ECE 661 - HW3

Bhavya Patel - pate1539@purdue.edu

9/13/24

Logic

Point to Point Logic

This homography creation logic closely follows the process we took for Hw02, so much of the logic will be repeated.
A Homography is a 3x3 non-singular matrix that does linear transformations on homogeneous 3-vectors. This allows a homography matrix to convert a homogeneous point in a domain space to a homogeneous point in the range space. We can define the domain space as x = (x, y, x)^T, the range space as x' = (x', y', z')^T, and the homography matrix as H. So we can rewrite this equation for linear transformations between images as

$$\mathbf{x'} = \mathbf{Hx}$$

Now lets define the Homography matrix, H, as

$$\mathbf{H} = \begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ v_1 & v_2 & 1 \end{bmatrix}$$

So now let's write the full equation for x' = Hx.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ v_1 & v_2 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

When converting homogenous cooridnates from 3d vector to 2d vector you know to find the point (x,y) in physical space $\mathbb{R}^2$ the x = $\frac{a}{c}$ and y = $\frac{b}{c}$. Where the 3d vector is (a,b,c)^T. So converting the z into a 1 will just make the conversion between the 3d vector into 2d physical points easier by simplifying the equation. Now let us rewrite the matrices into linear equations in the real coordinates (2d) by expanding the matrices and then simplifying.

$$\mathbf{x'_{2d}} = a_{1,1}x + a_{1,2}y + t_x - v_1xx' - v_2yx'$$
$$\mathbf{y'_{2d}} = a_{2,1}x + a_{2,2}y + t_y - v_1xy' - v_2yy'$$

We can create a system of equation and solve for the 8 unknown homography variables. Which means we need 4 points of the image from before and after the translation done by the homography. Each set of points results in 2 equations, so 4*2 = 8. Let those 4 points be represented by (x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4) and they map to (x'_1, y'_1), (x'_2, y'_2), (x'_3, y'_3), (x'_4, y'_4). The system of equations in matrix form is

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x_1' & -y_1x_1' \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y_1' & -y_1y_1' \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x_2' & -y_2x_2' \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y_2' & -y_2y_2' \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x_3' & -y_3x_3' \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y_3' & -y_3y_3' \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x_4' & -y_4x_4' \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y_4' & -y_4y_4' \end{bmatrix} \begin{bmatrix} a_{1,1} \\ a_{1,2} \\ t_x \\ a_{2,1} \\ a_{2,2} \\ t_y \\ v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} x_1' \\ y_1' \\ x_2' \\ y_2' \\ x_3' \\ y_3' \\ x_4' \\ y_4' \end{bmatrix}$$

The above is in the form of Ab = c which can be turned into b = A^{-1}c to solve for the vector b which contains the variables in the homography matrix. Then once you have the variables you have the H, so you can apply x' = Hx to transform all the pixels in the image to another picture. However, using this method I found there to be lots of bleeding. To fix this you can convert the equation to H^{-1}x' = x. So now you can take the pixels that you want to replace and find their corresponding pixel in the source image. This is how the transformations are carried out throughout this homework.

Two Step Logic

The general idea for a two-step transformation is first to remove the projective distortion by using the vanishing line method. Then take that new image and remove the remaining affine distortion by using a cos(θ) expression that uses the Dual Degenerate Conic C^*_∞ to set the θ back to 90°.

Step 1: Removing Projective Distortion using the vanishing line method.

A homography is affine if and only if l_∞ is mapped to l_∞. However, a projective transformation maps l_∞ to a physical line called the vanishing line. This means we can use a homography to send the vanishing line back to l_∞ = (0, 0, $0)^T$ to remove the projective distortion. The vanishing line is the line connecting the two vanishing points of an image. A vanishing point is the intersection point of parallel lines l and m in the non-distorted image. The vanishing line can be found by taking the cross product of the two vanishing points. Let us define the vanishing line as $l_{vanish} = (l_1,l_2,l_3)^T$. Then the homography to remove the projective distortion is written below.

$$\mathbf{H_{step1}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l_1 & l_2 & l_3 \end{bmatrix}$$

Then we want to apply this homography on lines and l' = H^{-T}l, so lets convert this H to H^{-T}.

$$\mathbf{H_{step1}^{-T}} = \begin{bmatrix} 1 & 0 & -\frac{l_1}{l_3} \\ 0 & 1 & -\frac{l_2}{l_3} \\ 0 & 0 & l_3 \end{bmatrix}$$

Then you can apply H_{step1}^{-T} to the parallel lines l and m to get l' and m', which should only have an affine distortion.

Step 2: Removing Affine Distortion using Dual Degenerate Conic C_∞^*.

Let us write the equation for cos(θ) in terms of the Dual Degenerate Conic C_∞^* and its lines l and m.

$$\mathbf{cos}(\theta) = \frac{l^T C_\infty^* m}{\sqrt{(l^T C_\infty^* l)(m^T C_\infty^* m)}}$$

Let us let cos(θ) be equal to 0 to represent the final angles being at 90°. This means that we need the numerator to be equal to zero to make this statement true, so we can ignore the denominator. Using the equations l = H^Tl', m = H^Tm', and $C_\infty^* = H^{-1} C_\infty^{*'} H^{-T}$ we can simplify the equation to

$$\mathbf{0} = l^{'T} H C_\infty^* H^T m'$$

Then we can expand and simplify this equation to get the below.

$$\begin{bmatrix} l_1' & l_2' & l_3' \end{bmatrix} \begin{bmatrix} AA^T & 0_{2x1} \\ 0_{1x2} & 0 \end{bmatrix} \begin{bmatrix} m_1' \\ m_2' \\ m_3' \end{bmatrix} = 0$$

Then we know that AA^T = S, which means that $s_{12} = s_{21}$. We can also write s_{22} as 1 because only the ratios matter. Now we can rewrite the above equation.

