[R|t], by using the following provided code (vgg_KR_from_P.m or vgg_KR_from_P.py). List the results, namely the K, R, t matrices, in your PDF file.

K matrix:

```
[[16.30219637,  0.18086396,  7.11613012],
 [ 0.        , 18.01669172, -1.93677821],
 [ 0.        ,  0.        ,  1.        ]]
```

R matrix:

```
[[ 0.97043795,  0.11716372, -0.21100439],
 [ 0.19048429, -0.90868113,  0.37150281],
 [-0.14820905, -0.40071345, -0.9041365 ]]
```

t matrix:

```
[11.51059845, 11.48220516, 16.15693432]
```

5.

What is the focal length (in the unit of pixel) of the camera?

The focal length of the camera can be extracted from the K matrix, where f(x) is the first diagonal element & f(y) is the second diagonal element. Therefore, here [10] –

$$\mathrm{K} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

f(x) = 16.30
f(y) = 18.01

What is the pitch angle of the camera with respect to the X-Z plane in the world coordinate system?

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

$$\theta_y = atan2\left(-r_{31}, \sqrt{r_{32}^2 + r_{33}^2}\right)$$

The pitch angle of the camera can be extracted from the R [10] matrix with the following formula
pitch = arctan2(-R[3][1], sqrt(R[3][2]^2 + R[3][3]^2)

In this case, we get pitch p = 0.148

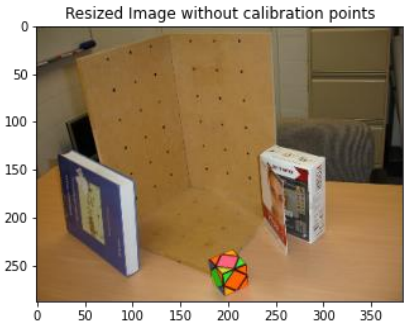
What is the camera centre coordinate in the XYZ coordinate system (world coordinate system)?
The camera centre coordinate can be directly extracted from the value of t matrix

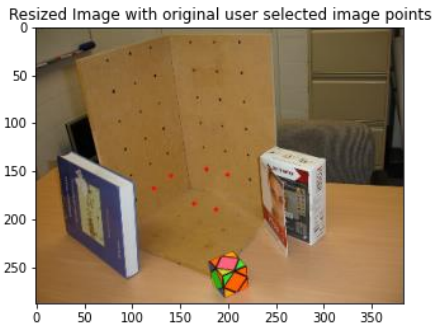
In this case, we get the centre c as (11.51,11.48,16.15)

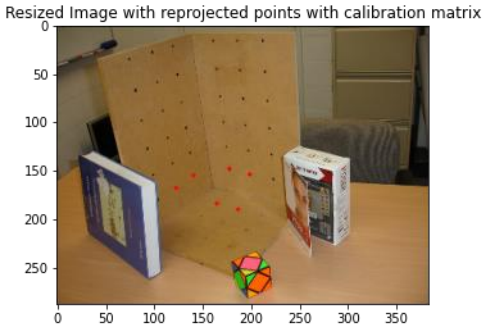
6. Please resize your selected image using builtin function from matlab or python to (H/2, W/2) where H, and W denote the original size of your selected image. Using the interface function, (ginput in Matlab, and matplotlib.pyplot.ginput in Python) to find the uv coordinates in the resized image.

Selected uv points =
[[165.5,184.5],[188,190],[178,148.5],[200,154.5],[141.5,155.5],[123.5,168]]

a) Resized image :







Camera calibration matrix P':

```
[[ 0.70762418, -0.05287326, -0.46887587,  0.03752043],
 [ 0.17781128, -0.74564872,  0.40374439, -0.00829428],
 [-0.00708621, -0.01915903, -0.04322884,  1.        ]]
```

Matrix K':

```
[[16.30219637,  0.18086396,  7.11613012],
 [ 0.        , 18.01669172, -1.93677821],
 [ 0.        ,  0.        ,  1.        ]]
```

Matrix R':

```
[[ 0.97043795,  0.11716372, -0.21100439],
 [ 0.19048429, -0.90868113,  0.37150281],
 [-0.14820905, -0.40071345, -0.9041365 ]]
```

Matrix t':

```
[11.51059845, 11.48220516, 16.15693432]
```

Focal length f'(x) : 16.302, f'(y) : (18.016)

Camera center c' : (11.51,11.48,16.15)

