HOMEWORK 3

Alexandre Olivé Pellicer

aolivepe@purdue.edu

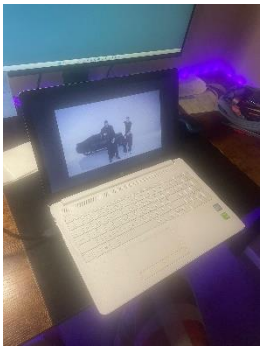
In this report we will show the steps, results and procedure and observation for each of the 3 different methods used to transform images. For each method we will show the result in 4 different images: 2 given in the instructions by task 1 and 2 selected by me. We show the results together.

Image 1: Board

Image 2: Corridor

Image 3: TV

Image 4: Laptop



Point to Point method

We know that we can map a point from the domain plain to the range plain by using homogeneous coordinates and applying a homography H as:

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

$$H = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

From this product of matrices, we can write the following equations:

$$x_1' = ax_1 + bx_2 + cx_3$$

$$x_2' = dx_1 + ex_2 + fx_3$$

$$x_3' = gx_1 + hx_2 + ix_3$$

Knowing that $x' = \frac{x_1'}{x_3'}$ and $y' = \frac{x_2'}{x_3'}$ we can write:

$$x' = \frac{x_1'}{x_3'} = \frac{ax_1 + bx_2 + cx_3}{gx_1 + hx_2 + ix_3}$$

$$y' = \frac{x_2'}{x_3'} = \frac{dx_1 + ex_2 + fx_3}{gx_1 + hx_2 + ix_3}$$

Dividing numerator and denominator by 3 and setting i=1 since we are just interested in ratios:

$$x' = \frac{ax + by + c}{gx + hy + 1}$$

$$y' = \frac{dx + ey + f}{gx + hy + 1}$$

We can write this as:

$$x' = ax + by + c - gxx' + hyx'$$

$$y' = dx + ey + f - gxy' - hyy'$$

In these 2 equations we have in total 8 unknowns corresponding to the 9 coefficients of the matrix H since we have set i=1. To find these 8 unknowns, we need 8 equations. By taking 4 points from the domain plain and looking at which would be their corresponding points in the range plain, we will get 8 equations that will allow us to find the remaining 8 coefficients of the matrix H. We can write these 8 equations as a product of matrixes where x_i' and y_i' correspond to the points in the 2D range plain and x_i and y_i correspond to the points in the 2D domain plain:

$$\begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x_1' & -y_1x_1' \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y_1' & -y_1y_1' \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x_2' & -y_2x_2' \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y_2' & -y_2y_2' \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x_3' & -y_3x_3' \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y_3' & -y_3y_3' \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x_4' & -y_4x_4' \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y_4' & -y_4y_4' \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{pmatrix} = \begin{pmatrix} x_1' \\ y_1' \\ x_2' \\ y_2' \\ x_3' \\ y_3' \\ x_4' \\ y_4' \end{pmatrix}$$

We can represent this product of matrices as Ah = b. Thus, if we solve $h = A^{-1}b$ we will find the remaining coefficients of the matrix H.

For the point to point method, we use 4 points in the domain image to solve $h = A^{-1}b$

These are the steps we follow:

1. Choose the ROI in the distorted image (range image) by selecting four points, P, Q, R, and S.
2. Select the points in the undistorted (domain) image taking into consideration the real distances provided in the instructions.
3. Once we have the points P, Q, R, and S for both the domain and range images, we compute the homography as mentioned above.
4. We then apply the inverse homography to transform each point in the range image back to the domain image, avoiding artifacts. We take every point in the range plane and apply inverse homography to convert it to the domain plain.

We follow this for each of the images

Points used in each image in the range plain:

Board:

P	(70,420)
Q	(422,1759)
R	(1356,1952)
S	(1222,139)

Corridor:

P	(829,576)
Q	(843,1060)
R	(1301,1339)
S	(1304,490)

TV:

P	(829,576)
Q	(843,1060)
R	(1301,1339)
S	(1304,490)