Logic

$$\begin{bmatrix} l_1' & l_2' \end{bmatrix} \begin{bmatrix} s_{11} & s_{12} \\ s_{12} & 1 \end{bmatrix} \begin{bmatrix} m_1' \\ m_2' \end{bmatrix} = 0$$

This can be rewritten into the linear equation below.

$$s_{11} l_1' m_1' + s_{12}(l_1' m_2' + l_2' m_1') = -l_2' m_2'$$

In the equation above there is 2 unknowns, so you need 2 pairs of parallel lines l and m to solve the system. Then once you solve it you will have an S matrix which is equal to AA^T. Then A is positive definite, so we can do an eigendecomposition to get A = VDV^T. Then AA^T = VDV^{T}VDV^T = VD^{2}V^T. Then you know D = $\begin{pmatrix} \lambda_1^2 & 0 \\ 0 & \lambda_2^2 \end{pmatrix}$. VV^T = I, so we know that by doing the eigendecomposition of S, we get eigenvectors of A, with its eigenvalues given by the positive square roots of the eigenvalues of S. Now we know that A = V$\begin{pmatrix} \sqrt{\lambda_1^2} & 0 \\ 0 & \sqrt{\lambda_2^2} \end{pmatrix}$V^T. So knowing A we can plug it into the homography that will remove the affine distortion of the image.

$$\mathbf{H_{step2}} = \begin{bmatrix} A_{11} & A_{12} & 0 \\ A_{21} & A_{22} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Now you can multiply the H$_{step1}$ and H$_{step1}$ homographies to get the combined homography which I will apply to the original image to remove the distortion.

One Step Logic

The one-step method removes both the projective and affine distortions at once rather than individually like the two-step method. It achieves this by using the homography that maps C$_\infty^{'*}$ back to C$_\infty^*$. We can write this mapping with the equation C$_\infty^{'*}$ = HC$_\infty^*$H^T. We want to solve for C$_\infty^{'*}$ which is

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

Logic

Given,

$$\mathbf{C}^*_\infty = \begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 0\end{bmatrix}, \mathbf{H} = \begin{bmatrix}A_{11} & A_{12} & 0\\ A_{21} & A_{22} & 0\\ v_0 & v_1 & 1\end{bmatrix}$$

we can expand and simplify the mapping equation to get

$$\begin{bmatrix}AA^T & Av\\ v^TA^T & v^Tv\end{bmatrix}$$

We can now calculate A and v in the homography, H, matrix using

$$\mathbf{AA^T} = \begin{bmatrix}a & \frac{b}{2}\\ \frac{b}{2} & c\end{bmatrix}, \mathbf{Av} = \begin{bmatrix}\frac{d}{2}\\ \frac{e}{2}\end{bmatrix}$$

Now we need to solve for the variables a, b, c, d, e, and f. Let us expand the equation $\mathrm{l}'^T\mathrm{C}'^*_\infty\mathrm{m}' = 0$ where l' and m' are orthogonal line pairs in the undistorted post-homography image. We can set f = 1 because only ratios matter in homographies.

$$\begin{bmatrix}l'_1 & l'_2 & 1\end{bmatrix}\begin{bmatrix}a & \frac{b}{2} & \frac{d}{2}\\ \frac{b}{2} & c & \frac{e}{2}\\ \frac{d}{2} & \frac{e}{2} & 1\end{bmatrix}\begin{bmatrix}m'_1\\ m'_2\\ 1\end{bmatrix} = 0$$

We can see that there are five unknown variables we need to solve for, so we will need five l' and m' orthogonal line pairs to solve the system of equations that is written below. Note: There should be 5 equations below with the same form.

$$a(l'_1m'_1) + \frac{b}{2}(l'_1m'_1 + l'_1m'_1) + c(l'_1m'_1) + \frac{d}{2}(l'_1m'_1) + \frac{e}{2}(l'_1m'_1) = -1$$

After you have the values of a, b, c, d, e, and f, so you will be able to solve for A using the eigendecomposition of AA^T. Then with A you can solve for v. Now you can reconstruct H using A and v and apply the homography on the original image to remove all distortions.

Vectorization and Discussion on Results

Vectorization

I used vectorized numpy operations when applying my homography to an image. This can be found in the source code function ”apply-homography’. Instead of looping through the pixels of the image one by one, I vectorized the image making the computation much quicker.

Discussion on Results

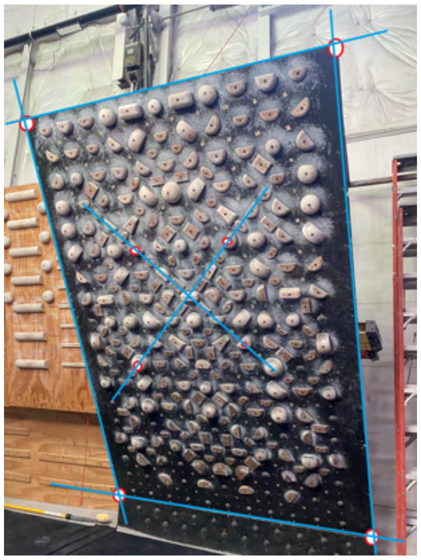
In this section I will just talk about some of the things that I noticed and clarify things I thought would be important to discuss. Some of my final images might seem to not have a 90-degree angle, however, I have checked the angle between orthogonal lines after the homography had been applied. This confirms that all of the results that should have no distortion are correct. I found that when applying the homography sometimes the resulting image would be too big for my computer to run because of RAM limitations. Thus, I used the scaling homography that was discussed on pizza to make the computation possible. I found that scaling the homography would have minor effects on the results, which could be a reason that the images could look deceiving. I found this more true with the two-step approach because you would have to apply the homography on pairs of lines and then apply another homography. So the slight inaccuracies were amplified, but I managed to mitigate them to where the images seem to look 90 degrees. However, the actual angle between the pair of lines could be in the range of 87-89 degrees.
I would also like to mention that I was unable to get the corridor images to properly work for any of the methods outlined in the homework document. I narrowed down the issue to my apply-homography function for the point to point method. The generated homography seemed to be accurate because I would apply the generated homography using cv2.warpPerspective() get an accurate image. However, with my function, I found that I could only get a black image or a black image with a small section with a very distorted and indiscernible image. However, when applying the homography on the lines created by the points the angle between them was 90 degrees. For the two-step approach, I could not find any solution and the angle between the lines was 60 degrees when applying the homography on the lines. I suspect that it was not 90 degrees because of all the homography scaling done in this function. I found that a scaled homography minor distortion messes up the angle between the lines. For the one-step approach applying the final homography on the lines resulted in a

Vectorization and Discussion on Results

angle of 90 degrees. This means the homography is correct, but I was unable to get the cv.warpPerspective() to work. **So for the corridor image results, if there is a result I used the cv.warpPerspective() function.**