Pitch p' : 0.148

b) Looking at all three matrices of the original image (K,R,t) and comparing them to the ones of the resized image (K',R',t') – we can observe that the matrices are exactly the same. There is no change in the parameters whatsoever. These matrices are extracted from the camera calibration matrix and are known as the intrinsic parameters of a camera. Intrinsic means "internal", which indicates that the external environment should not affect these parameters. As we can observe from the comparison between original & resized images, the camera intrinsics indeed do not depend on the resizing of the image. The image points in the resized image are halved from the original image, which means that the ratio of distances between the points is maintained. Thus, the intrinsic parameters depend only on the ratio of the distance between points and do not vary with scale.

c) Analyzing the focal length(f & f') and principal points (p & p'), we can observe that they remain exactly the same between original and resized images. The reasoning behind the same

is similar to the previous answer, where focal length & principal points are a part of camera's intrinsic parameters and they don't depend on image size.

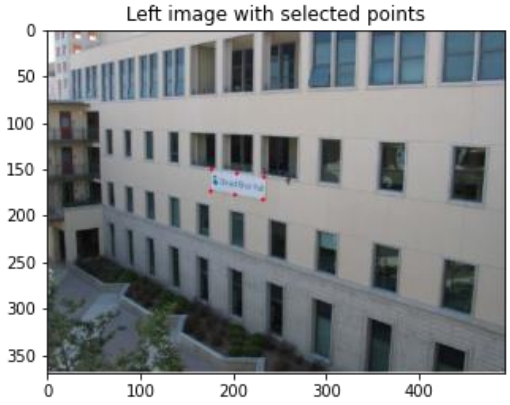
Task 2 Reference [1] [2] [3] [9] [10]s

1. List your source code for homography estimation function and display the two images and the location of six pairs of selected points (namely, plotted those points on images). Explain what you have done for homography and what is shown.

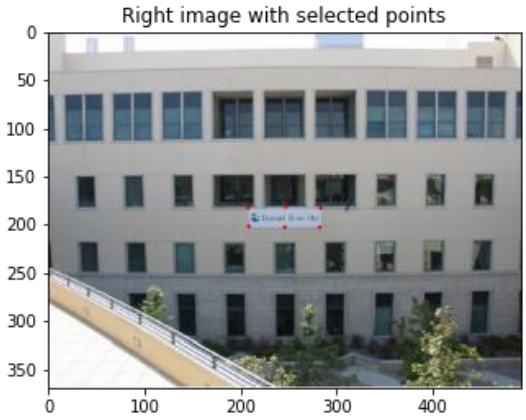
```
def homography(u2Trans,v2Trans,uBase,vBase):

    '''
    %% TASK 2:
    % Computes the homography H applying the Direct Linear Transformation
    % The transformation is such that
    % p = np.matmul(H, p.T), i.e.,
    % (uBase, vBase, 1).T = np.matmul(H, (u2Trans , v2Trans, 1).T)
    % Note: we assume (a, b, c) => np.concatenate((a, b, c), axis), be careful when
    % deal the value of axis
    %
    % INPUTS:
    % u2Trans, v2Trans - vectors with coordinates u and v of the transformed image point (p')
    % uBase, vBase - vectors with coordinates u and v of the original base image point p
    %
    % OUTPUT
    % H - a 3x3 Homography matrix
    %
    % Kirat Alreja, 23rd May 2022
    '''
    #concatenate x & y points into single image
    right_uv = np.concatenate((u2Trans,v2Trans),axis=1)
    left_uv = np.concatenate((uBase,vBase),axis=1)
    #use homography equations to form the matrix for SVD operation
    A_0 = np.column_stack((left_uv,np.ones(left_uv.shape[0]),np.zeros((left_uv.shape[0],3)),-
left_uv[:,0]*right_uv[:,0],-left_uv[:,1]*right_uv[:,0],-right_uv[:,0]))
    A_1 = np.column_stack((np.zeros((left_uv.shape[0],3)),left_uv,np.ones(left_uv.shape[0]),-
left_uv[:,0]*right_uv[:,1],-left_uv[:,1]*right_uv[:,1],-right_uv[:,1]))
    A = np.vstack((A_0,A_1))

    #calculating the SVD operation, and extracting solution from the last column of V matrix
    H = np.linalg.svd(A)[-1][-1]
    #normalise the matrix with last column
    H = H/H[-1]
    #reshape result to 3x3
```

```
    H = H.reshape(3,3)

    return H
```

Left image with selected points



Right image with selected points



Explanation

Homography can be defined as the process of defining a transformation between two planes, where one plane can be warped to the other with the calculated homography matrix. Here, we have two images of the same object taken at different angles. Although the angle of the pictures is different, they can be mapped to the same plane. The homography equation can be defined as

$$\begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & H_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix}$$

The homography matrix has a degree of freedom with a value of 8, which means there are 8 unknowns that need to be determined. Therefore, the minimum number of points required to compute the matrix is four. If we add more points, we would get better results. Here, according to the specification, I have used six points for the computation.

