Homework 3

Task 1

Task 1.1: Point to point correspondence

From the JSON files, we have the coordinates of points P, Q, R, S. By definition, in homogeneous coordinates: $\mathbf{x}' = \mathbf{H}\mathbf{x}$.

$$\begin{pmatrix} x_1' \\ x_2' \\ x_3' \end{pmatrix} = \mathbf{H} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

We also know that $\mathbf{H}$ has the following form:

$$\mathbf{H} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix}$$

But due to the definition of homography, the very last value h_{33} can be 1, resulting in:

$$\mathbf{H} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{pmatrix}$$

$$\mathbf{x}' = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

With this we now have a system of equations:

$$\begin{cases} x_1' = h_{11}x_1 + h_{12}x_2 + h_{13}x_3 \\ x_2' = h_{21}x_1 + h_{22}x_2 + h_{23}x_3 \\ x_3' = h_{31}x_1 + h_{32}x_2 + x_3 \end{cases}$$

With our vector $\mathbf{x}$ in homogeneous coordinates, the equivalent in the physical space, (x, y), is given by $x = \frac{x_1}{x_3}$ and $y = \frac{x_2}{x_3}$; the same happens with (x', y'). Replacing these in our equation system (divide both by x_3'):

$$\begin{cases} x' = \dfrac{h_{11}x_1 + h_{12}x_2 + h_{13}x_3}{h_{31}x_1 + h_{32}x_2 + x_3} \\ y' = \dfrac{h_{21}x_1 + h_{22}x_2 + h_{23}x_3}{h_{31}x_1 + h_{32}x_2 + x_3} \end{cases}$$

Then we divide by x_3 and rearrange them:

$$\begin{cases} x' = \dfrac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + 1} \\ y' = \dfrac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + 1} \end{cases}$$

$$\begin{cases} 0 = h_{11}x + h_{12}y + h_{13} - h_{31}xx' + h_{32}yx' - x' \\ 0 = h_{21}x + h_{22}y + h_{23} - h_{31}xy' + h_{32}yy' + y' \end{cases}$$

Since we have 4 points (P, Q, R, S), using the equations above we have 8 equations, which is the same as the number of unknown variables. The can plug the 4 points into the system resulting in

$$\begin{cases} 0 = h_{11}x_P + h_{12}y_P + h_{13} - h_{31}x_Px'_P - h_{32}y_Px'_P - x'_P \\ 0 = h_{21}x_P + h_{22}y_P + h_{23} - h_{31}x_Py'_P - h_{32}y_Py'_P - y'_P \\ 0 = h_{11}x_Q + h_{12}y_Q + h_{13} - h_{31}x_Qx'_Q - h_{32}y_Qx'_Q - x'_Q \\ 0 = h_{21}x_Q + h_{22}y_Q + h_{23} - h_{31}x_Qy'_Q - h_{32}y_Qy'_Q - y'_Q \\ 0 = h_{11}x_R + h_{12}y_R + h_{13} - h_{31}x_Rx'_R - h_{32}y_Rx'_R - x'_R \\ 0 = h_{21}x_R + h_{22}y_R + h_{23} - h_{31}x_Ry'_R - h_{32}y_Ry'_R - y'_R \\ 0 = h_{11}x_S + h_{12}y_S + h_{13} - h_{31}x_Sx'_S - h_{32}y_Sx'_S - x'_S \\ 0 = h_{21}x_S + h_{22}y_S + h_{23} - h_{31}x_Sy'_S - h_{32}y_Sy'_S - y'_S \end{cases}$$

We can also express this system as the matrix operation

$$\begin{pmatrix} x'_P \\ y'_P \\ x'_Q \\ y'_Q \\ x'_R \\ y'_R \\ x'_S \\ y'_S \end{pmatrix} = \begin{pmatrix} x_P & y_P & 1 & 0 & 0 & 0 & -x_Px'_P & -y_Px'_P \\ 0 & 0 & 0 & x_P & y_P & 1 & -x_Py'_P & -y_Py'_P \\ x_Q & y_Q & 1 & 0 & 0 & 0 & -x_Qx'_Q & -y_Qx'_Q \\ 0 & 0 & 0 & x_Q & y_Q & 1 & -x_Qy'_Q & -y_Qy'_Q \\ x_R & y_R & 1 & 0 & 0 & 0 & -x_Rx'_R & -y_Rx'_R \\ 0 & 0 & 0 & x_R & y_R & 1 & -x_Ry'_R & -y_Ry'_R \\ x_S & y_S & 1 & 0 & 0 & 0 & -x_Sx'_S & -y_Sx'_S \\ 0 & 0 & 0 & x_S & y_S & 1 & -x_Sy'_S & -y_Sy'_S \end{pmatrix} \begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{pmatrix}$$

And with this we can rearrange the equation into:

$$\begin{pmatrix} x_P & y_P & 1 & 0 & 0 & 0 & -x_Px'_P & -y_Px'_P \\ 0 & 0 & 0 & x_P & y_P & 1 & -x_Py'_P & -y_Py'_P \\ x_Q & y_Q & 1 & 0 & 0 & 0 & -x_Qx'_Q & -y_Qx'_Q \\ 0 & 0 & 0 & x_Q & y_Q & 1 & -x_Qy'_Q & -y_Qy'_Q \\ x_R & y_R & 1 & 0 & 0 & 0 & -x_Rx'_R & -y_Rx'_R \\ 0 & 0 & 0 & x_R & y_R & 1 & -x_Ry'_R & -y_Ry'_R \\ x_S & y_S & 1 & 0 & 0 & 0 & -x_Sx'_S & -y_Sx'_S \\ 0 & 0 & 0 & x_S & y_S & 1 & -x_Sy'_S & -y_Sy'_S \end{pmatrix}^{-1} \begin{pmatrix} x'_P \\ y'_P \\ x'_Q \\ y'_Q \\ x'_R \\ y'_R \\ x'_S \\ y'_S \end{pmatrix} = \begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{pmatrix}$$

Which we can solve easily using numpy, giving us 8 values of the homography **H** matrix and the last one is 1.
As for this particular case, we have that the undistorted image undergoes a transformation through a homography matrix. We define this transformation $\mathbf{x}_{distorted} = \mathbf{H}\mathbf{x}_{undistorted}$. From this we can get the homography matrix that maps $\mathbf{x}_{undistorted}$ to $\mathbf{x}_{distorted}$.

Since we have the distorted image and we want to get the undistorted image, we get the dot product between $\mathbf{x}_{distorted}$ and $\mathbf{H}^{-1}$, and then map the RGB values from the distorted image to the undistorted one, with the undistorted coordinates.

Task 1.2: Two-step method

Removing projective distortion: First, we need to calculate the vanishing line. To do this, we take 2 pairs of what we know are parallel lines in the scene (might not look parallel in the image). In this case we take the lines created by pairs PQ and RS, and PS QR. We calculate the lines created by these pairs:

$$\mathbf{l}_1 = \mathbf{p} \times \mathbf{q}$$

$$\mathbf{l}_2 = \mathbf{r} \times \mathbf{s}$$

$$\mathbf{l}_3 = \mathbf{p} \times \mathbf{s}$$

$$\mathbf{l}_4 = \mathbf{q} \times \mathbf{r}$$

Now with this 2 pair of lines we can get the vanishing line. First we find the intersection point between $\mathbf{l}_1$ and $\mathbf{l}_2$, and the intersection between $\mathbf{l}_3$ and $\mathbf{l}_4$. After this, we use those 2 points to get the vanishing line.

$$\mathbf{x}_{vanishing_1} = \mathbf{l}_1 \times \mathbf{l}_2$$

$$\mathbf{x}_{vanishing_2} = \mathbf{l}_3 \times \mathbf{l}_4$$

And now the vanishing line:

$$\mathbf{l}_{vanishing} = \mathbf{x}_{vanishing_1} \times \mathbf{x}_{vanishing_2}$$

We need to get rid of the purely projective distortion, in other words, the distortion that maps $\mathbf{l}_\infty$ to another line, called the vanishing line. To achieve this, we need to find a homography $\mathbf{H}$ that maps the vanishing line, $\mathbf{l}_{vanishing}$ back to $\mathbf{l}_\infty$ according to the following formula:

$$\mathbf{l}_\infty = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \mathbf{H}^{-\top}\mathbf{l}_{vanishing}$$

For this homography matrix to transform a line into the line at infinity, it must have the following form:

$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l_1 & l_2 & l_3 \end{pmatrix}$$

Where l_1, l_2 and l_3 are the elements of $\mathbf{l}_v anishing$.