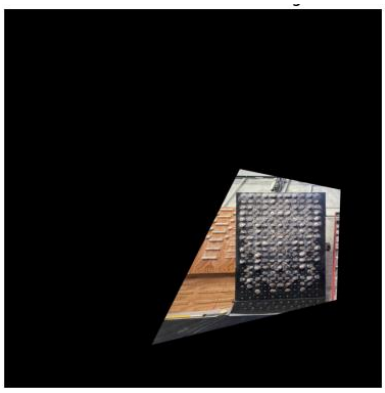
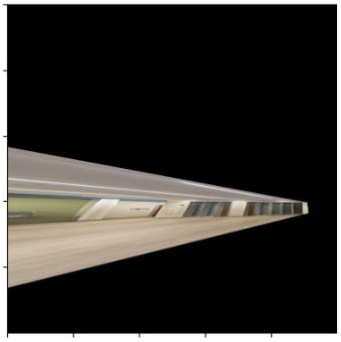
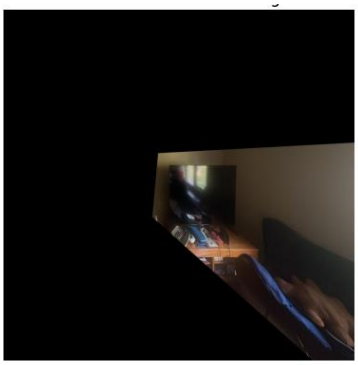
Laptop

P	(829,576)
Q	(843,1060)
R	(1301,1339)
S	(1304,490)

Homographies computed:

$$H\_\text{board} = \begin{bmatrix} 1.28 & -0.34 & 121.01 \\ 0.28 & 0.61 & 111.63 \\ 2.75\text{ e-}4 & \text{-}2.21\text{e-}4 & 1 \end{bmatrix}$$

Obtained results:


Observations:

The presented images clearly show the distortion we were looking for. It is clear that for all images parallel lines go to parallel lines and the angles are the same. The advantage of using this method is that it is simple and that there are no need of intermediate steps to obtain the results. The negative aspect is that many points need to be captured in order to fund the homography.

<u>Two-step method</u>

1. Select 2 pairs of 2 parallel lines. Select 2 points per line. In total 8 points. We convert them to homogeneous coordinates by adding “1”.
2. Given the two points that define a line, we compute the equation of the line using the cross product. We do this for the 4 lines.

3. For each pair of parallel lines, we compute the vanishing point using the cross product of the two parallel lines. We do it for both pairs of parallel lines so that we get 2 vanishing points. We compute the vanishing line by computing the product of both vanishing points.

4. If the vanishing line in homogeneous coordinates is given by $(l1, l2, l3)$, we can write the homography matrix that caused the projective distortion as:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l1 & l2 & l3 \end{bmatrix}$$

Demonstration:
Lines are transformed by H^{-T}

$$H^{-T} = \begin{bmatrix} 1 & 0 & -\frac{l1}{l3} \\ 0 & 1 & -\frac{l2}{l3} \\ 0 & 0 & \frac{1}{l3} \end{bmatrix}$$

$$H^{-T}l = \begin{bmatrix} 1 & 0 & -\frac{l1}{l3} \\ 0 & 1 & -\frac{l2}{l3} \\ 0 & 0 & \frac{1}{l3} \end{bmatrix} \begin{matrix} l1 \\ [l2] \\ l3 \end{matrix} = \begin{matrix} l1 \\ [l2] \\ l3 \end{matrix} = l_\infty$$

5. We compute the inverse of the image and apply it to the distorted images to remove the projective distortion.

Removing Affine Distortion with Dual Degenerate Conic

1. In the original distorted image we select two pairs of right angles. The right angles is described by 3 points. The first line "l'" is obtained by the point in the right angle and one of the other two points. Line "m'" is obtained using the point in the right angle and the other point.
2. We estimate H such that $l'^T H C_\infty * H^T m' = 0$

$$C_\infty * = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$H = \begin{bmatrix} a11 & a12 & 0 \\ a21 & a22 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$l' = \begin{pmatrix} l_1 \\ l_2 \\ l_3 \end{pmatrix}$$

$$m' = \begin{pmatrix} m_1 \\ m2 \\ m3 \end{pmatrix}$$

$$A = \begin{bmatrix} a11 & a12 \\ a21 & a22 \end{bmatrix}$$

$$A^T A = \begin{bmatrix} s11 & s12 \\ s12 & s22 \end{bmatrix}$$

We get the following matrix equation:

$$(l1'm1' \;\; l1'm2'+m1'l2' \;\; l2'm2') \begin{pmatrix} s11 \\ s12 \\ s22 \end{pmatrix} = 0$$

