CV HW2 Report

-111065524 資應二 李懷

1. Fundamental Matrix Estimation: (Total 4 image)

(a) Linear least-squares eight-point algorithm

Step 1. Set up A matrix

Define a function lls_eight_point() that takes in 2 inputs, which are correspondent points, we will define mx9 A matrix, where m is the number of points.

```
def lls_eight_point(pts1, pts2):

    # m x 9 (where m>=8)
    A = []
    for i in range(pts1.shape[0]):
        row = [ pts1[i][0]*pts2[i][0], pts1[i][0]*pts2[i][1], pts1[i][0],
                pts1[i][1]*pts2[i][0], pts1[i][1]*pts2[i][1], pts1[i][1],
                pts2[i][0], pts2[i][1], 1 ]
        A.append(row)

    A = np.array(A)
```

Step 2. Compute Fundamental matrix

We then perform SVD on A, then find out Fundamental matrix F by solving a least squares problem, which minimizes |AF-0|^2 with constraint of |F|=1, so F will be the last row of V(or the last column of V^T) which corresponds to the smallest singular value of S (that is sn, the last singular value in S):

```
    U, S, Vt = np.linalg.svd(A)

    # get last row of V
    F = Vt.T[:,-1].reshape(3, 3)
```

Step 3. Enforce Rank-2 constraint

We then enforce F to rank-2 by doing SVD on it, and make the last(3rd) singular value in the singular value matrix 0, and new rank-2 F becomes U S' VT

```
    # compute rank 2 F UD'Vt
    Uf, Sf, Vft = np.linalg.svd(F)
    print(Uf.shape, Sf.shape, Vft.shape)

    # D -> D'
    Sf[-1] = 0

    F = np.dot(np.dot(Uf,np.diag(Sf)),Vft)
    print(f"funcdamental matrix(rank-2): {F}")
    print("least square constraint for SVD:")
    print(f"norm(F): {np.linalg.norm(F)}")
    print(f"norm(AF)^2(minimized to 0) : {np.linalg.norm(np.dot(A,F.flatten()))**2}")
    # print(np.sum(np.dot(A,F.flatten())**2))

    return F
```

Result:

Returned fundamental matrix for given correspondence:

```
Fundamental matrix(wo normalized):
[[ 5.73020056e-07  1.84318195e-06 -3.95681789e-04]
 [ 3.49579115e-06  1.02101960e-06  5.43607932e-03]
 [-2.91687398e-03 -8.06156955e-03  9.99948396e-01]]
least square constraint for SVD:
norm(F): 0.9999999999966476
norm(AF)^2(minimized to 0) : 3.3458382367633823
```

since it is a least-squares problem, we want to find F by solving a least-squares problem where we minimize $|AF|^2$ to 0 under the constraint of $|F|$ being 1(unit norm vector).

(b) Normalized eight-point algorithm

Step 1. Set up T and A matrix

Define a function normalized_eight_point() that takes in 2 inputs, which are correspondent points. We will first define a translation and scale matrix T, which will move the origin of the image to the center and then scale it down to (1,1) (-1,1),(-1,-1),(1,-1) space, each point will be mapped to new normalized point by T, we then generate a mx9 A matrix:

```python
def normalizled_eight_point(pts1, pts2):

    # translation(move origin to the center) and scale
    T = np.array([[2/image1.shape[1], 0, -1], [0, 2/image1.shape[0], -1], [0, 0, 1]])
    # m x 9 (where m>=8)
    A = []
    for i in range(pts1.shape[0]):
        pt1_homo = np.hstack((pts1[i],1))
        pt1_hat = np.dot(T, pt1_homo)[:2]
        pt2_homo = np.hstack((pts2[i],1))
        pt2_hat = np.dot(T, pt2_homo)[:2]
        row = [ pt1_hat[0]*pt2_hat[0], pt1_hat[0]*pt2_hat[1], pt1_hat[0],
                pt1_hat[1]*pt2_hat[0], pt1_hat[1]*pt2_hat[1], pt1_hat[1],
                pt2_hat[0], pt2_hat[1], 1 ]

        A.append(row)

    A = np.array(A)
```