Removing affine distortion: The equation for the angle θ that forms between 2 (perpendicular in the scene) lines is given by:

$$\cos\theta = \frac{\mathbf{l}^\top \mathbf{C}^*_\infty \mathbf{m}}{\sqrt{(\mathbf{l}^\top \mathbf{C}^*_\infty \mathbf{l})(\mathbf{m}^\top \mathbf{C}^*_\infty \mathbf{m})}}$$

Where $\mathbf{l}$ and $\mathbf{m}$ are lines that define a degenerate conic and $\mathbf{C}^*_\infty$ is the dual generate conic:

$$\mathbf{C}^*_\infty = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Since conics transform as $\mathbf{C}' = \mathbf{H}^{-\top}\mathbf{C}^*\mathbf{H}^\top$, we can substitute this in the formula for $\cos\theta$:

$$\cos\theta|_{numerator} = \left(\mathbf{l}'^\top\mathbf{H}\right)\left(\mathbf{H}^{-1}\mathbf{C}'^*_\infty\mathbf{H}^{-\top}\right)\left(\mathbf{H}^\top\mathbf{m}'\right) = \mathbf{l}'^\top\mathbf{C}'^*_\infty\mathbf{m}'$$

As we mentioned that the lines would be perpendicular in the planar scene, the angle θ between them is 0. Replacing this result above yields the following

$$\mathbf{l}'^*\mathbf{H}\mathbf{C}^*_\infty\mathbf{H}^\top\mathbf{m}' = 0$$

And replacing the elements of these matrices:

$$\begin{pmatrix} l'_1 & l'_2 & l'_3 \end{pmatrix}\begin{pmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{0}^\top & 1 \end{pmatrix}\begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0}^\top & 0 \end{pmatrix}\begin{pmatrix} \mathbf{A}^\top & \mathbf{0} \\ \mathbf{t}^\top & 1 \end{pmatrix}\begin{pmatrix} m'_1 \\ m'_2 \\ m'_3 \end{pmatrix} = 0$$

$$\begin{pmatrix} l'_1 & l'_2 & l'_3 \end{pmatrix}\begin{pmatrix} \mathbf{A}\mathbf{A}^\top & \mathbf{0} \\ \mathbf{0}^\top & 0 \end{pmatrix}\begin{pmatrix} m'_1 \\ m'_2 \\ m'_3 \end{pmatrix} = 0$$

We define $\mathbf{S} = \mathbf{A}\mathbf{A}^\top$, resulting in

$$\begin{pmatrix} l'_1 & l'_2 \end{pmatrix}\begin{pmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{pmatrix}\begin{pmatrix} m'_1 \\ m'_2 \end{pmatrix} = 0$$

And since $\mathbf{S}$ is symmetric, because $\mathbf{A}\mathbf{A}^\top$ is symmetric, there's only 3 variables that we need to find, as $s_{12} = s_{21}$.

$$s_{11}l'_1m'_1 + s_{12}(l'_1m'_2 + l'_2m'_1) + s_{22}l'_2m'_2 = 0$$

Since only the ratios are important, we can set $s_{22} = 1$, leaving us with only 2 unknowns. Our new equations is as follows.

$$s_{11}l'_1m'_1 + s_{12}(l'_1m'_2 + l'_2m'_1) + l'_2m'_2 = 0$$

Now, we have 2 pairs of lines, replacing them in this gives us a system of equations we can solve:

$$s_{11}l'_{1_1}l'_{2_1} + s_{12}(l'_{1_1}l'_{2_2} + l'_{1_2}l'_{2_1}) + l'_{1_2}l'_{2_2} = 0$$
$$s_{11}l'_{3_1}l'_{4_1} + s_{12}(l'_{3_1}l'_{4_2} + l'_{3_2}l'_{4_1}) + l'_{3_2}l'_{4_2} = 0$$

With a matrix representation of:

$$\begin{pmatrix} l'_{1_1}l'_{2_1} & (l'_{1_1}l'_{2_2}+l'_{1_2}l'_{2_1}) \\ l'_{3_1}l'_{4_1} & (l'_{3_1}l'_{4_2}+l'_{3_2}l'_{4_1}) \end{pmatrix}\begin{pmatrix} s_{11} \\ s_{12} \end{pmatrix} = \begin{pmatrix} -l'_{1_2}l'_{2_2} \\ -l'_{3_2}l'_{4_2} \end{pmatrix}$$

Having the values of $\mathbf{S}$ helps estimate $\mathbf{A}$. Decomposing $\mathbf{A}$ using SVD results in:

$$\mathbf{A} = \mathbf{V}\mathbf{D}\mathbf{V}^\top$$

$$\mathbf{A}\mathbf{A}^\top = \mathbf{V}\mathbf{D}\mathbf{V}^\top\mathbf{V}\mathbf{D}\mathbf{V}^\top = \mathbf{V}\mathbf{D}^2\mathbf{V}^\top = \mathbf{V}\begin{pmatrix} \lambda_1^2 & 0 \\ 0 & \lambda_2^2 \end{pmatrix}\mathbf{V}^\top$$

From this we get both the eigenvalues and eigenvectors of $\mathbf{A}$, which we use to calculate $\mathbf{A}$. And after finding this, we can obtain the homography matrix that removes affine distortion:

$$\mathbf{H}_{affine} = \begin{pmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0}^\top & 1 \end{pmatrix}$$

The procedure we follow is find both $\mathbf{H}_{projective}$ and $\mathbf{H}_{affine}$, and use them both to map each position in the distorted image to one in the undistorted one:

$$\mathbf{x}_{undistorted} = \mathbf{H}_{affine}^{-1}\mathbf{H}_{projective}^{-1}\mathbf{x}_{distorted}$$

Task 1.3: One-step method

Considering again our equation with the angle θ between lines that are orthogonal in the planar scene.

$$\cos\theta = \mathbf{l}'^\top\mathbf{C}'^*_\infty\mathbf{m} = 0$$

Where $\mathbf{C}'^*_\infty = \mathbf{H}\mathbf{C}'^*_\infty\mathbf{H}^\top$. Now, if we let

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

We can fix $f = 1$ since we are interested in the ratios. By knowing 5 pairs of orthogonal lines (in world coordinates) we can construct a system of equations to solve.

$$\begin{pmatrix} l'_1 & l'_2 & l'_3 \end{pmatrix}\begin{pmatrix} a & \frac{b}{2} & \frac{d}{2} \\ \frac{b}{2} & c & \frac{e}{2} \\ \frac{d}{2} & \frac{e}{2} & 1 \end{pmatrix}\begin{pmatrix} m'_1 \\ m'_2 \\ m'_3 \end{pmatrix} = 0$$

$$l'_1m'_1a + (l'_2m'_1 + l'_1m'_2)\frac{b}{2} + l'_2m'_2c + (l'_3m'_1 + l'_1m'_3)\frac{d}{2} + (l'_3m'_2 + l'_2m'_3)\frac{e}{2} + l'_3m'_3 = 0$$

We can further simplify this system by using the fact that both l'_3 and m'_3 are equal to 1.

$$l'_1m'_1a + (l'_2m'_1 + l'_1m'_2)\frac{b}{2} + l'_2m'_2c + (l'_3m'_1 + l'_1m'_3)\frac{d}{2} + (l'_3m'_2 + l'_2m'_3)\frac{e}{2} + 1 = 0$$

The 5 pair of lines will construct a system of equations to find the 5 unknowns.