Representing the coordinates of the second right angle by “:

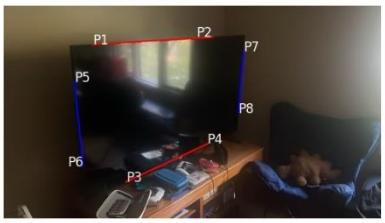
$$\begin{pmatrix} l1'\,m1' & l1'\,m2' + m1'\,l2' & l2'\,m2' \\ l1''\,m1'' & l1''\,m2'' + m1''\,l2'' & l2''\,m2'' \end{pmatrix} \begin{pmatrix} s11 \\ s12 \\ s22 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

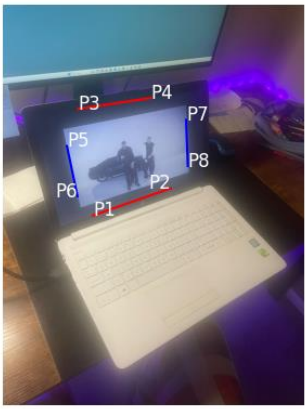
The above equation is of the the type PS=0. Thus, S is the null space of P. When we find S we find $A^T A$

Once $A^T A$ is found, we can do Singular Value Decomposition to find A. We take the eigenvalues of A as the square root of $A^T A$ and the eigenvectors of A as the eigenvectors of $A^T A$. Finally, compute A as U.Sigma.V^T

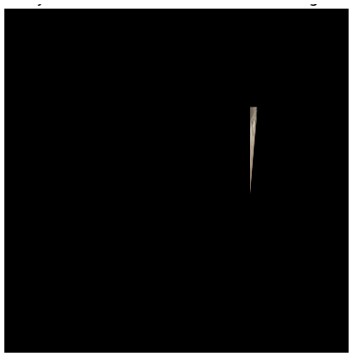
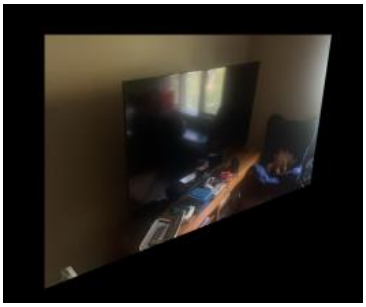
Once A is found, we have already found the affine homography. We multiply the projective homography and affine homography and we apply it to the distorted image to get rid of the distortion

Parallel lines used for each image:

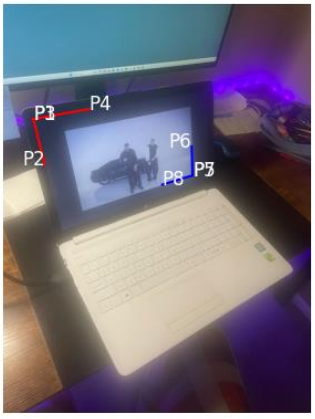




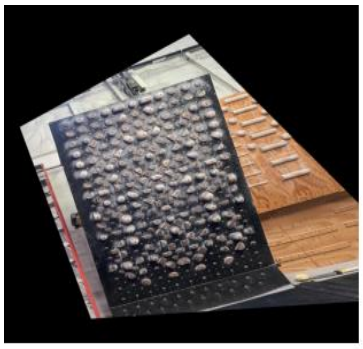
Projective distortion corrected

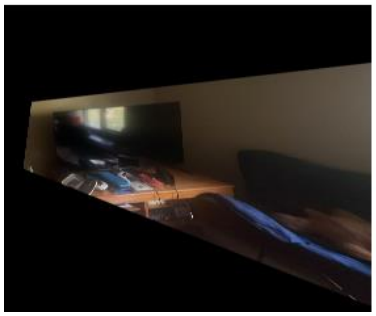





Perpendicular lines used for each image:



Affine distortion corrected:






Observations:

We clearly see that for images 1, 3 and 4 our implementation works. First, by removing the pure projective transform, we set parallel lines. This is clearly achieved for the 3 mentioned images. Afterwards, the affine distortion correction is applied, and we see how angles are kept. For some reason the obtained results with image 2 are not the expected ones. One possible reason is that the angle between the camera and the wall with the windows is very small. Therefore, the imprecision when taking the correct points to define the desired parallel lines is higher. We have tried with different sets of points, and it has not been possible to solve the problem. Since the results with the other methods is the desired one, we consider that we successfully completed the task.