Step 2. Compute Normalized Fundamental matrix

We then compute normalized Fundamental matrix and enforce it to rank-2 via SVD, finally, $T^T F T$ (since $T = T'$ here) to compute the normalized fundamental matrix (Refer to slide p.12):

```python
    U, S, Vt = np.linalg.svd(A)

    # get last row of V
    F = Vt.T[:,-1].reshape(3, 3)

    # compute rank 2 F UD'Vt
    Uf, Sf, Vft = np.linalg.svd(F)
    print(Uf.shape, Sf.shape, Vft.shape)
    # D -> D'
    Sf[-1] = 0
    F = np.dot(np.dot(Uf,np.diag(Sf)),Vft)

    F = np.dot(np.dot(T.T, F), T)
    print(f"normalized funcdamental matrix(rank-2): {F}")

    return F
```

Result:

Returned normalized fundamental matrix(rank-2) for given correspondence:

```
Fundamental matrix(normalized):
[[-3.98266443e-08 -2.02336406e-07 -9.63507955e-05]
 [-3.60316474e-07  1.17698782e-08 -1.61464340e-03]
 [ 1.61100923e-04  1.83927254e-03 -2.19121017e-02]]
```

(c) Plot the epipolar lines for (a) and (b)

Step 1. Compute epipolar lines

We define a function compute_line() that takes into 3 inputs, the 2 correspondence sets, and the fundamental matrix that will be used to calculate the epipolar lines for two sets of correspondences by computing Fp' for points1, and F^Tp for points2, it will be it will return epipolar lines for points1 and points2:

```
def compute_line(pts1, pts2, F):
    epi_lines_1 = []
    epi_lines_2 = []

    # l = Fp, l' =F^Tp'
    for i in range(pts1.shape[0]):
        pt1_homo = np.hstack((pts1[i],1))
        pt2_homo = np.hstack((pts2[i],1))
        epi_lines_1.append(np.dot(F, pt2_homo))
        epi_lines_2.append(np.dot(F.T, pt1_homo))

    epi_lines_1 = np.array(epi_lines_1)
    epi_lines_2 = np.array(epi_lines_2)

    return epi_lines_1, epi_lines_2
```

Step 2. Draw lines

We define a function drawlines() which will draw feature points and their corresponding epipolar lines on the image, and also calculate the average distance between them:

```python
def drawlines(image,lines,pts1):
    ''' img1 - image on which we draw the epilines for the points in img2
        lines - corresponding epilines '''
    img = image.copy()
    total_distance = 0
    total_points = len(pts1)
    colors = [(int(255 * (total_points - i) / total_points), 0, int(255 * i / total_points)) for i in range(total_points)]

    for line, pt1, color in zip(lines, pts1, colors):

        # we draw lines from leftmost to rightmost, so x0 = 0
        x0,y0 = 0, int(-line[2]/line[1])
        # x1 = the rightmost pixel
        x1,y1 = img.shape[1], int(-(line[2]+line[0]*img.shape[1])/line[1])

        # distance of current correspondance point to its epipolar line
        total_distance += abs(np.cross(np.array([x1,y1])-np.array([x0,y0]), pt1 - np.array([x0,y0]))) / np.linalg.norm(np.ar

        img = cv2.line(img, (x0,y0), (x1,y1), color,1)
        img = cv2.circle(img,(int(pt1[0]), int(pt1[1])),5,color,-1)

    avg_distance = total_distance/len(pts1)

    return img, avg_distance
```

Create graduation of colors from red to blue, this will be used to plot the epipolar lines with different colors for the given point correspondences:

```python
    total_distance = 0
    total_points = len(pts1)
    colors = [(int(255 * (total_points - i) / total_points), 0, int(255 * i / total_points)) for i in range(total_points)]
```