Rearranging the above equation results in the following result

$$\begin{aligned} \mathbf{a}_x &= (-x_1, -y_1, -1, 0, 0, 0, x_2' x_1, x_2' y_1, x_2')^T \\ \mathbf{a}_y &= (0, 0, 0, -x_1, -y_1, -1, y_2' x_1, y_2' y_1, y_2')^T . \end{aligned}$$

Where the two a matrices can be concatenated to a matrix A and then used to solve for H

$$A\mathbf{h} = \mathbf{0}$$

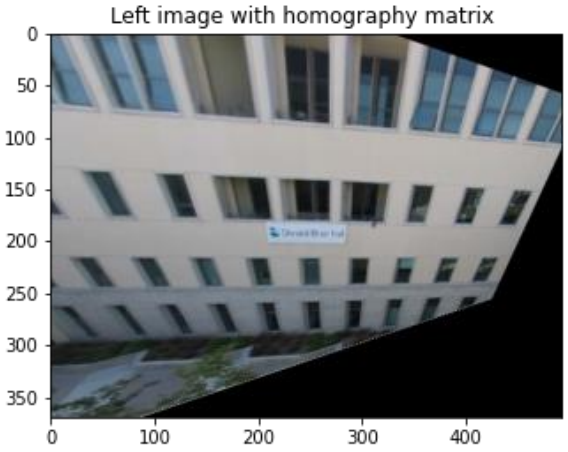
In this case, it is possible that we get a trivial solution with h = 0. Therefore, instead of directly solving the equation, we aim to minimize it for the value of h. By taking the SVD decomposition of A, we can find the minimum solution for H in the last column of V – since the last column of V holds the smallest singular value. Thus, I followed this procedure and then normalised & reshaped the solution from V to get the final result.

2. List the 3x3 camera homography matrix H that you have calculated

```
[ 1.99985469e+01  6.15298599e+00 -2.56404595e+03]
[ 3.96069291e+00  1.30528261e+01 -1.00935137e+03]
[ 2.68917925e-02  2.25193304e-02  1.00000000e+00]
```

3. Warp the left image according to the calculated homography. Study the factors that affect the rectified results, e.g., the distance between the corresponding points, e.g the selected points and the warped ones.

Here is the result after warping the left image with homography matrix



Studying the factors that affect the homography results :

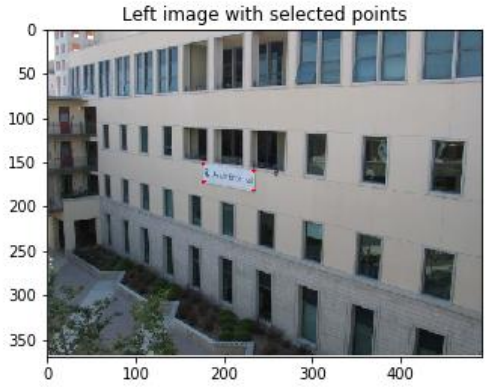
For the homography done above, here are the results -

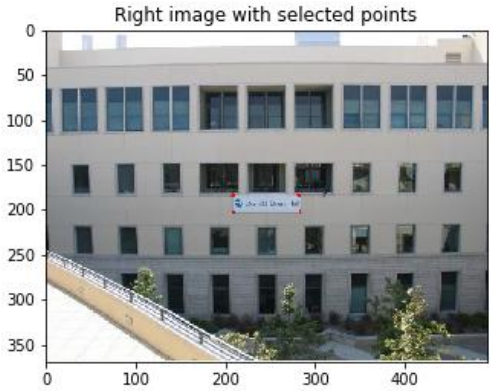
Distance between points : [1 1 3 2 3 2] Mean distance : 2.0

There is some variation between the true points & warped points, but the result is quite good & acceptable in my opinion. Here are two experiments I conduced for further analysis -