The main advantage of this method with respect to the point to point method is that simplifies the process of solving it. The inconvenience is that it needs precision in defining the parallel lines otherwise the results can be unexpected. Same for perpendicular lines.

One-step method

1. We take 5 right angles in original distorted images
2. Each right angle is represented by three points as mentioned in the "Two-step method"
3. We get the equations of perpendicular lines by the cross product of the three points as explained before. In total we get 5 pairs of perpendicular lines
4. Given 2 perpendicular lines with their homogeneous coordinates l=(l11, l12, l13) and m=(m11, m12, m13) we compute $C^{*'}$ such that $l'^{T}C^{*'}m' = 0$:

$$[l11\,l12\,l13]\begin{bmatrix} a & b/2 & d/2 \\ b/2 & c & e/2 \\ d/2 & e/2 & f \end{bmatrix}\begin{pmatrix} m11 \\ m12 \\ m13 \end{pmatrix} = 0$$

5. Solving it we get:
   a(l11 m11) +b(l12 m11/2 + l11 m12/2) + c(l12 m12) d(l13 m11/2 + l11 m13/2) + e(l13 m12/2 + l12 m13/2) = -l13 m13

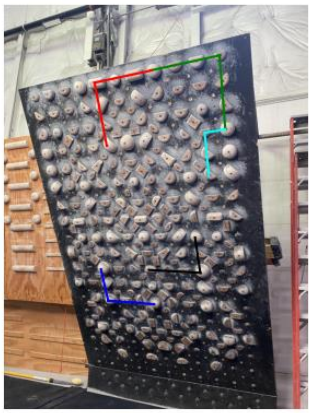
For the five pairs of perperdicular equations we can write the system of equations

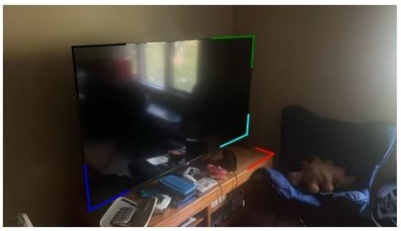
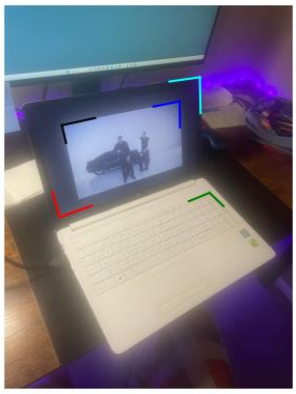
$$\begin{array}{llllll} l11\,m11 & l12\,m11/2 + l11\,m12/2 & l12\,m12 & l13\,m11/2 + l11m13/2 & l13\,m12/2 + l12\,m13 \\ l21\,m21 & l22\,m21/2 + l21\,m22/2 & l22\,m22 & l23\,m21/2 + l21m13/2 & l23\,m22/2 + l22\,m23 \\ l31\,m31 & l32\,m31/2 + l31\,m32/2 & l32\,m32 & l33\,m31/2 + l31m13/2 & l33\,m32/2 + l32\,m33 \\ l41\,m41 & l42\,m41/2 + l41\,m42/2 & l42\,m42 & l43\,m41/2 + l41m13/2 & l43\,m42/2 + l42\,m43 \\ l51\,m51 & l52\,m51/2 + l51\,m52/2 & l52\,m52 & l53\,m51/2 + l51m13/2 & l53\,m52/2 + l52\,m53 \end{array}$$

a -l13 m13
b -l23 m23
c= -l33 m33
d -l43 m43
d -l53 m53

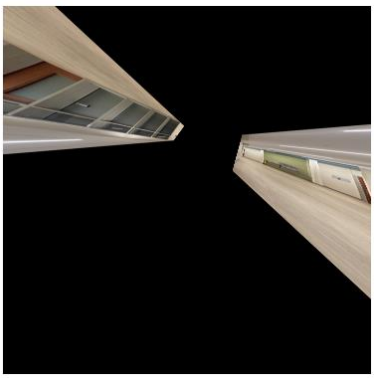
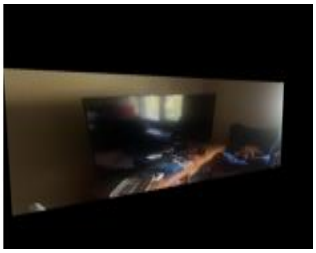
6. Solving this system of equations we get a, b, c, d, e so that we can define the conic
7. We get $A^T A$ and AV from the equations described in the instructions
8. We find the homography matrix as stated in the Two-step case.