And we draw each feature point and its epipolar line one by one, we draw line from the leftmost y ordinate to the rightmost y ordinate of the epipolar line, and calculate their distance along the way:

```python
    for line, pt1, color in zip(lines, pts1, colors):

        # we draw lines from leftmost to rightmost, so x0 = 0
        x0,y0 = 0, int(-line[2]/line[1])
        # x1 = the rightmost pixel
        x1,y1 = img.shape[1], int(-(line[2]+line[0]*img.shape[1])/line[1])

        # distance of current correspondance point to its epipolar line
        total_distance += abs(np.cross(np.array([x1,y1])-np.array([x0,y0]), pt1 - r

        img = cv2.line(img, (x0,y0), (x1,y1), color,1)
        img = cv2.circle(img,(int(pt1[0]), int(pt1[1])),5,color,-1)
```

We calculate the distance of each feature point to its epipolar line using $|\vec{a}\ \mathrm{x}\ \vec{b}|$ / $|\vec{b}|$, and add them up to get total distance:

```python
        # distance of current correspondance point to its epipolar line
        total_distance += abs(np.cross(np.array([x1,y1])-np.array([x0,y0]), pt1 - np.array([x0,y0]))) / np.linalg.norm(np.array([x1,y1]) - np.array([x0,y0]))
```

We then calculate the average distance for all feature points by dividing the total number of points, finally, return avg_distance and the drawn image:

```
        total_distance += abs(np.cross(np.array([x1,y1])-np.array([x0,

        img = cv2.line(img, (x0,y0), (x1,y1), color,1)
        img = cv2.circle(img,(int(pt1[0]), int(pt1[1])),5,color,-1)

    avg_distance = total_distance/len(pts1)

    return img, avg_distance
```

Step 3. Try out the functions

After defining all the functions, we test them out in __main__, we first read points from text files:

```
if __name__=="__main__":

    pt_txt1 = open("assets/pt_2D_1.txt")
    pt_txt2 = open("assets/pt_2D_2.txt")
    image1 = cv2.imread("assets/image1.jpg")
    image2 = cv2.imread("assets/image2.jpg")

    print(image1.shape)

    image1 = cv2.cvtColor(image1, cv2.COLOR_BGR2RGB)
    image2 = cv2.cvtColor(image2, cv2.COLOR_BGR2RGB)

    pts1 = []
    pts2 = []

    for line in zip(pt_txt1.readlines()[1:], pt_txt2.readlines()[1:]):
        x1, y1 = map(float, line[0].strip().split())
        x2, y2 = map(float, line[1].strip().split())

        pts1.append([x1,y1])
        pts2.append([x2,y2])


    pts1 = np.array(pts1, dtype=np.float32)
    pts2 = np.array(pts2, dtype=np.float32)
```

We then feed our points into lls_eight_point() and normalized_ eight_point() respectively, and call compute_line() to calculate estimated epipolar lines for each feature point, then finally, we call drawlines() to draw them on to the image and get the avg. distance:

Unnormalized:

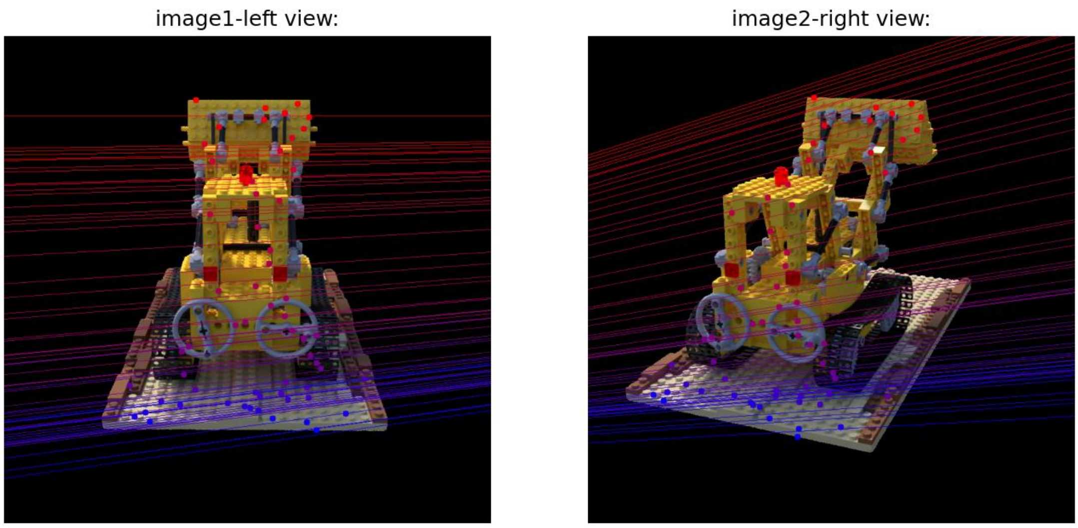
```python
    F = lls_eight_point(pts1, pts2)

    epi_lines_1, epi_lines_2 = compute_line(pts1, pts2, F)

    image_l, avg_distance = drawlines(image1,epi_lines_1,pts1)
    print(f"image left - average distance: {avg_distance}")
    image_r, avg_distance = drawlines(image2,epi_lines_2,pts2)
    print(f"image right - average distance: {avg_distance}")
```

Result:

Result image for unnormalized eight point:



Print out the average distance between the feature points and their corresponding epipolar lines for point1 and points2:

```
image left - average distance: 24.879910920938364
image right - average distance: 25.965776700669988
```

Normalized:

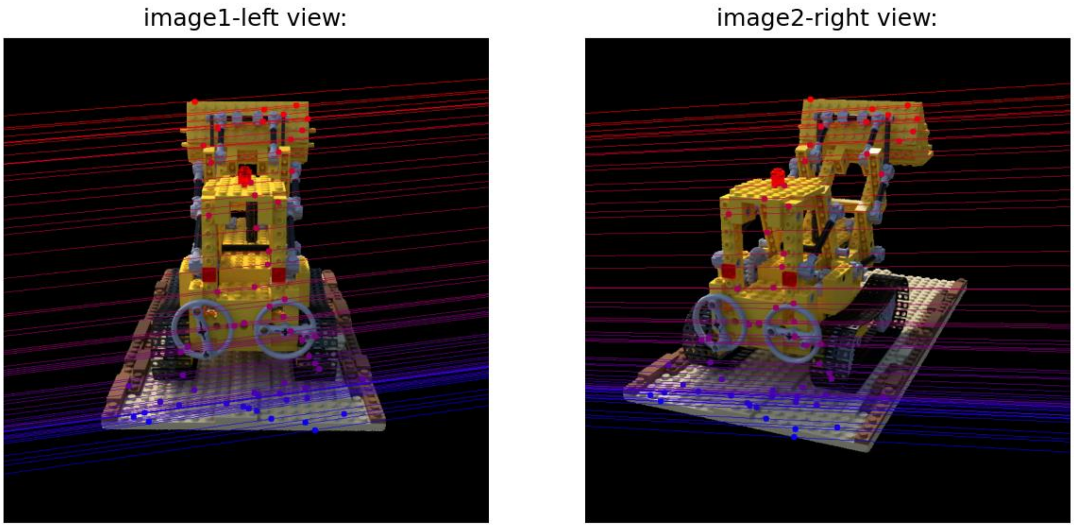
```
    F_n = normalizled_eight_point(pts1, pts2)

    epi_lines_1_n, epi_lines_2_n = compute_line(pts1, pts2, F_n)

    image_l, avg_distance = drawlines(image1,epi_lines_1_n,pts1)
    print(f"image left - average distance: {avg_distance}")
    image_r, avg_distance = drawlines(image2,epi_lines_2_n,pts2)
    print(f"image right - average distance: {avg_distance}")
```

Result:

Result image for normalized eight point:



Print out the average distance between the feature points and their corresponding epipolar lines for point1 and points2, the results are much better than unnormalized one above (more lines passing through the feature points and are closer in averge)

```
image left - average distance: 1.0453876241757725
image right - average distance: 1.0335475004265173
```

2. Homography transform:(Total 1 image)

(a) Implement a function that estimates the homography matrix H that maps a set of interest points to a new set of interest points.