Task 1

Images and Points used in Task 1


(a) Board Image

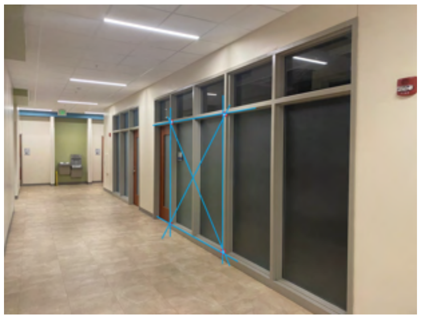

(b) Corridor Image

Table 1: Points used in Task 1

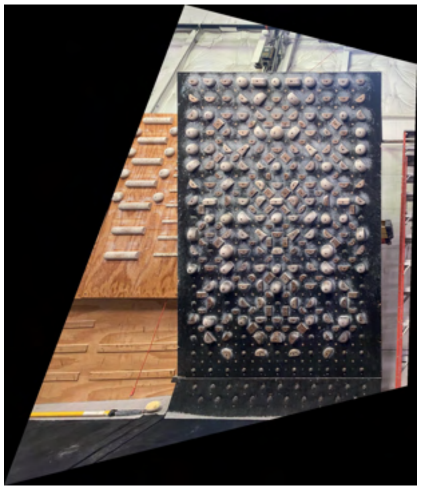
Image	**P**	**Q**	**R**	**S**
Board	(73,425)	(1221,146)	(1354,1956)	(423,1798)
Corridor	(815,576)	(1073,531)	(1068,1212)	(811,1069)

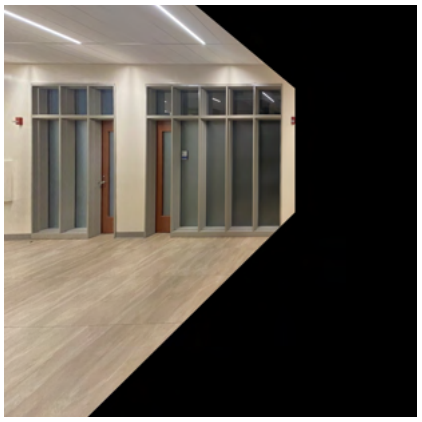
Note: P,Q,R,S starts in the upper left most point and goes clockwise.