5 pairs of perpendicular lines used:



Transformed images:


Observations:

We are getting similar results as we were getting in the 2 steps approach. The results obtained for images 1, 3, 4 follow the expected results. For some reason, the transformed image for image 2 does not follow the expected results. One possible reason is that the angle between the camera and the wall with the windows is very small. Therefore, the imprecision when taking the correct points to define the desired parallel lines is higher. We have tried with different sets of points, and it has not been possible to solve the problem. Since the results with the other methods is the desired one, we consider that we successfully completed the task.

The advantage of this method in comparison with the others is that it is more efficient but the disadvantage is that it requires far more calculations than other methods.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

```
from scipy.linalg import lstsq
from matplotlib.lines import Line2D
import cv2
from scipy.linalg import null_space

#####################
#POINT TO POINT
#####################
def obtain_h(x, x_prima):
    # compute h following the mathematics of the report
    a = [
    [x[0], x[1], 1, 0, 0, 0, -x[0]*x_prima[0], -x[1]*x_prima[0]],
    [0, 0, 0, x[0], x[1], 1, -x[0]*x_prima[1], -x[1]*x_prima[1]],
    [x[2], x[3], 1, 0, 0, 0, -x[2]*x_prima[2], -x[3]*x_prima[2]],
    [0, 0, 0, x[2], x[3], 1, -x[2]*x_prima[3], -x[3]*x_prima[3]],
    [x[4], x[5], 1, 0, 0, 0, -x[4]*x_prima[4], -x[5]*x_prima[4]],
    [0, 0, 0, x[4], x[5], 1, -x[4]*x_prima[5], -x[5]*x_prima[5]],
    [x[6], x[7], 1, 0, 0, 0, -x[6]*x_prima[6], -x[7]*x_prima[6]],
    [0, 0, 0, x[6], x[7], 1, -x[6]*x_prima[7], -x[7]*x_prima[7]]
    ]
    b = x_prima

    a_inverse = np.linalg.inv(a)

    # h = A^{-1}b
    h = np.dot(a_inverse, b)

    #Add known coefficient and reshape
    h = np.append(h, 1)
    h = h.reshape(3, 3)
    return h

def project(img, homography_matrix, new_img_dims=(5000, 5000, -5000, -
5000)):
    # Homography matrix and image dimensions
    H_matrix = np.asarray(homography_matrix)
    original_height, original_width, _ = img.shape
    new_height, new_width, y_offset, x_offset = new_img_dims

    # Create a grid for the new image dimensions
    y_grid, x_grid = np.indices((new_height, new_width))
    y_grid = y_grid + y_offset
    x_grid = x_grid + x_offset

    # Homogeneous coordinates for the new image
    homogeneous_coords = np.vstack([x_grid.ravel(), y_grid.ravel(),
np.ones(x_grid.size)])

    # Apply the inverse homography
```

```
    H_inverse = np.linalg.inv(H_matrix)
    transformed_coords = H_inverse @ homogeneous_coords
    transformed_coords /= transformed_coords[2]

    # Integer coordinates for transformed pixels
    x_transformed = transformed_coords[0].astype(int)
    y_transformed = transformed_coords[1].astype(int)

    # Mask for valid pixel locations within original image bounds
    valid_pixel_mask = (
        (x_transformed >= 0) & (x_transformed < original_width) &
        (y_transformed >= 0) & (y_transformed < original_height)
    )

    # Initialize the transformed image
    transformed_image = np.zeros((new_height, new_width, 3),
dtype=np.uint8)

    # Assign valid transformed pixels to the new image
    valid_y_grid = y_grid.ravel()[valid_pixel_mask] - y_offset
    valid_x_grid = x_grid.ravel()[valid_pixel_mask] - x_offset
    transformed_image[valid_y_grid, valid_x_grid] =
img[y_transformed[valid_pixel_mask], x_transformed[valid_pixel_mask]]

    # Display the image
    plt.imshow(transformed_image)
    plt.axis('off')
    plt.show()

    return transformed_image

#Example for image 1
img = mpimg.imread('HW3_images/board_1.jpeg')

# Coordinates of point P,Q,R,S on Image 1
p = (70,420)