After finding $\mathbf{C}'^*_\infty$, we can now find the homography that maps $\mathbf{C}'^*_\infty$ to $\mathbf{C}^*_\infty$.

$$\mathbf{C}'^*_\infty = \mathbf{H}\mathbf{C}'^*_\infty\mathbf{H}^\top$$

$$\mathbf{C}'^*_\infty = \begin{pmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{v}^\top & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \mathbf{A}^\top & \mathbf{v} \\ \mathbf{0}^\top & 1 \end{pmatrix}$$

$$\mathbf{C}'^*_\infty = \begin{pmatrix} \mathbf{A}\mathbf{A}^\top & \mathbf{A}\mathbf{v} \\ \mathbf{v}^\top\mathbf{A}^\top & \mathbf{v}^\top\mathbf{v} \end{pmatrix}$$

So from this we can see that:

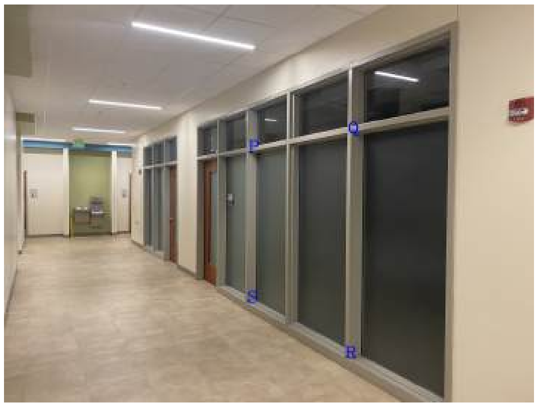
$$\mathbf{A}\mathbf{A}^\top = \begin{pmatrix} a & \frac{b}{2} \\ \frac{b}{2} & c \end{pmatrix}$$

$$\mathbf{A}\mathbf{v} = \begin{pmatrix} \frac{d}{2} \\ \frac{e}{2} \end{pmatrix}$$

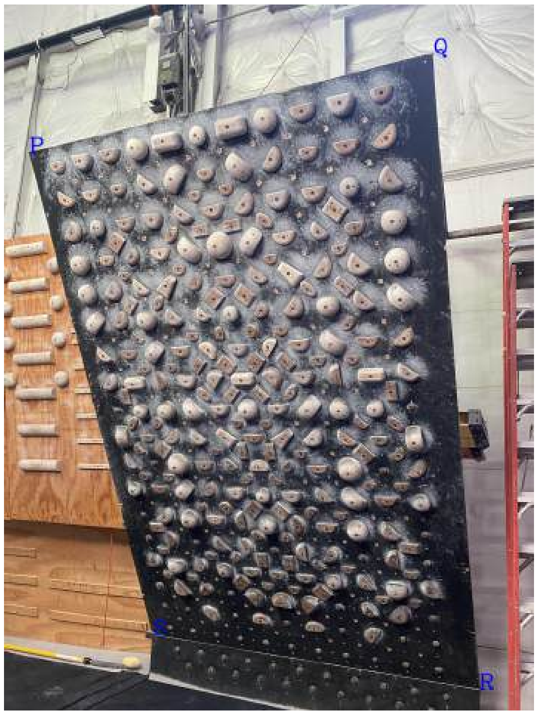
And from this we can extract both $\mathbf{A}$ and $\mathbf{v}$ using SVD, similar to how we handled the two-step method case.

Task 1: Results

Task 1.1



(a) Corridor image points.



(b) Board image points.

Figure 1: PQRS points for both images.

For the corridor image, we used the points in the distorted image: $P' = (928, 558)$, $Q' = (1306, 488)$, $R' = (1296, 1340)$, $S' = (923, 1131)$. For the reference points, or the coordinates in the undistorted image, we used: $P = (928, 558)$, $Q = (1328, 558)$, $R = (1328, 958)$, $S = (928, 958)$.
As for the board image, we used the distorted image coordinates of: $P' = (71, 421)$, $Q' = (1223, 139)$, $R' = (1354, 1954)$, $S' = (423, 1799)$. Our reference points were the following: $P = (70, 420)$, $Q = (870, 420)$, $R = (870, 1620)$, $S = (70, 1620)$.
It looks good but only for the points around that vicinity. The other points are also heavily distorted, this can be seen notoriously in the corridor image. The quality of the output in this case does suffer. The transformation in the case of the corridor image also behaved in a strange manner, see more info in the code.

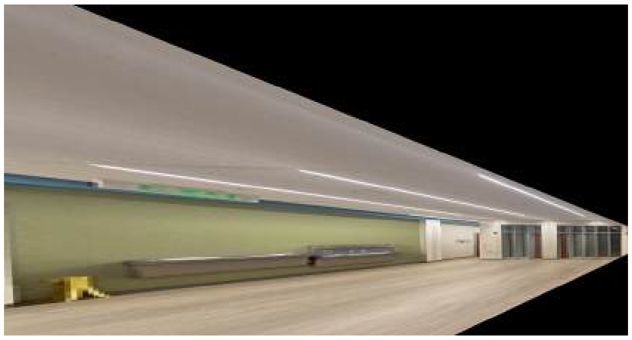


Figure 2: Corridor image undergoing point to point correspondence.

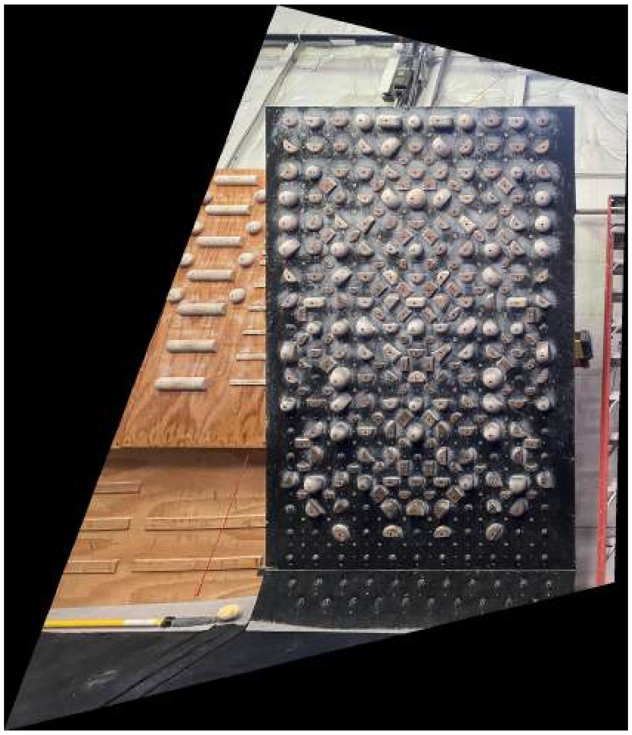
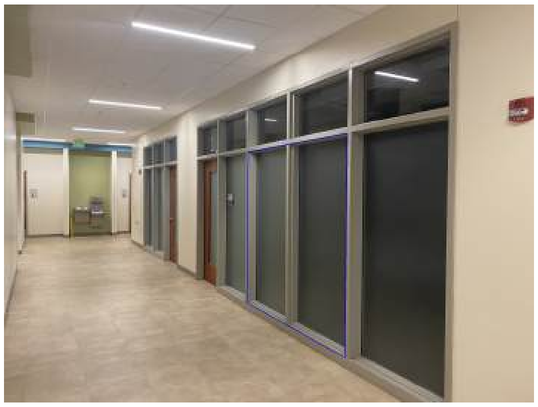


Figure 3: Board image undergoing point to point correspondence.

Task 1.2


(a) Corridor image parallel lines.

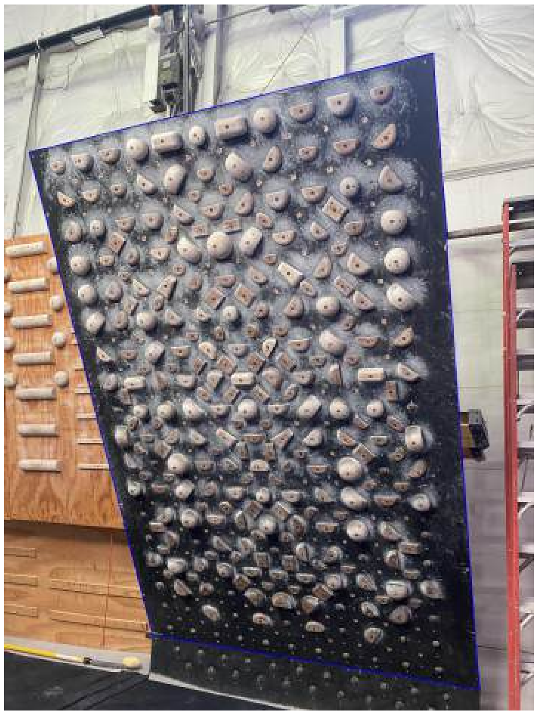

(b) Board image parallel lines.

Figure 4: Parallel lines used.

In this case, even with just the purely projective distortion, the results look like they have better quality than just point correspondence. This method also resulted in images not as stretched as point-to-point. The quality is still not the best and also there is quite a few parts in the scene that are heavily distorted, as is the case with point to point.