Define a function Find_Homography() that takes in 2 inputs, world and camera, which are set of point correspondences, and return a 3x3 homography matrix H that maps world's points to its correspondent points on camera's points, that is, p' = Hp:

```
def Find_Homography(world,camera):
    '''
    given corresponding point and return the homagraphic matrix
    '''

    A = []
    for p1,p2 in zip(world, camera):
        row1 = [p1[0], p1[1], 1, 0, 0, 0, -p2[0]*p1[0], -p2[0]*p1[1], -p2[0]]
        row2 = [0, 0, 0, p1[0], p1[1], 1, -p2[1]*p1[0], -p2[1]*p1[1], -p2[1]]
        A.append(row1)
        A.append(row2)

    A = np.array(A)

    U, S, Vt = np.linalg.svd(A)
    # print(U.shape, S.shape, Vt.shape)

    # row of V that corresponds to the smallest singular value of S (that is eigenvect
    H = Vt.T[:,-1].reshape(3, 3)

    print("least square constraint for SVD:")
    print(f"norm(H): {np.linalg.norm(H)}")
    print(f"norm(AH)^2(minimized to 0) : {np.linalg.norm(np.dot(A,H.flatten()))**2}")

    return H
```

Step 1.

We create A matrix with shape of 2n,9 (n is the number of correspondences, n>=4):

```
    A = []
    for p1,p2 in zip(world, camera):
        row1 = [p1[0], p1[1], 1, 0, 0, 0, -p2[0]*p1[0], -p2[0]*p1[1], -p2[0]]
        row2 = [0, 0, 0, p1[0], p1[1], 1, -p2[1]*p1[0], -p2[1]*p1[1], -p2[1]]
        A.append(row1)
        A.append(row2)

    A = np.array(A)
```

Step 2.

We then perform SVD on A, then find out H by solving a least squares problem, which minimizes |AH-0|^2 with constraint of |H|=1, the solution for h(or $\hat{h}$) is the eigenvector of $A^T A$ with the smallest eigenvalue, so it is the last row of V(or the last column of V^T) which corresponds to the smallest singular value of S (that is sn, the third one in singular value matrix):

```
    U, S, Vt = np.linalg.svd(A)
    # print(U.shape, S.shape, Vt.shape)

    # row of V that corresponds to the smallest singular value of S (that is eigenvector
    H = Vt.T[:,-1].reshape(3, 3)

    print("least square constraint for SVD:")
    print(f"norm(H): {np.linalg.norm(H)}")
    print(f"norm(AH)^2(minimized to 0) : {np.linalg.norm(np.dot(A,H.flatten()))**2}")

    return H
```

(b) Select 4 corresponding straight lines from display image (that is 4 corner points) and maps these points to the 4 corner points of the source image (that is (0,0), (m,0), (m,n), (0,n) of CV image, where m is width and n is height) by computing the homography matrix.

Step 1. Prepare work

We first sort the selected corner points in the order of left-top -> right-top -> right-down -> left-down after clicking 4 points (this ensures the order if you click in random order):

```
        if(len(corner_list)==4):

            corner_list = np.array(corner_list)

            # order corner points from left top, right top, right down, left down
            # sort by y ascending first (y1 then y2)
            corner_list = corner_list[np.argsort(corner_list[:,-1])]
            # for first 2, sort by x ascending (x1,y1 then x2,y1)
            corner_list[:2] = corner_list[np.argsort(corner_list[:2, 0])]
            # for last 2, sort by x descending (x2,y2 then x1,y2)
            corner_list[2:] = corner_list[2:][np.argsort(corner_list[2:, 0])[::-1]]
```

```
selected corner points (order: left top -> right top -> right down -> left down):
[[ 829  407]
 [ 982  441]
 [1002  628]
 [ 709  695]]
```