Task 1

Point-to-Point Correspondences

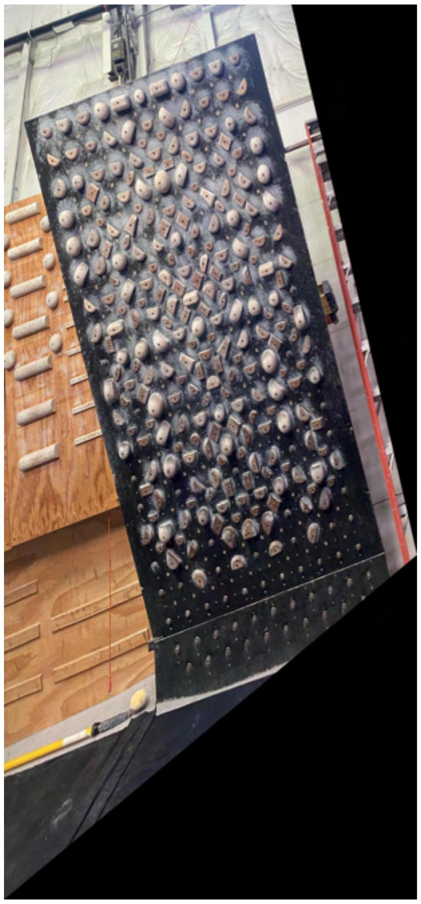
Point to Point Correspondences Results


(a) Board Point to Point Image


(b) Corridor Point to Point Image

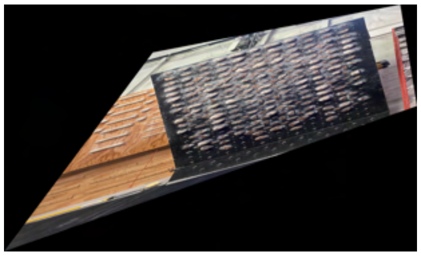
Task 1

Two Step Approach



(a) Board without Projective Distortion

(b) Corridor without Projective Distortion



(c) Board without Any Distortion

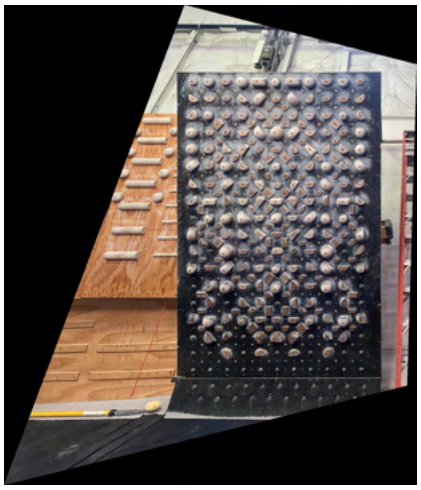
(d) Board without Any Distortion

Figure 3: Two Step Approach Results

Task 1

One Step Approach

One Step Approach Results

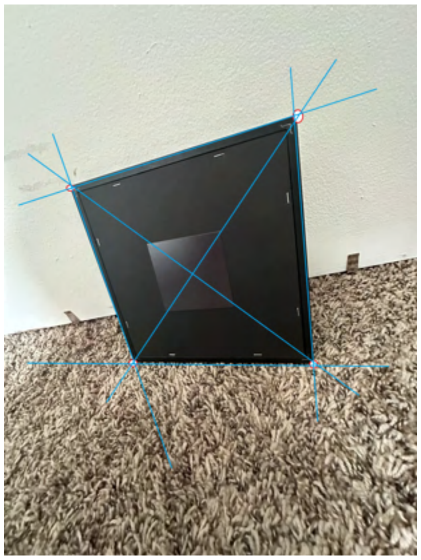


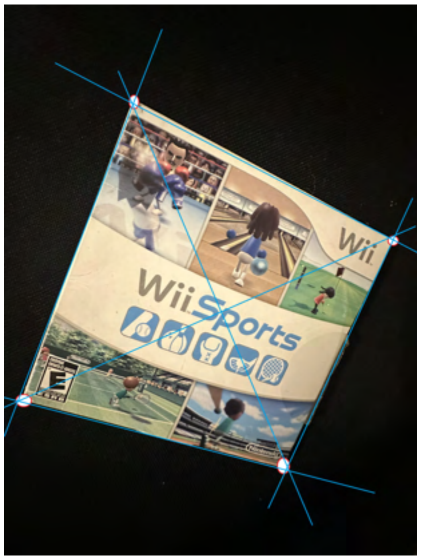
(a) Board One Step Approach Image

(b) Corridor One Step Approach Image

Task 2

Images and Points used in Task 2


(a) My Image 1



(b) My Image 2

Table 2: Points used in Task 2

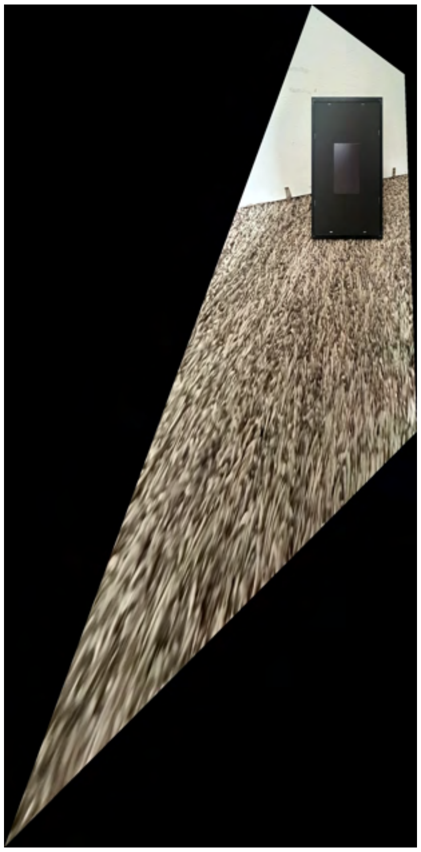
Image	P	Q	R	S
My Image 1	(481,1330)	(2141,805)	(2266,2627)	(950,2612)
My Image 2	(954,704)	(2841,1729)	(2038,3380)	(141,2889)

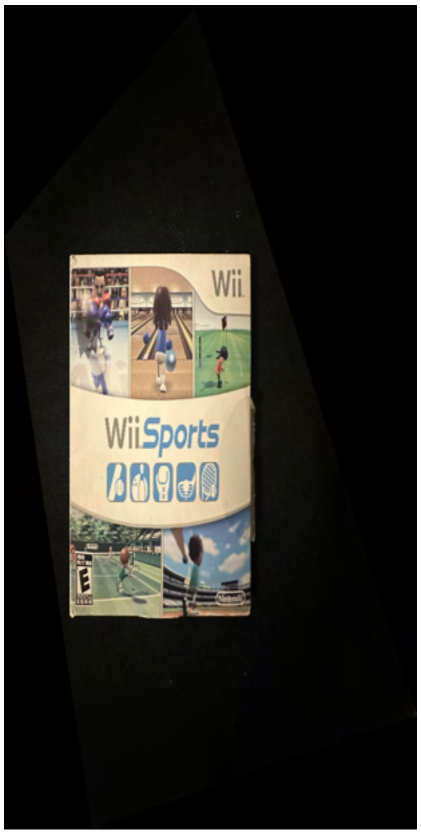
Note: P,Q,R,S starts in the upper left most point and goes clockwise.