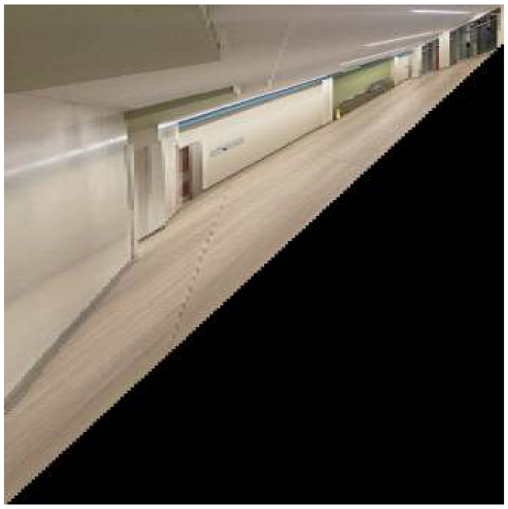
Task 1.3

For this, we tried to make use of square shapes in the image, as with that we can then reuse all the points to create our 5 pairs of orthogonal lines: (PQ-QR, QR-RS, RS-SP, SP-PQ, PR-QS). The results for the corridor image were not good, it was highly sensitive to the points used, and this combination of orthogonal lines did not yield good results. However, this contrasts sharply with the results from the board image, which are the best out of all the methods, but still leaves some parts to be desired. Not every single line in the image can be transformed to its undistorted state.

Task 2: Results

Task 2.1

For the Adobe building image, we used the following point correspondences (primes are from the distorted image): $P' = (2104, 2816)$, $Q' = (2471, 2943)$, $R' = (2475, 3437)$, $S' =$


(a) Corridor image without purely projective distortion.

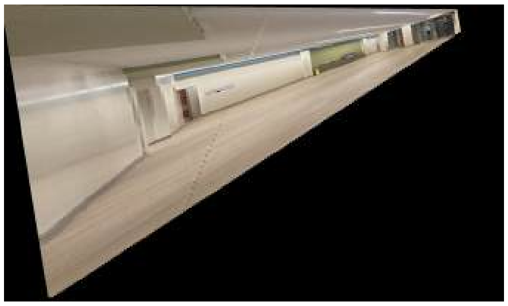

(b) Corridor image after removing affine distortion.

Figure 5: Board image undergoing 2-step method.

$(2102, 3325)$; $P = (2100, 2800)$, $Q = (2700, 2800)$, $R = (2700, 3400)$, $S = (2100, 3400)$. This is under the assumption that the area denoted by these points is roughly a square, it does seem like it is a square.
For the square image, we used the following point correspondences: $P' = (609, 493)$, $Q' = (2182, 800)$, $R' = (2086, 2364)$, $S' = (652, 2665)$; $P = (200, 200)$, $Q = (500, 200)$, $R = (500, 500)$, $S = (200, 500)$. There is no assumption about the shape since it is an image of a square. As we can see, it looks good, but only around the vicinity of the points used. Just like in task 1.

Task 2.2

Similarly to point-to-point correspondence, not all the image gets undistorted, just the parts around the vicinity or lines that were near or have the same orientation.

Task 2.3

We can see that this has produced the cleaner results. One-step method produces good results, but it is quite sensitive to the points chosen. We noticed it was also fastest compared to the other methods, and it didn't produce too much of a stretched output as compared to the other 2 approaches.