Step 2. Compute homography matrix

We then use the Find_Homography function that maps 4 selected corner points of the display to the 4 corners of the CV image, the reason we map display image points to CV image instead of the other way around is that we want to perform inverse mapping (or inverse warping) in order to use bi-linear interpolation on the CV image, so our original mapped destination points(display image) now becomes the source, and the source points becomes the destination:

```
        # inverse homography mapping
        H = Find_Homography(corner_list, corners_src)
```

※corner_list stores the 4 corner points of the display image, corners_src stores the 4 corner points of CV image :

```
    img_src = cv2.imread("assets/post.png")
    src_H,src_W,channels=img_src.shape

    corners_src = np.array([[0, 0],[src_W-1, 0],[src_W-1, src_H-1],[0, src_H-1]])
```

Result:

Output of homography matrix:

```
homography matrix:
[[-9.78904957e-04 -4.07877066e-04  9.77518175e-01]
 [ 2.10319837e-04 -9.46439265e-04  2.10845636e-01]
 [ 6.39802343e-07 -7.09945934e-07 -5.20006101e-04]]
least square constraint for SVD:
norm(H): 1.0000000000000002
norm(AH)^2(minimized to 0) : 6.583715248454858e-22
```

Step 3. Create coordinates

We then use create_region() function that is used to create coordinates within the MBR (minimum bounding rectangle, that is the red box in below result image) of the corner points:

create_region() uses numpy.meshgrid and numpy.arange to create X and Y (both shape of m,n), X's each row ranges from min_x to max_x (min_x + m), and has n rows, Y's each column ranges from min_y to max_y (min_y + n), and has m columns:

```python
def create_region(corner_points):
    '''
    given 4 corner points, create a box region that contains them(meshgrid)
    '''

    min_x, min_y = corner_points.min(axis=0)
    max_x, max_y = corner_points.max(axis=0)

    # X,Y are all the point coordinates within the red box region
    X, Y = np.meshgrid(np.arange(min_x, max_x+1), np.arange(min_y, max_y+1))

    return X, Y
```

We use np.column_stack to stack xi and yi to create every coordinates' points, then convert them to homogeneous coordinates (xi, yi, 1):

```python
            # create all coordinates within red box region on screen image
            X, Y = create_region(corner_list)
            points = np.column_stack((X.flatten(), Y.flatten()))

            # change to homogenous
            homo_points = np.column_stack([points, np.ones(len(points))])
            # print(homo_points.shape)
```

Step 4. Compute mapped coordinates using homography

We then compute estimated mapped coordinates of all homogenous points of the display image(that is points within red box region) on the CV image using the homography we just computed and convert them back to 2D by dividing x,y with the homogeneous:

```python
            # homography matrix maps screen image pixel to CV image(since inverse mapping)
            output_points = np.dot(H, homo_points.T).T

            # change back from homogeneous to 2D
            output_points = output_points[:,:2]/output_points[:,[-1]]
```

Step 5. Map the image

We then assign each CV image pixel value(using bi-linear interpolation) to its corresponding coordinate on the display image, the if statement filter out those that was mapped outside the CV image boundaries (since those pixels that are outside 4 green lines region and within the red box region will be mapped to coordinates outside the CV image boundaries, and we don't care about those points):

```
        # bi-linear interpolation
        for point_src, point_target in zip(points, output_points):
            if point_target[0]<img_src.shape[1]-1 and point_target[1]<img_src.shape[0]-1 and point_target[1]>=0 and point_target[0]>=0:
                # image[y, x]
                fig[point_src[1], point_src[0]] = bilinear(img_src,point_target)
```

Compute every pixel value of CV image using bi-linear interpolation and assign them to their corresponding coordinates on the display image.