Task 2

Point-to-Point Correspondences

Point to Point Correspondences Results


(a) My Image 1 Point to Point

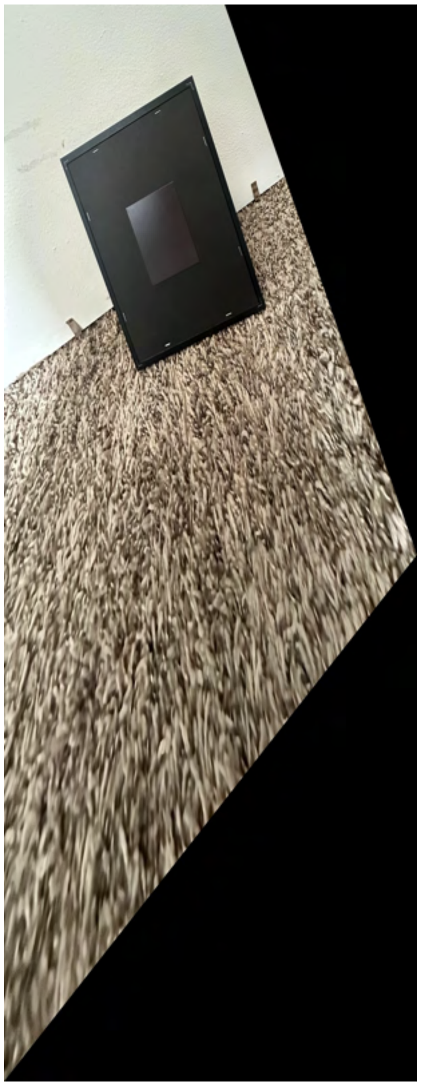


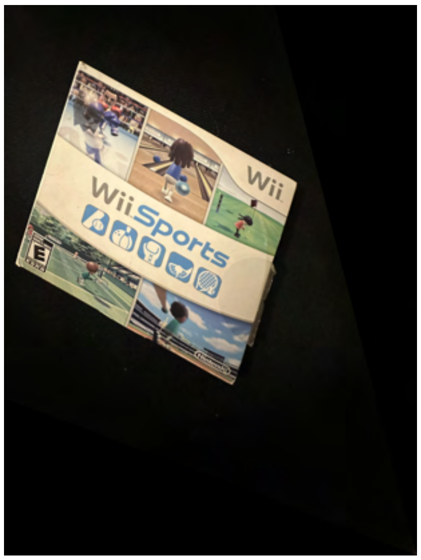
(b) My Image 2 Point to Point

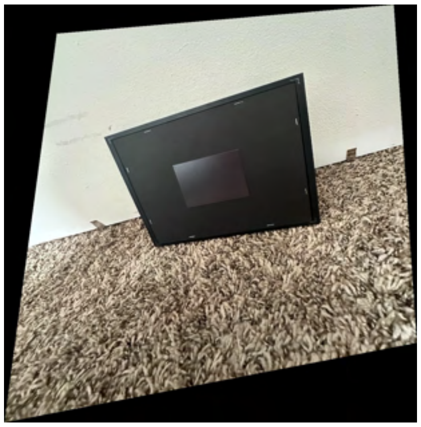
Task 2

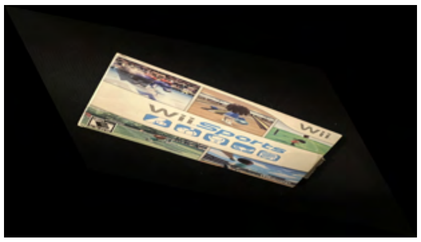
Two Step Approach

Two Step Approach Results


(a) My Image 1 without Projective Distortion


(b) My Image 2 without Projective Distortion

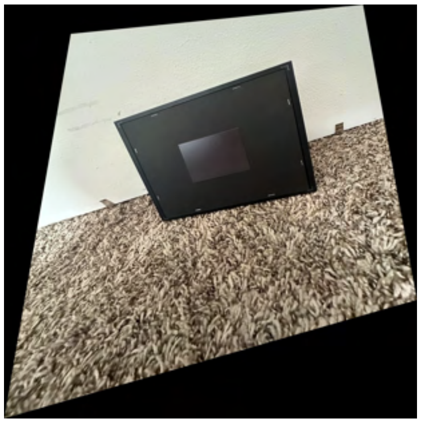

(c) My Image 1 without Any Distortion

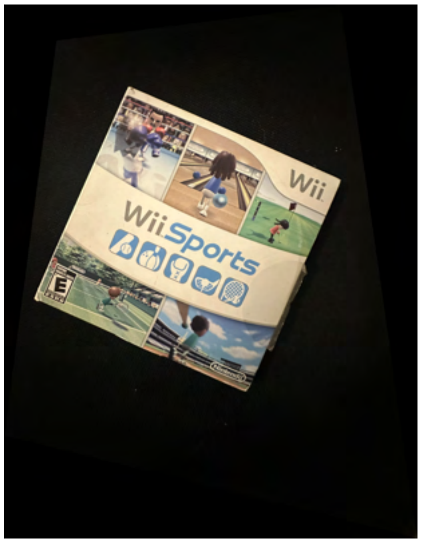

(d) My Image 2 without Any Distortion

Task 2

One Step Approach

One Step Approach Results


(a) My Image 1 One Step Approach



(b) My Image 1 One Step Approach

Oberservations

Observations

First let's talk about the point-to-point method. This way is the most simple to implement because we only need to use four points to achieve a result. There was also never a time in the point to point where I needed to do matrix scaling for RAM limitations. The results of the point-to-point are very good and angles seem to be 90 degrees at all points.

The next method was the two-step approach. This method requires you to find 2 pairs of parallel lines and then 2 pairs of transposed orthogonal lines from the first step to execute. However, I found that there were lots of scaling issues when applying the homographies because the scaling would result in slightly distorted images. But, then you do the second step which builds on the first homography, and then the slight inconsistencies compound which can give bad results. This can be alleviated by finding the smallest scaling factor which is under the RAM threshold, but this can be tedious.

The final method was the one-step approach. This one was similar in complexity to the two-step approach because you needed to find lines, but instead of the interdependices between the lines, all you need is five pairs of orthogonal lines. This method also had some scaling issues but was much more manageable than the two-step method. I did initially have lots of trouble with the one-step method because I was using the four corner points and another same rotation orthogonal line pairs. This resulted in very poor results. However, for the fifth pair, I found an orthogonal pair that had a different rotation than the 4 corner line pairs which made my results great. This could be difficult to do if your shape is not a square, but is well worth the effort to get the best results. As shown in the board image. In my opinion, I found that the point-to-point method was the best in terms of results, efficiency of implementation, and execution time.

Source Code

```
import cv2
import numpy as np

#this function creates Homography Matrix for translation
def create_homography(x , x_prime):

    #formatting list of list into 1 list
    x = [item for sublist in x for item in sublist]
    x = np.array(x)
    x_prime = [item for sublist in x_prime for item in sublist]
    x_prime = np.array(x_prime)

    # b = A^-1 * c
    # create matricies
    c = x_prime.reshape((8,1))
    A = np.zeros((8,8))

    for i in range(0, 4):
        A[2*i] = np.array([x[2*i], x[2*i+1], 1, 0, 0, 0, -x[2*i]*x_prime[2*i],
                          -x[2*i+1]*x_prime[2*i]])
        A[2*i + 1] = np.array([0, 0, 0, x[2*i], x[2*i+1], 1, -x[2*i]*x_prime[2*i+1],
                          -x[2*i+1]*x_prime[2*i+1]])

    #get inverse of A
    A_inv = np.linalg.inv(A)

    #compute b which is the homography values
    b = np.dot(A_inv, c)

    #construct homography 3x3 matrix
    homography = np.zeros((3, 3))
    homography.flat[:b.size] = b
    homography[2][2] = 1

    return(homography)

def apply_homography_2(img, homography):

    #calculate new image boundaries
    height, width = img.shape[:2]

    # Define the corner points of the input image
    corners = np.array([
        [0, 0],
        [width, 0],
        [width, height],
        [0, height]
    ], dtype=np.float32)

    corners_homogeneous = np.hstack([corners, np.ones((4, 1))])
    new_corners = np.matmul(homography, corners_homogeneous.T).T
    new_corners /= new_corners[:, 2:3]

    # new_corners = np.matmul(homography, corners)
```

Source Code

```
    # new_corners /= new_corners[2, :]

    min_x = np.floor(np.min(new_corners[:, 0])).astype(int)
    max_x = np.ceil(np.max(new_corners[:, 0])).astype(int)
    min_y = np.floor(np.min(new_corners[:, 1])).astype(int)
    max_y = np.ceil(np.max(new_corners[:, 1])).astype(int)

    #now create new image
    dest_width = max_x - min_x
    dest_height = max_y - min_y

    # print(dest_width)
    # print(dest_height)

    # Create a zero matrix for the destination image with the new size
    dest_img = np.full((dest_height, dest_width, img.shape[2]), 0, dtype=img.dtype)

    # Get inverse homography
    inverse_homography = np.linalg.inv(homography)

    # Iterate through the destination image and replace pixels if in frame with transformation
    for y in range(dest_height):
        for x in range(dest_width):
            # Offset the coordinates by the min_x and min_y values (to handle negative offsets)
            x_prime = x + min_x
            y_prime = y + min_y

            # Perform the inverse transformation to get the corresponding source coordinates
            three_d_coords = np.matmul(inverse_homography, np.array([x_prime, y_prime, 1]))
            x_real, y_real, _ = three_d_coords / three_d_coords[2]

            # Check if the source coordinates are within the bounds of the source image
            if 0 <= x_real < width and 0 <= y_real < height:
                x_real, y_real = int(x_real), int(y_real)

                # Assign the pixel value from the source image to the destination image
                dest_img[y, x] = img[y_real, x_real]

    return dest_img



def apply_homography(img, homography, output_size=None):

    #calculate new image boundaries
    height, width = img.shape[:2]

    # Define the corner points of the input image
    corners = np.array([
        [0, 0],
        [width, 0],
        [width, height],
        [0, height]
    ], dtype=np.float32)
```