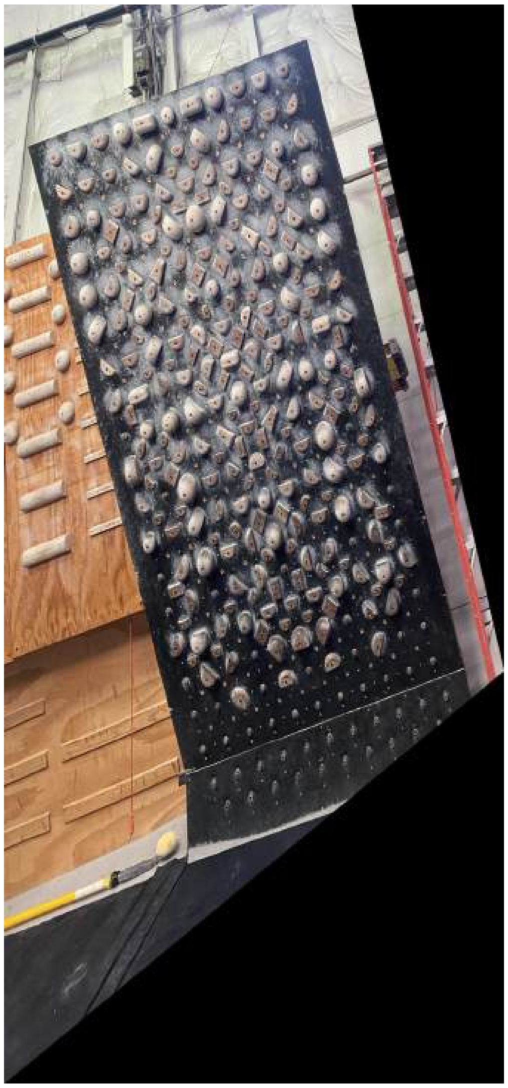
0.1 Source Code

```
import cv2
import matplotlib.pyplot as plt
import numpy as np

def point_to_point_system(domain, range_):
    # requires both dictionaries to have PQRS in [x,y] format
    # this will fill the matrices used to find H

    mat1 = np.empty((0,8),dtype=float)
    mat2 = np.empty((0,0),dtype=float)
    for i in range(domain.shape[0]):

        x = domain[i,0]
        x_prime = range_[i,0]
        y = domain[i,1]
        y_prime = range_[i,1]

        mat1 = np.append(mat1,np.array([[x, y, 1, 0, 0, 0, -x*x_prime, -y*x_p
        mat1 = np.append(mat1,np.array([[0, 0, 0, x, y, 1, -x*y_prime, -y*y_p
        mat2 = np.append(mat2,x_prime)
        mat2 = np.append(mat2,y_prime)

    return mat1,mat2.reshape(8,1)

def calculate_H_matrix(domain,range):
    mat1, mat2 = point_to_point_system(domain, range)
    # we use this function to solve the equation described in the logic,
    # I replaced np.dot with @ as that is what they suggest in the numpy webs
    sol = np.linalg.inv(mat1) @ mat2
    # we append the 1 since this will only have 8 values, the 1 is missing
    sol = np.append(sol,np.array([[1]]),axis=0)
    return sol.reshape((3,3)).astype(float)

def homography_transform(H, point):
    # this function maps coordinates (x,y) to (x',y') using the homography H
    #p_prime = np.array([point[0],point[1],1],)
    p = np.append(np.array(point),[1]).astype(int)
    prime_p = H @ p
    return (prime_p/prime_p[2]).astype(int)

def apply_homography(image, H):
    # this function transforms the entire image according to homography matri
    # I print out the corners, since our transformation may make the image "g
    # I created another function to handle such cases, and that is the one we
    corners = find_corners(image,H)
    print(corners)
    H_inv = np.linalg.inv(H)
    canvas = np.zeros_like(image)
    for i in range(canvas.shape[1]):
        for j in range(canvas.shape[0]):
            pos = [i,j]
            tpos = homography_transform(H_inv,pos)
            if 0 < tpos[0] < canvas.shape[1] and 0 < tpos[1] < canvas.shape[0
                canvas[j,i] = image[tpos[1],tpos[0]]
    return canvas

def find_corners(image,H, mod=False):
  #need to find these corners to be able to resize the image
  # I found a strange interaction in some cases, where the x' values should b
  # so I handle those with mod=True/False, this mainly happens with the corri
  p = [0,0]
  q = [image.shape[1],0]
  r = [image.shape[1],image.shape[0]]
  s = [0,image.shape[0]]

  p_p = homography_transform(H,p)
  q_p = homography_transform(H,q)
  r_p = homography_transform(H,r)
  s_p = homography_transform(H,s)
  if mod:
    p_p = -p_p
    s_p = -s_p
  points_prime = np.array([p_p,q_p,r_p,s_p])
  corn = {}
  corn["xmin"] = np.min(points_prime[:,0])
  corn["xmax"] = np.max(points_prime[:,0])
  corn["ymin"] = np.min(points_prime[:,1])
  corn["ymax"] = np.max(points_prime[:,1])
  return corn

def apply_homography_resize(image, H, mod=False):
    # this function transforms the entire image according to homography matri
    # this is the function that resizes the image depending on where the corn
    # need to add the top left (xmin,ymin) to each position before transformi
    corners = find_corners(image,H,mod=mod)
    print(corners)
    w = int(corners["xmax"] - corners["xmin"])
    h = int(corners["ymax"] - corners["ymin"])
    canvas = np.zeros((h,w,3)).astype(np.uint8)
    H_inv = np.linalg.inv(H)
    for i in range(canvas.shape[1]):
        for j in range(canvas.shape[0]):
            pos = [int(i+corners["xmin"]),int(j+corners["ymin"])]
            tpos = homography_transform(H_inv,pos)
            if 0 <= tpos[0] < image.shape[1] and 0 <= tpos[1] < image.shape[0
                canvas[j,i] = image[tpos[1],tpos[0]]
    return canvas

def get_point_or_line(point1, point2):
    #take in 2d or 3d vectors and get their cross product, we end up dividing

    if len(point1) == 2 and len(point2)== 2:
        point1 = np.array(point1)
        point2 = np.array(point2)
        point1 = np.append(point1,1)
        point2 = np.append(point2,1)
    r = np.cross(point1,point2)
    return r/r[2]

##### TWO STEP

def get_vanishing_line(arr):
    #we use the pairs of, what we know are orthogonal in the scene, lines: PQ

    #PQ
    line_PQ = get_point_or_line(arr[0],arr[1])
    #RS
    line_RS = get_point_or_line(arr[2],arr[3])
    #PS
    line_PS = get_point_or_line(arr[0],arr[3])
    #QR
    line_QR = get_point_or_line(arr[1],arr[2])

    vpoint1 = get_point_or_line(line_PQ, line_RS)

    vpoint2 = get_point_or_line(line_PS, line_QR)

    return get_point_or_line(vpoint1,vpoint2)

def remove_vanishing_line(arr):
    # we use this to calculate the H matrix that sends the vanishing line to
    vanishing_line = get_vanishing_line(arr)
    H_vl = np.eye(3)
    H_vl[2,0] = vanishing_line[0]
    H_vl[2,1] = vanishing_line[1]
    return H_vl

def two_step_system(arr):
    #PQ
    line_PQ = get_point_or_line(arr[0],arr[1])
    #QR
    line_QR = get_point_or_line(arr[1],arr[2])
    #RS
    line_RS = get_point_or_line(arr[2],arr[3])
    #PS
    line_PS = get_point_or_line(arr[0],arr[3])

    #following the linear system from the logic
    mat1 = np.empty((0,2),dtype=float)
    mat2 = np.empty((0,0),dtype=float)
    mat1 = np.append(mat1,np.array([[line_PQ[0] * line_QR[0], line_PQ[0] * lin
    mat1 = np.append(mat1,np.array([[line_RS[0] * line_PS[0], line_RS[0] * lin
    mat2 = np.append(mat2,np.array([-line_PQ[1] * line_QR[1]]))
    mat2 = np.append(mat2,np.array([-line_RS[1] * line_PS[1]]))

    return mat1, mat2.reshape((2,1))

def remove_affine_distortion(arr):
    # follows from the logic
    mat1, mat2 = two_step_system(arr)
    res = np.linalg.inv(mat1)@mat2
    S = np.eye(2)
    S[0,0] = res[0]
    S[0,1] = res[1]
    S[1,0] = S[0,1]

    V, D, _ = np.linalg.svd(S)
    A_eigenvalues = np.sqrt(np.diag(D))
    A = V @ A_eigenvalues @ V.transpose()
    H = np.eye(3,3)
    H[:2,:2] = A

    return np.linalg.inv(H)


##### TWO STEP

def orthogonal_lines(arr):
    #orthogonal pair 1
    line_11 = get_point_or_line(arr[0][0],arr[0][1])
    line_12 = get_point_or_line(arr[0][2],arr[0][3])
    #orthogonal pair 2
    line_21 = get_point_or_line(arr[1][0],arr[1][1])
    line_22 = get_point_or_line(arr[1][2],arr[1][3])
    #orthogonal pair 3
    line_31 = get_point_or_line(arr[2][0],arr[2][1])
    line_32 = get_point_or_line(arr[2][2],arr[2][3])
    #orthogonal pair 4
    line_41 = get_point_or_line(arr[3][0],arr[3][1])
    line_42 = get_point_or_line(arr[3][2],arr[3][3])
    #orthogonal pair 5
    line_51 = get_point_or_line(arr[4][0],arr[4][1])
    line_52 = get_point_or_line(arr[4][2],arr[4][3])

    return np.array([[line_11,line_12],[line_21,line_22],[line_31, line_32],[

def one_step_system(arr):
    # following the system of equations from the logic
    lines = orthogonal_lines(arr)
    print(lines)
    mat1 = np.empty((0,5),dtype=float)
    mat2 = np.empty((0,0),dtype=float)
    for pair in lines:
        mat1 = np.append(mat1,np.array([[pair[0][0]*pair[1][0],pair[0][0]*pai
        mat2 = np.append(mat2,-1)
    return mat1,mat2.reshape((5,1))

def calculate_conic_prime(arr):
    # we get the conic C'_\infty
    mat1,mat2 = one_step_system(arr)
    res = (np.linalg.inv(mat1)@mat2).reshape((5,))
    res /= np.max(res)
    C = [[res[0],res[1],res[3]],[res[1],res[2],res[4]],[res[3],res[4],1]]
    return np.array(C,dtype=float)

def get_H_onestep(C):
    # get the H matrix from the conic
    V, D, _ = np.linalg.svd(C[:2,:2])
    A_eigenvalues = np.sqrt(np.diag(D))
    A = V.transpose() @ A_eigenvalues @ V
    H = np.zeros((3,3),dtype=float)
    H[:2,:2] = A
    v = np.linalg.inv(A) @ [C[2,0],C[2,1]]
    H[2,:2] = v.transpose()
    H[2,2] = 1
    return np.linalg.inv(H)

def rescale(image, new_width,new_height):
    # this function I use to rescale, in some cases where the image is too bi
    H = np.array([[new_width/image.shape[1],0,0],[0,new_height/image.shape[0]
    new_image = apply_homography_resize(image,H)
    return new_image

##### TASK 1 : CORRIDOR

image = cv2.imread("corridor.jpeg")
corridor = np.array([[928, 558],[1306, 488],[1296, 1340],[923, 1131]])
ref = np.array([[928,558],[1328,558],[1328,958],[928,958]])

H = calculate_H_matrix(ref,corridor)
# the only time we use the modification for the apply_homography function
a = apply_homography_resize(image, np.linalg.inv(H),mod=True)

cv2.imwrite("corridor_p2p.jpg",a)

H_v = remove_vanishing_line(corridor)
print(H_v)
b = apply_homography_resize(image, np.linalg.inv(H_v))

new_b = rescale(b, 3000,3000)

cv2.imwrite("corridor_2stepint.jpg",new_b)

H_aff = remove_affine_distortion(corridor)

c = apply_homography_resize(b, H_aff)

cv2.imwrite("corridor_2step.jpg",c)

P = [815,577]
Q = [1073,531]
R = [1069,1214]
S = [811,1069]
# these are the orthogonal lines
o_lines = np.array([[
            # PR- QS, diagonals
            P,R,
            Q,S],
                [
                    #PQ - QR
                P,Q,
                Q,R],
                [ # QR - RS
                Q,R,
                R,S],
                [ # RS - SP
                R,S,
                S,P],
                [ # SP - PQ
                S,P,
                P,Q

                ]])

conic = calculate_conic_prime(o_lines)
H = get_H_onestep(conic)

d = apply_homography_resize(image, np.linalg.inv(H))
# we rescale here, otherwise it is too thin
new_d = rescale(d, 3000,3000)

cv2.imwrite("corridor_1step.jpg",new_d)

######## TASK 1: BOARD

image2 = cv2.imread("board_1.jpeg")
board = np.array([[71,421],[1223,139],[1354,1954],[423,1799]])
ref = np.array([[70,420],[870,420],[870,1620],[70,1620]])

plt.imshow(image)

H = calculate_H_matrix(ref,board)
a = apply_homography_resize(image2, np.linalg.inv(H))

cv2.imwrite("board_p2p.jpg",a)

H_v = remove_vanishing_line(board)
b = apply_homography_resize(image2, H_v)
cv2.imwrite("board_2stepint.jpg",b)

H_aff = remove_affine_distortion(board)
c = apply_homography_resize(b, H_aff)
cv2.imwrite("board_2step.jpg",c)
```

```
P = [811,375]
Q = [916,353]
R = [929,470]
S = [825,490]

o_lines = np.array([[
            # PR- QS, diagonals
            P,R,
            Q,S],
            [
                #PQ - QR
            P,Q,
            Q,R],
            [ # QR - RS
            Q,R,
            R,S],
            [ # RS - SP
            R,S,
            S,P],
            [ # SP - PQ
            S,P,
            P,Q

            ]])
conic = calculate_conic_prime(o_lines)
H = get_H_onestep(conic)
d = apply_homography_resize(image2, H)

cv2.imwrite("board_1step.jpg",d)

#### TASK 2: ADOBE BUILDING
adobe_img = cv2.imread("adobe.jpeg")
P = [2104,2816]
Q = [2471,2943]
R = [2475,3437]
S = [2102,3325]
adobe = np.array([P,Q,R,S])
ref = np.array([[2100,2800],[2700,2800],[2700,3400],[2100,3400]])

H = calculate_H_matrix(ref,adobe)
a = apply_homography_resize(adobe_img, np.linalg.inv(H))

cv2.imwrite("adobe_point2point.jpg",a)
```

```
H_v = remove_vanishing_line(adobe)
b = apply_homography_resize(adobe_img, H_v)

cv2.imwrite("adobe_2stepint.jpg",b)

H_aff = remove_affine_distortion(adobe)
c = apply_homography_resize(b, H_aff)
# this image is too big in size, so I rescale it here
new_c = rescale(c, 3000,1400)

cv2.imwrite("adobe_2step.jpg",new_c)

o_lines = np.array([[
            # PR- QS, diagonals
            P,R,
            Q,S],
            [
                #PQ - QR
            P,Q,
            Q,R],
            [ # QR - RS
            Q,R,
            R,S],
            [ # RS - SP
            R,S,
            S,P],
            [ # SP - PQ
            S,P,
            P,Q

            ]])

conic = calculate_conic_prime(o_lines)
H = get_H_onestep(conic)
d = apply_homography_resize(adobe_img, H)

cv2.imwrite("adobe_onestep.jpg",d)

square_img = cv2.imread("square.jpg")
P = [609,493]
Q = [2182,800]
R = [2086,2364]
S = [652,2665]
square = np.array([P,Q,R,S])
ref = np.array([[200,200],[500,200],[500,500],[200,500]])
```

```
H = calculate_H_matrix(ref,square)
a = apply_homography_resize(square_img, np.linalg.inv(H))

cv2.imwrite("square_p2p.jpg",a)

H_v = remove_vanishing_line(square)
b = apply_homography_resize(square_img, H_v)
cv2.imwrite("square_2stepint.jpg",b)

H_aff = remove_affine_distortion(square)
c = apply_homography_resize(b, H_aff)
# this image size is too big, so we resize it here
new_c = cv2.resize(c,(1920,1150))

cv2.imwrite("square_2step.jpg",new_c)

o_lines = np.array([[
            # PR- QS, diagonals
            P,R,
            Q,S],
            [
                #PQ - QR
            P,Q,
            Q,R],
            [ # QR - RS
            Q,R,
            R,S],
            [ # RS - SP
            R,S,
            S,P],
            [ # SP - PQ
            S,P,
            P,Q

            ]])
conic = calculate_conic_prime(o_lines)
H = get_H_onestep(conic)
d = apply_homography_resize(square_img, H)
cv2.imwrite("square_onestep.jpg",d)
```