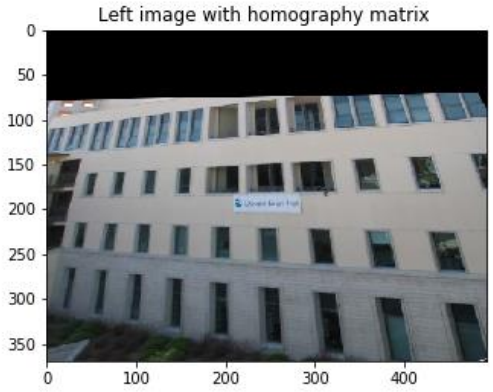
1. Number of points

I experimented with using four points instead of six, and the results were as follows





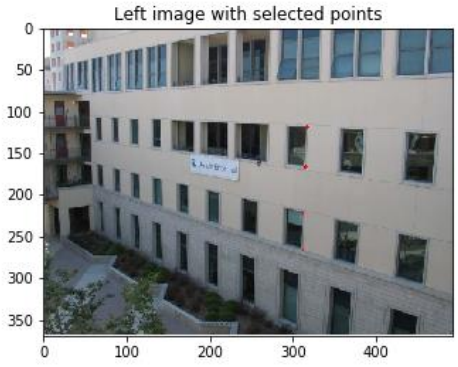
Calculating the homography matrix and warping the left image,

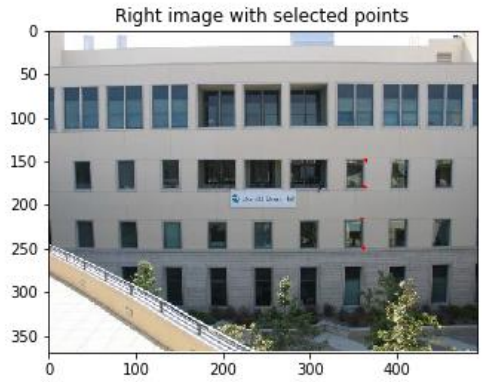


As we can observe, the warping process doesn't work well. Although the mean distance between true points & warped points is 2.0 (the same as six points), the result is rather poor. There is some change of perspective, but the image still looks like it was taken from the left angle. On the other hand, the warped image with 6 points looks much more like it was taken from the front – which is the intended goal here.

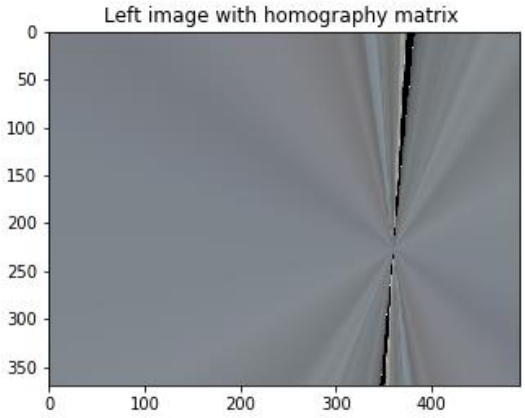
2. Location of points

The points for homography should be chosen strategically, preferably an object or a plane-like structure. If we choose points randomly without any structure, the homography process would not work properly and we will get trivial results. Here is an example –





Trivial homography result:



As we can observe, choosing points randomly leads to the homography algorithm falling apart & resulting in a trivial result.

References

[1] : https://math.stackexchange.com/questions/1801564/solving-for-homography-svd-vs-linear-least-squares-matlab
[2] : https://docs.opencv.org/4.x/d9/dab/tutorial_homography.html
[3] : https://cseweb.ucsd.edu/classes/wi07/cse252a/homography_estimation/homography_estimation.pdf
[4]: http://saurabhg.web.illinois.edu/teaching/ece549/sp2020/slides/lec14_calibration.pdf
[5] : https://answers.opencv.org/question/118918/does-the-resolution-of-an-image-affect-the-distortion-co-efficients/
[6] : https://ksimek.github.io/2012/08/14/decompose/
[7] : https://www.mail-archive.com/floatcanvas@mithis.com/msg00513.html
[8] : https://www.cs.cmu.edu/~16385/s17/Slides/10.2_2D_Alignment__DLT.pdf
[9] : https://github.com/w181496/homography/blob/master/homography.py
[10] : http://saurabhg.web.illinois.edu/teaching/ece549/sp2020/slides/lec14_calibration.pdf