Source Code

```
    corners_homogeneous = np.hstack([corners, np.ones((4, 1))])
    new_corners = np.matmul(homography, corners_homogeneous.T).T
    new_corners /= new_corners[:, 2:3]

    min_x = np.floor(np.min(new_corners[:, 0])).astype(int)
    max_x = np.ceil(np.max(new_corners[:, 0])).astype(int)
    min_y = np.floor(np.min(new_corners[:, 1])).astype(int)
    max_y = np.ceil(np.max(new_corners[:, 1])).astype(int)

    #now create new image
    new_width = max_x - min_x
    new_height = max_y - min_y

    # print(new_width)
    # print(new_height)
    new_img = np.zeros((new_height, new_width, img.shape[2]), dtype=img.dtype)
    mask = np.zeros((new_height, new_width), dtype=bool)

    #create meshgrid
    x = np.arange(new_width)
    y = np.arange(new_height)
    X, Y = np.meshgrid(x, y)
    X_flat = X.flatten()
    Y_flat = Y.flatten()

    #convert to homogeneous form
    homogeneous_coords = np.vstack([X_flat + min_x , Y_flat + min_y, np.ones(X.size)])

    #apply the homography
    inverse_homography = np.linalg.inv(homography)
    original_coords = np.matmul(inverse_homography, homogeneous_coords).T
    original_coords /= original_coords[:, 2:3]

    #turn back into 2-d
    X_orig = np.clip(original_coords[:, 0].astype(int), 0, width - 1)
    Y_orig = np.clip(original_coords[:, 1].astype(int), 0, height - 1)

    # Update the mask where the coordinates are valid
    valid_mask = (original_coords[:, 0] >= 0) & (original_coords[:, 0] < width) & \
                 (original_coords[:, 1] >= 0) & (original_coords[:, 1] < height)
    mask[Y_flat, X_flat] = valid_mask

    # Assign pixel values where mask is True
    new_img[mask] = img[Y_orig[valid_mask], X_orig[valid_mask]]

    return new_img

def point_to_point(pqrs_src, pqrs_dest, img):

    point_to_point_homography = create_homography(pqrs_src, pqrs_dest)
    new_img = apply_homography(img, point_to_point_homography)

    return hi

def angle_between_lines(H, l1, l2):
```

Source Code

```
    # Convert lines to numpy arrays
    l1 = np.array(l1)
    l2 = np.array(l2)

    # Compute transformed lines
    l1_prime = np.dot(np.linalg.inv(H).T, l1)
    l2_prime = np.dot(np.linalg.inv(H).T, l2)

    #compute direction vectors from line coefficients
    dir1 = l1_prime[:2]  # (a, b) from ax + by + c = 0
    dir2 = l2_prime[:2]  # (a, b) from ax + by + c = 0

    dot_product = np.dot(dir1, dir2)
    mag1 = np.linalg.norm(dir1)
    mag2 = np.linalg.norm(dir2)

    # Compute cosine of the angle
    cos_theta = dot_product / (mag1 * mag2)

    cos_theta = np.clip(cos_theta, -1.0, 1.0)

    theta_rad = np.arccos(cos_theta)
    theta_deg = np.degrees(theta_rad)

    return theta_rad, theta_deg

def plot_lines_on_image(img, lines):
    img_with_lines = img.copy()
    h, w = img.shape[:2]

    for line in lines:
        # Homogeneous line equation is: ax + by + c = 0
        a, b, c = line

        # Find two points on the line
        if b != 0:
            x1, y1 = 0, -c / b
            x2, y2 = w, -(a * w + c) / b
        # If b == 0, it's a vertical line (x = -c / a)
        else:
            x1, y1 = -c / a, 0
            x2, y2 = -c / a, h

        # Draw the line on the image
        img_with_lines = cv2.line(img_with_lines, (int(x1), int(y1)),
                                  (int(x2), int(y2)), (0, 255, 0), 2)

    return img_with_lines

def two_step_approach(pqrs, img, name):
#this function uses the two-step approach to get the nondistorted image

    #Step 1: Removing the Projective Distortion
    lines = get_4_lines(pqrs)
    projective_homography = get_vanishing_line_homography(lines)
```

Source Code

```
#scale homography down if needed
if(name == 'task_2_2step_my_img1_affine.jpg'):
    projective_homography = np.matmul(projective_homography,np.array([
            [0.8, 0, 0],
            [0, 0.8, 0],
            [0, 0, 1]
        ]))
affine_img = apply_homography(img, projective_homography)


#Step 2: Removign Affine Distortion

#get new lines (l_prime and m_prime in equations)
new_line = homography_on_line(lines, projective_homography)

L = np.zeros((2,2))
L[0][0] = new_line[0][0] * new_line[2][0]
L[0][1] = new_line[0][0] * new_line[2][1] + new_line[0][1] * new_line[2][0]
L[1][0] = new_line[1][0] * new_line[3][0]
L[1][1] = new_line[1][0] * new_line[3][1] + new_line[1][1] * new_line[3][0]

c = np.zeros((2,1))
c[0] = -1* new_line[0][1] * new_line[2][1]
c[1] = -1* new_line[1][1] * new_line[3][1]

# solve for s = L^-1 * c
L_inv = np.linalg.pinv(L)
s = np.matmul(L_inv, c)

#create the S matrix
S = np.ones((2,2))
S[0][0] = s[0]
S[0][1] = s[1]
S[1][0] = s[1]

#Now use SVD to find matrix A
V, D, V_T  = np.linalg.svd(S)
D_of_A = np.sqrt(np.diag(D))
A = np.dot(V, D_of_A)
A = np.dot(A, V_T)

#now create the Homography for affine
H_affine = np.zeros((3,3))
H_affine[0][0] = A[0][0]
H_affine[0][1] = A[0][1]
H_affine[1][0] = A[1][0]
H_affine[1][1] = A[1][1]
H_affine[2][2] = 1

#calculate new homography for entire image
combined_homography = np.matmul(np.linalg.inv(H_affine), projective_homography)

#confim that angle between lines is 90 degrees
print(angle_between_lines(combined_homography, lines[0], lines[3]))
```