(a) Board image without purely projective distortion.

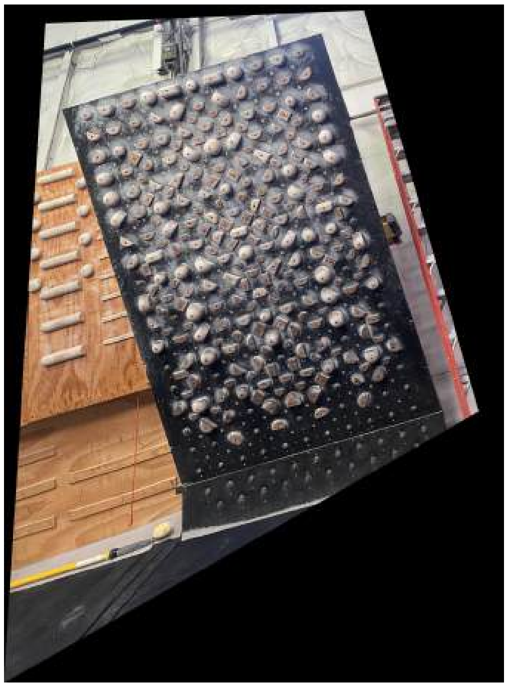
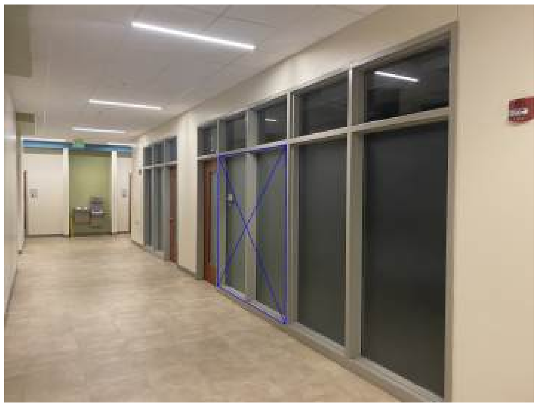

(b) Board image after removing affine distortion.

Figure 6: Board image undergoing 2-step method.


(a) Corridor image orthogonal lines.

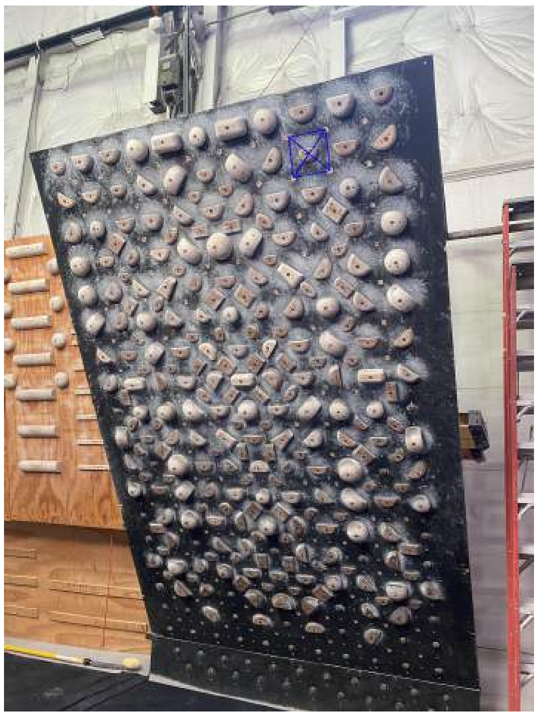

(b) Board image orthogonal lines.

Figure 7: Orthogonal lines used.

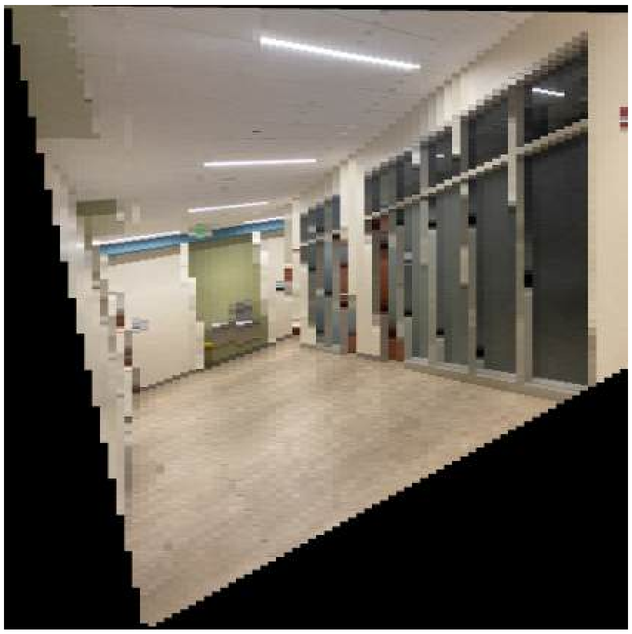


Figure 8: Corridor image undergoing one-step method.