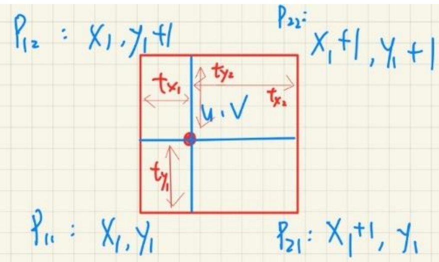
```
def bilinear(img, pt):
    x1 = int(pt[0])
    x2 = int(pt[0]) + 1
    y1 = int(pt[1])
    y2 = int(pt[1]) + 1

    a = pt[0] - x1
    b= pt[1] - y1

    new_pt = (1-a)*(1-b)*img[y1, x1] + a*(1-b)*img[y1, x2] + b*(1-a)*img[y2, x1] + a*b*img[y2, x2]

    return new_pt
```

Bi-linear interpolation:



$t_x = u - x_1$

$t_y = v - y_1$

$t_{x_2} = 1 - t_{x_1}$

$t_{y_2} = 1 - t_{y_1}$

$$f(u,v) = \frac{f(P_{11})}{1\times 1} t_{x_2} t_{y_2} + \frac{f(P_{21})}{1\times 1} t_{x_1} t_{y_2} + \frac{f(P_{12})}{1\times 1} t_{x_2} t_{y_1} + \frac{f(P_{22})}{1\times 1} t_{x_1} t_{y_1}$$

Step 6. Draw lines and vanishing point

We then draw 4 green lines that connect 4 corner points we selected, and draw the intersection of two parallel lines (parallel in 3D space, top and bottom edge of the display), which is their vanishing points (see **(c)** for implementation):

```
# draw four corresponding straight lines(green lines)
fig = cv2.line(fig, corner_list[0], corner_list[1], (0, 255, 0), 2)
fig = cv2.line(fig, corner_list[1], corner_list[2], (0, 255, 0), 2)
fig = cv2.line(fig, corner_list[2], corner_list[3], (0, 255, 0), 2)
fig = cv2.line(fig, corner_list[3], corner_list[0], (0, 255, 0), 2)

# compute vanishing point using monitor's top and bottom parallel lines
intersect = intersection((corner_list[0], corner_list[1]), (corner_list[3], corner_list[2]))
print(f"vanishing point: {intersect}")
cv2.circle(fig,intersect, 5,(0,255,0),3)
```

Draw MBR for 4 selected points (red box):

```
# drawing red box region
min_x, min_y = corner_list.min(axis=0)
max_x, max_y = corner_list.max(axis=0)
fig = cv2.line(fig, (min_x, min_y), (max_x, min_y), (0, 0, 255), 2)
fig = cv2.line(fig, (max_x, min_y), (max_x, max_y), (0, 0, 255), 2)
fig = cv2.line(fig, (max_x, max_y), (min_x, max_y), (0, 0, 255), 2)
fig = cv2.line(fig, (min_x, max_y), (min_x, min_y), (0, 0, 255), 2)

break
```

Result Image:

with mapped CV image on screen, the selected correspondence line pairs(green), MBR region(red box) and the vanishing point for upper and bottom screen edge(green point on the right):

(c) Compute the vanishing point of two parallel lines (upper and bottom) of the screen in the image, show the vanishing point's coordinate and mark it on the image.

Define a function intersection() that takes into 2 parallel(in 3D) lines and compute the estimated vanishing point (see above image)

```python
def intersection(line1, line2):
    '''
    Finds the intersection of two lines, i.e. vanishing line
    '''
    p1, q1 = line1
    p2, q2 = line2

    p1 = np.hstack((p1,1))
    q1 = np.hstack((q1,1))
    p2 = np.hstack((p2,1))
    q2 = np.hstack((q2,1))

    intersection = np.cross(np.cross(p1, q1), np.cross(p2, q2))

    intersection = intersection[:-1]/intersection[-1]
    intersection = intersection.astype(int)
    return intersection
```

```
vanishing point: [1406  535]
```

Here we only use two lines, more parallel lines (in 3D space) the more accurate the estimated vanishing point is.