Source Code

```
    #if vector is too large scale it down
    if(name == 'task_2_2step_my_img1_affine.jpg' or name == 'task_2_2step_my_img2_affine.jpg'):
        combined_homography = np.matmul(combined_homography,np.array([
                [0.3, 0, 0],
                [0, 0.3, 0],
                [0, 0, 1]
            ]))

    #apply the homography
    new_img = apply_homography(img, combined_homography)

    return new_img


def homography_on_line(lines, homography):
# this turns in the lines to lines without porjective distortion

    homography_inv = np.linalg.inv(homography)
    homography_inv_transpose = homography_inv.T

    new_lines = []
    for i in lines:
        new_line = np.dot(homography_inv_transpose, i)
        new_lines.append(new_line)

    return(new_lines)


#this function find the vanishing line homography when given the pqrs points
def get_vanishing_line_homography(line):

    #find the 2 vanishing points between the lines
    vanishing_point0 = np.cross(line[0], line[1])
    vanishing_point1 = np.cross(line[2], line[3])

    #get vanishing line from vanishing points
    vanishing_line = np.cross(vanishing_point0, vanishing_point1)

    #turn vanishing line to Homogarphy Matrix for projection removal
    projection_homography = np.identity(3)
    projection_homography[2] = vanishing_line
    projection_homography[2] = projection_homography[2] / projection_homography[2][2]

    return projection_homography


#get the 4 lines created by the four points
def get_4_lines(points):

    line_pq = get_line(points[0][0], points[0][1], points[1][0], points[1][1])
    line_rs = get_line(points[3][0], points[3][1], points[2][0], points[2][1])
    line_ps = get_line(points[3][0], points[3][1], points[0][0], points[0][1])
    line_qr = get_line(points[2][0], points[2][1], points[1][0], points[1][1])
```

Source Code

```
    return(np.array([line_pq ,line_rs ,line_ps ,line_qr]))


# this function takes 2-D points and returns a line in 3-d
def get_line(point0, point1, point2, point3):

    homogenous_point0 = np.array([point0, point1, 1])
    homogenous_point1 = np.array([point2, point3, 1])

    line = np.cross(homogenous_point0, homogenous_point1)
    if np.abs(line[2]) >= 1e-6:
        line = line / line[2]

    return(line)

#this function does the one step approach
def one_step_approach(pqrs, fifth_line_points, img, scale = False):

    #get the 5 orthognal lines from original image
    line = get_4_lines(pqrs)
    fifth_line = np.array(get_line(fifth_line_points[0][0], fifth_line_points[0][1],
                                   fifth_line_points[1][0], fifth_line_points[1][1]))
    sixth_line = np.array(get_line(fifth_line_points[2][0], fifth_line_points[2][1],
                                   fifth_line_points[3][0], fifth_line_points[3][1]))

    line = np.vstack((line, fifth_line, sixth_line))

    hi = plot_lines_on_image(img, line)
    cv2.imshow('Image Window', hi)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
    # print("lines")
    # print(line)

    A_ = np.zeros((5,5))
    A_[0] = [line[0][0]*line[3][0], line[0][0]*line[3][1] + line[0][1]*line[3][0],
             line[0][1]*line[3][1], line[0][0]+line[3][0], line[0][1]+line[3][1]]
    A_[1] = [line[0][0]*line[2][0], line[0][0]*line[2][1] + line[0][1]*line[2][0],
             line[0][1]*line[2][1], line[0][0]+line[2][0], line[0][1]+line[2][1]]
    A_[2] = [line[1][0]*line[3][0], line[1][0]*line[3][1] + line[1][1]*line[3][0],
             line[1][1]*line[3][1], line[1][0]+line[3][0], line[1][1]+line[3][1]]
    A_[3] = [line[1][0]*line[2][0], line[1][0]*line[2][1] + line[1][1]*line[2][0],
             line[1][1]*line[2][1], line[1][0]+line[2][0], line[1][1]+line[2][1]]
    A_[4] = [line[5][0]*line[4][0], line[5][0]*line[4][1] + line[5][1]*line[4][0],
             line[5][1]*line[4][1], line[5][0]+line[4][0], line[5][1]+line[4][1]]

    c = np.zeros((5,1))
    c[0] = -1
    c[1] = -1
    c[2] = -1
    c[3] = -1
    c[4] = -1

    #solve for conic variables using b = A^-1 * c
    A_inv = np.linalg.pinv(A_)
```

Source Code

```
    b = np.dot(A_inv, c)
    b = b/np.max(b)

    #create the AA_T matrix
    AA_T = np.zeros((2,2))
    AA_T[0][0] = b[0]
    AA_T[0][1] = b[1]
    AA_T[1][0] = b[1]
    AA_T[1][1] = b[2]

    #do eigendecompostion to get value of A
    V, D, V_transpose = np.linalg.svd(AA_T)
    D_of_A = np.sqrt(np.diag(D))
    A = np.dot(np.dot(V,D_of_A), V.transpose())

    #using A find v by v = A^-1 * g
    g = np.zeros((2,1))
    g[0] = b[3]
    g[1] = b[4]
    v = np.dot(np.linalg.pinv(A), g)

    #now create the Homography
    homography = np.zeros((3,3))
    homography[0][0] = A[0][0]
    homography[0][1] = A[0][1]
    homography[1][0] = A[1][0]
    homography[1][1] = A[1][1]
    homography[2][0] = v[0][0]
    homography[2][1] = v[1][0]
    homography[2][2] = 1

    #confim that angle between lines is 90 degrees
    print(angle_between_lines(np.linalg.inv(homography), line[0], line[3]))

    #scale the homography
    homography = np.linalg.inv(homography)
    if(scale is True):
        homography = np.matmul(homography,np.array([
                [0.4, 0, 0],
                [0, 0.4, 0],
                [0, 0, 1]
            ]))

    #apply the homography
    new_img = apply_homography(img, homography)

    return new_img

def main():

    #First get the points from the provided images
    board_img = cv2.imread('board_1.jpeg')
    corridor_img = cv2.imread('corridor.jpeg')
    my_img1 = cv2.imread('my_img1.jpg')
    my_img2 = cv2.imread('my_img2.jpg')
```