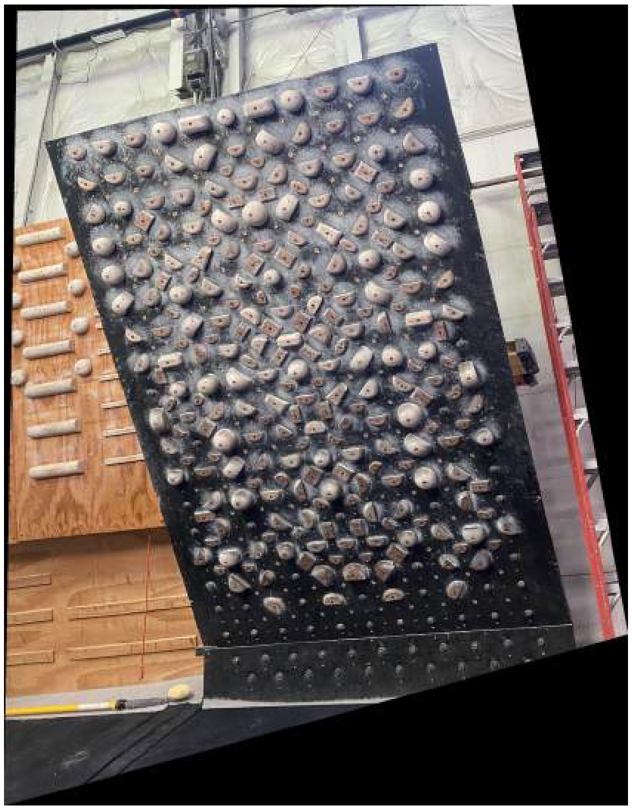
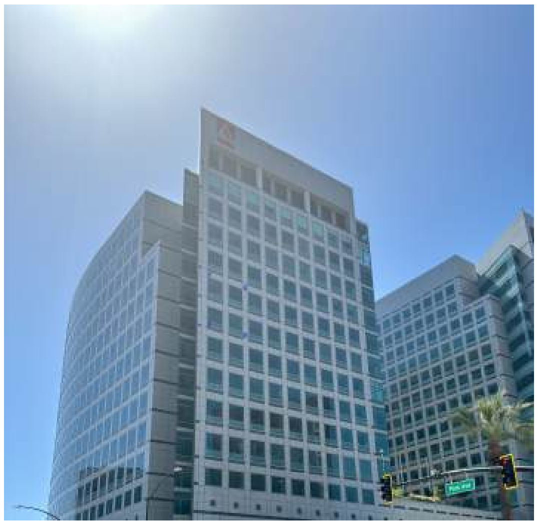


Figure 9: Board image undergoing one-step method.


(a) Adobe building image points.

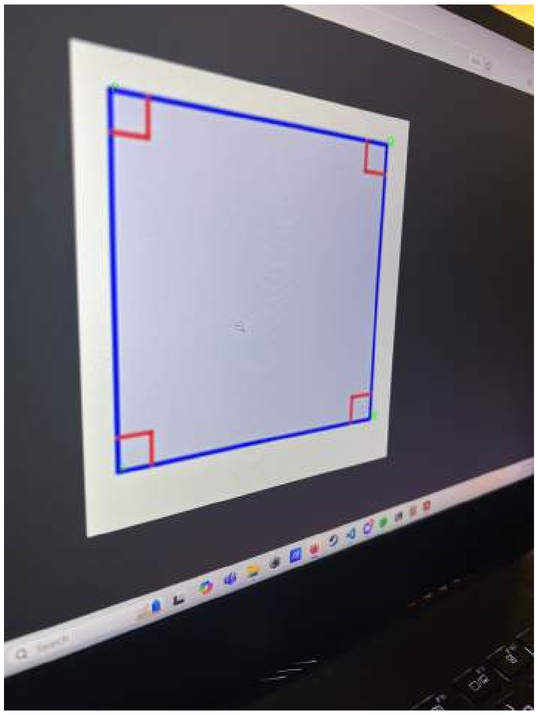

(b) Square image points.

Figure 10: PQRS points for both images.

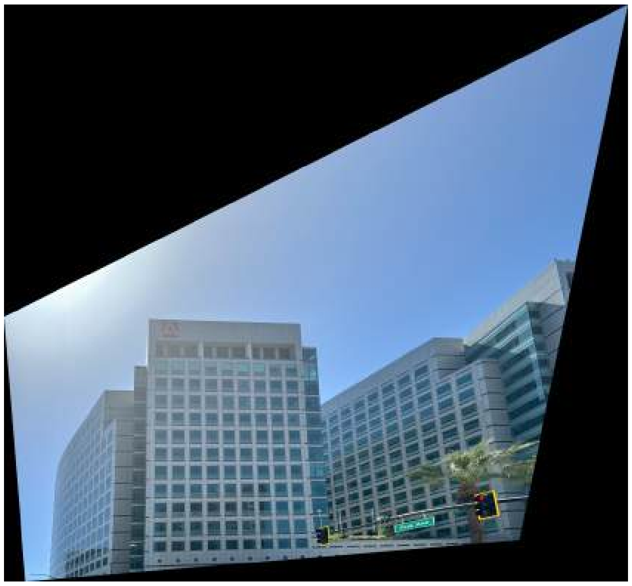


Figure 11: Adobe building undergoing point-to-point correspondence.

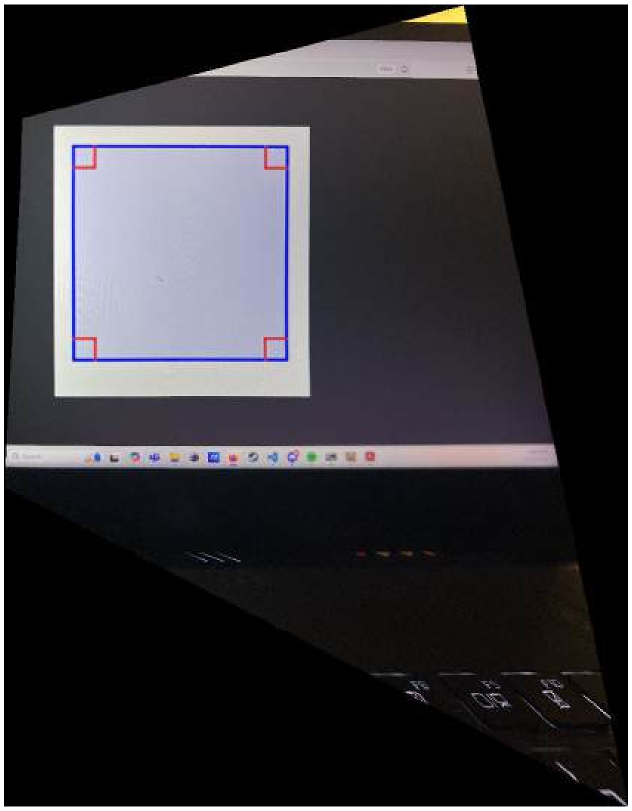
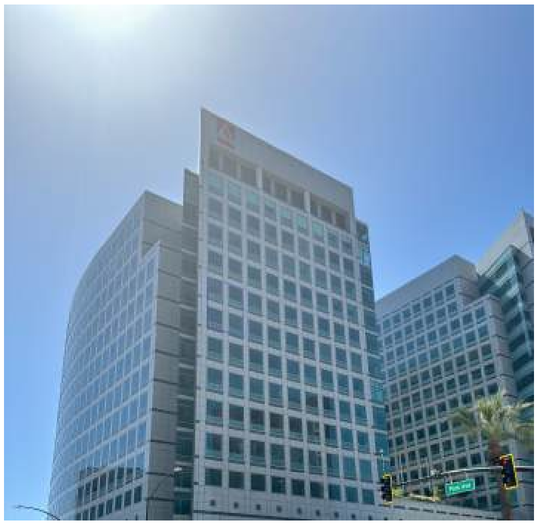


Figure 12: Square image undergoing point-to-point correspondence.


(a) Adobe building image parallel lines.