Source Code

```
# cv2.imshow('Image Window', corridor_img)
# cv2.waitKey(0)
# cv2.destroyAllWindows()

#List the points for each Picture
#Board Points
p_board = [73,425]
q_board = [1221,146]
r_board = [1354,1956]
s_board = [423,1798]
pqrs_board = np.array([p_board,q_board,r_board,s_board])

#Point to Point Board Points
p_new_board = [0,0]
q_new_board = [1000,0]
r_new_board = [1000,1500]
s_new_board = [0,1500]
pqrs_new_board = np.array([p_new_board,q_new_board,r_new_board,s_new_board])

#Corridor Points
p_corridor = [813,576]
q_corridor = [1073,531]
r_corridor = [1068,1212]
s_corridor = [813,1069]
pqrs_corridor = np.array([p_corridor,q_corridor,r_corridor,s_corridor])

#Point to Point Corridor Points
p_new_corridor = [813,550]
q_new_corridor = [1070,550]
r_new_corridor = [1070,1100]
s_new_corridor = [813,1100]
pqrs_new_corridor = np.array([p_new_corridor,q_new_corridor,r_new_corridor,s_new_corridor])

#My Img 1 Points
p_my_img1 = [481,1330]
q_my_img1 = [2141,805]
r_my_img1 = [2266,2627]
s_my_img1 = [950,2612]
pqrs_my_img1 = np.array([p_my_img1,q_my_img1,r_my_img1,s_my_img1])

#Point to Point My Img 1 Points
p_new_my_img1 = [0,0]
q_new_my_img1 = [300,0]
r_new_my_img1 = [300,600]
s_new_my_img1 = [0,600]
pqrs_new_my_img1 = np.array([p_new_my_img1,q_new_my_img1,r_new_my_img1,s_new_my_img1])

#My Img 2 Points
p_my_img2 = [954,704]
q_my_img2 = [2841,1729]
r_my_img2 = [2038,3380]
s_my_img2 = [141,2889]
pqrs_my_img2 = np.array([p_my_img2,q_my_img2,r_my_img2,s_my_img2])
```

Source Code

```
#Point to Point My Img 2 Points
p_new_my_img2 = [0,0]
q_new_my_img2 = [300,0]
r_new_my_img2 = [300,600]
s_new_my_img2 = [0,600]
pqrs_new_my_img2 = np.array([p_new_my_img2,q_new_my_img2,r_new_my_img2,s_new_my_img2])

########### Task 1 - Point to Point Correspondance ####################

# Get Point to Point for Board image
# img1 = point_to_point(pqrs_board, pqrs_new_board, board_img)
# cv2.imwrite('task_1_ptp_board.jpg', img1)

# # Get Point to Point for Corridor image
# img2 = point_to_point(pqrs_corridor, pqrs_new_corridor, corridor_img)
# cv2.imwrite('task_1_ptp_corridor.jpg', img2)

########### Task 1 - Two Step ##########################################

# Apply two-step approach to the board image
# img3 = two_step_approach(pqrs_board, board_img, 'task_1_2step_board_affine.jpg')
# cv2.imwrite('task_1_2step_board_undist.jpg', img3)

# Apply two-step approach to the corridor image
# img4 = two_step_approach(pqrs_corridor, corridor_img, 'task_1_2step_corridor_affine.jpg')
# cv2.imwrite('task_1_2step_corridior_undist.jpg', img4)

########### Task 1 - One Step ##########################################

#get orthongonal points on board image
fifth_line_point_board_1 = [824,869]
fifth_line_point_board_2 = [484,1320]
fifth_line_point_board_3 = [480, 895]
fifth_line_point_board_4 = [879, 1241]
fifth_line_points_board = np.array([fifth_line_point_board_1, fifth_line_point_board_2,
                                    fifth_line_point_board_3, fifth_line_point_board_4])

# Apply one-step approach to the board image
# img5 = one_step_approach(pqrs_board, fifth_line_points_board, board_img)
# cv2.imwrite('task_1_1step_board.jpg', img5)

#get orthongonal points on corrdior image
fifth_line_point_corridor_1 = [815,576]
fifth_line_point_corridor_2 = [1068,1212]
fifth_line_point_corridor_3 = [1073,531]
fifth_line_point_corridor_4 = [811,1069]
fifth_line_points_corridor = np.array([fifth_line_point_corridor_1, fifth_line_point_corridor_2,
                                       fifth_line_point_corridor_3, fifth_line_point_corridor_4])

# Apply one-step approach to the corridor image
# img6 = one_step_approach(pqrs_corridor, fifth_line_points_corridor, corridor_img)
# cv2.imwrite('task_1_1step_corridior.jpg', img6)

########### Task 2 - Point to Point Correspondance ####################
```

Source Code

```
    # # Get Point to Point for My img 1
    # img7 = point_to_point(pqrs_my_img1, pqrs_new_my_img1, my_img1)
    # cv2.imwrite('task_2_ptp_my_img1.jpg', img7)

    # # # Get Point to Point for My img 2
    # img8 = point_to_point(pqrs_my_img2, pqrs_new_my_img2, my_img2)
    # cv2.imwrite('task_2_ptp_my_img2.jpg', img8)


    ########### Task 2 - Two Step ##########################################

    # Apply two-step approach to my img 1
    # img9 = two_step_approach(pqrs_my_img1, my_img1, 'task_2_2step_my_img1_affine.jpg')
    # cv2.imwrite('task_2_2step_my_img1_undist.jpg', img9)

    # Apply two-step approach to my img 2
    # img10 = two_step_approach(pqrs_my_img2, my_img2, 'task_2_2step_my_img2_affine.jpg')
    # cv2.imwrite('task_2_2step_my_img2_undist.jpg', img10)


    ########### Task 2 - One Step ##########################################

    #get orthongonal points on my image 1
    fifth_line_point_my_img1_1 = [481,1330]
    fifth_line_point_my_img1_2 = [2266,2627]
    fifth_line_point_my_img1_3 = [2141, 805]
    fifth_line_point_my_img1_4 = [950, 2612]
    fifth_line_points_my_img1 = np.array([fifth_line_point_my_img1_1, fifth_line_point_my_img1_2,
                                          fifth_line_point_my_img1_3, fifth_line_point_my_img1_4])

    # Apply one-step approach to my image 1
    # img11 = one_step_approach(pqrs_my_img1, fifth_line_points_my_img1, my_img1, scale = True)
    # cv2.imwrite('task_2_1step_my_img1.jpg', img11)

    #get orthongonal points on my image 1
    fifth_line_point_my_img2_1 = [954,704]
    fifth_line_point_my_img2_2 = [2038,3380]
    fifth_line_point_my_img2_3 = [2841,1729]
    fifth_line_point_my_img2_4 = [141,2889]
    fifth_line_points_my_img2 = np.array([fifth_line_point_my_img2_1, fifth_line_point_my_img2_2,
                                          fifth_line_point_my_img2_3, fifth_line_point_my_img2_4])

    # Apply one-step approach to my image 1
    # img12 = one_step_approach(pqrs_my_img2, fifth_line_points_my_img2, my_img2)
    # cv2.imwrite('task_2_1step_my_img2.jpg', img12)


    return



if __name__=="__main__":
    main()
```