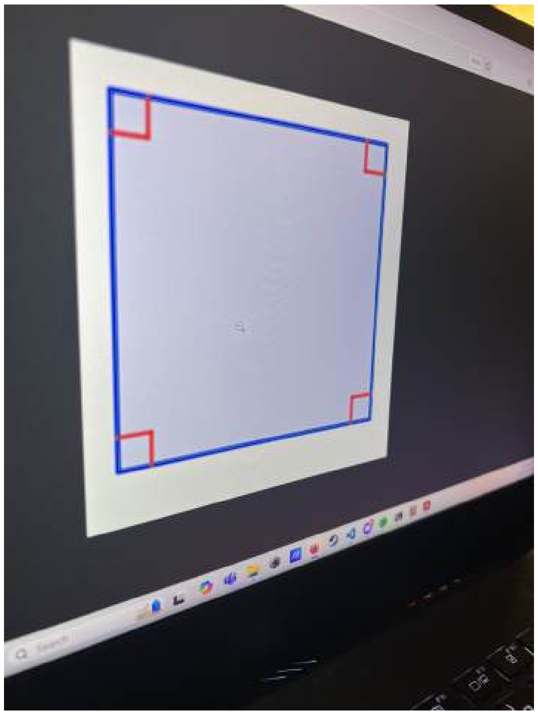
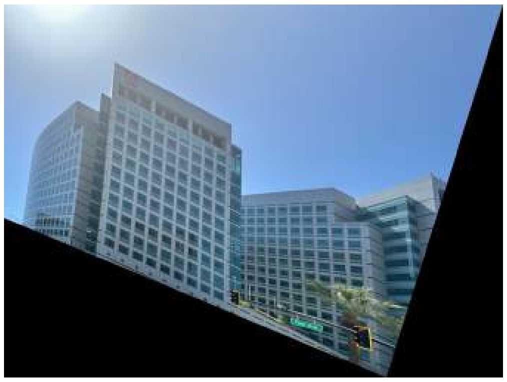

(b) Square image parallel lines.

Figure 13: Parallel lines used.


(a) Adobe image without purely projective distortion.

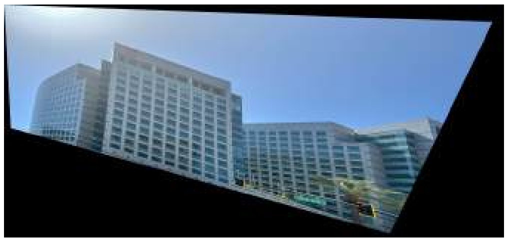
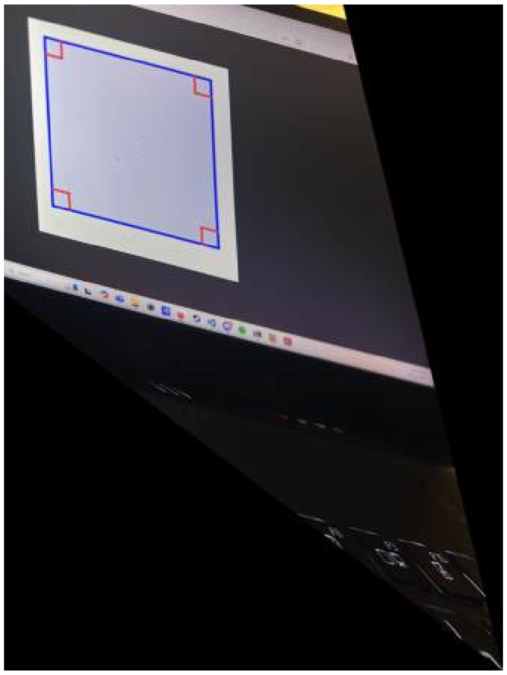

(b) Adobe image after removing affine distortion.

Figure 14: Adobe building image undergoing 2-step method.


(a) Square image without purely projective distortion.

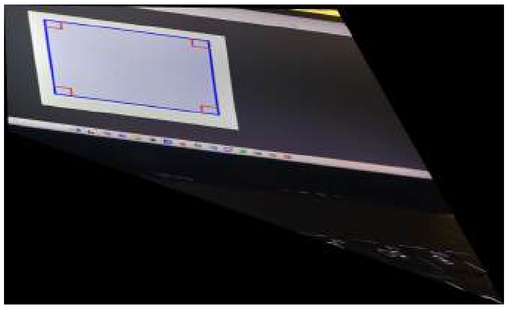

(b) Square image after removing affine distortion.

Figure 15: Square image undergoing 2-step method.

(a) Adobe building image orthogonal lines.

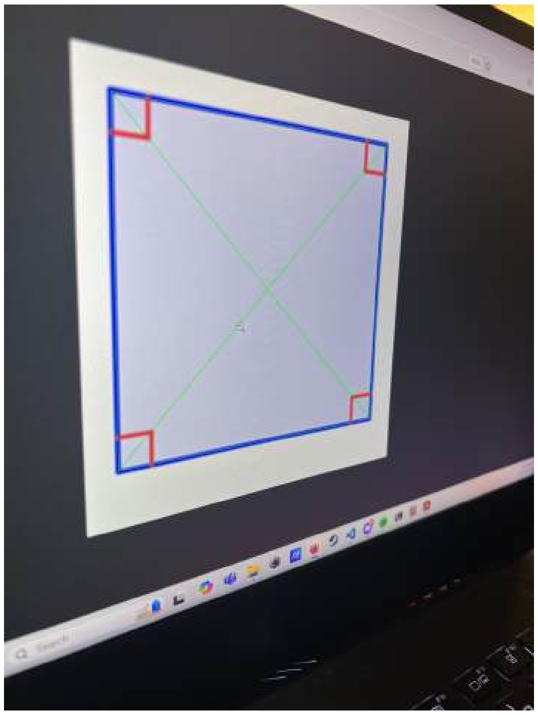

(b) Square image orthogonal lines.

Figure 16: Orthogonal lines used.

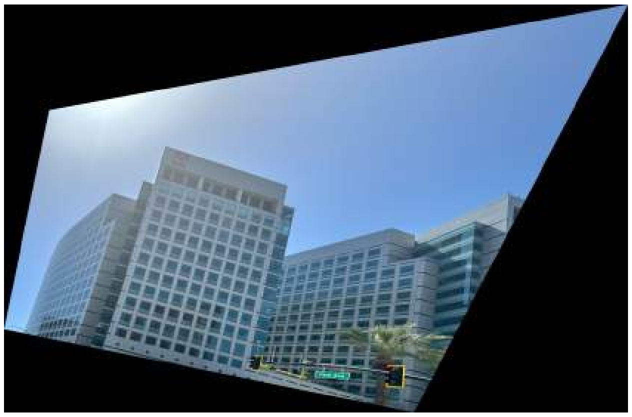


Figure 17: Adobe building image after one-step method.

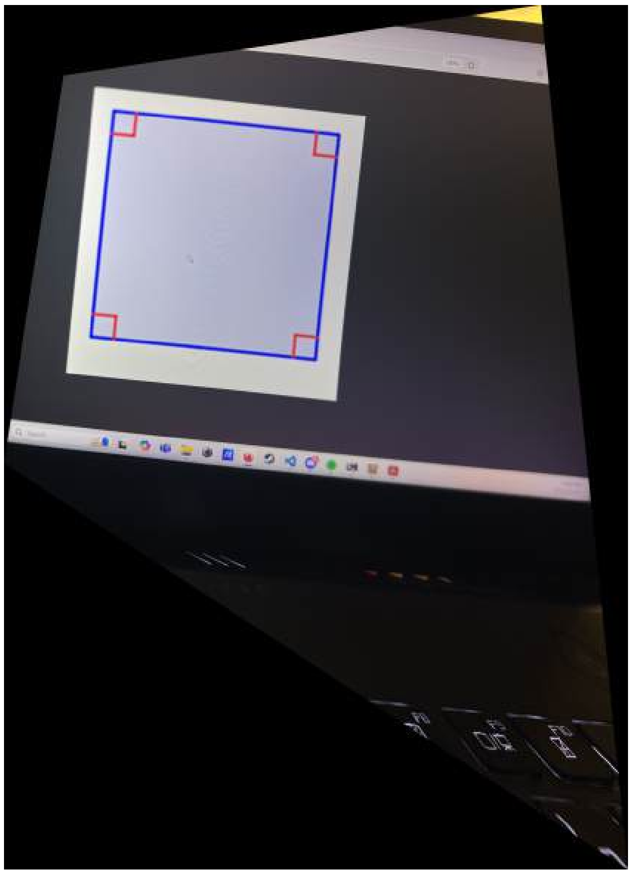


Figure 18: Square image after one-step method.