q = (422,1759)
r = (1356,1952)
s = (1222,139)
# Coordinates of point P,Q,R,S on Image 1 (undistorted)
p_prima = (70,420)
q_prima = (70,1804)
s_prima = (1255,420)
r_prima = (1255,1804)

pqrs = [p[0], p[1], q[0], q[1], r[0], r[1], s[0], s[1]]
pqrs_prima1 = [p_prima[0], p_prima[1], q_prima[0], q_prima[1],
r_prima[0], r_prima[1], s_prima[0], s_prima[1]]
h = obtain_h(pqrs, pqrs_prima1)
```

```
print("Homography: \n",h)
img_corrected = project(img,h, new_img_dims=(5000, 5000, -2000, -3000))

#####################
#TWO STEP APPROACH
#####################


def projective_homography(points, image):
    # Extract x and y coordinates from points
    x = [point[0] for point in points]
    y = [point[1] for point in points]

    # Compute lines between pairs of points
    lines = []
    for i in range(0, len(points), 2):  # Pairs of points: (1,2), (3,4),
(5,6), (7,8)
        line = np.cross(points[i], points[i+1]).astype(np.float64)  #
Ensure float64 dtype
        line /= line[2]  # Normalize the line
        lines.append(line)

    # Compute vanishing points by crossing pairs of lines
    v_as = []
    for i in range(0, len(lines), 2):  # Pairs of lines: (1,2), (3,4)
        v_a = np.cross(lines[i], lines[i+1]).astype(np.float64)  # Ensure
float64 dtype
        v_a /= v_a[2]  # Normalize the vanishing point
        v_as.append(v_a)

    # Compute vanishing line by crossing the two vanishing points
    v_l = np.cross(v_as[0], v_as[1]).astype(np.float64)  # Ensure float64
dtype
    v_l /= v_l[2]  # Normalize the vanishing line

    # Construct the projective homography matrix using the vanishing line
    h_p = np.array([
        [1, 0, 0],
        [0, 1, 0],
        [v_l[0], v_l[1], v_l[2]]
    ], dtype=np.float64)  # Ensure float64 dtype

    # Compute the inverse of the homography for distortion correction
    h_p_d = np.linalg.inv(h_p)

    return h_p_d

def affine_homography(points_o,
image,point_pairs,homography_projective_distortion):
```

```
    # Initialize lists for transformed coordinates with direct array
initialization
    x, y = [], []

    # Apply inverse homography and append transformed x and y coordinates
for each point
    for point in points_o:
        transformed_point =
np.dot(np.linalg.inv(homography_projective_distortion),
np.transpose(np.array(point)))
        x.append(transformed_point[0] / transformed_point[2])  #
Normalize by third coordinate
        y.append(transformed_point[1] / transformed_point[2])

    x = [x[0], x[1], x[0], x[2], x[3], x[4], x[3], x[5]]
    y = [y[0], y[1], y[0], y[2], y[3], y[4], y[3], y[5]]

    # Compute the cross products using a list comprehension
    cross_products = [np.cross(p1, p2) for p1, p2 in point_pairs]

    # Unpack the cross products into variables
    l_dash, m_dash, l_dash_dash, m_dash_dash = cross_products

    # Extract coefficients for the line equations
    l1_dash, l2_dash = l_dash[0], l_dash[1]
    m1_dash, m2_dash = m_dash[0], m_dash[1]

    l1_dash_dash, l2_dash_dash = l_dash_dash[0], l_dash_dash[1]
    m1_dash_dash, m2_dash_dash = m_dash_dash[0], m_dash_dash[1]

    # Construct the A matrix using perpendicular line equations
    A = np.array([
        [l1_dash * m1_dash, l1_dash * m2_dash + l2_dash * m1_dash,
l2_dash * m2_dash],
        [l1_dash_dash * m1_dash_dash, l1_dash_dash * m2_dash_dash +
l2_dash_dash * m1_dash_dash, l2_dash_dash * m2_dash_dash]
    ])

    # Find the solution for Ax=0, i.e., the nullspace of A
    s = null_space(A).flatten()  # Ensure s is flattened to a 1D array

    # Normalize s to avoid division by zero
    if s[-1] != 0:
        s = s / s[-1]  # Normalize the last element

    # Construct the S matrix from the solution vector
    if len(s) >= 3:  # Ensure s has at least 3 elements to construct S
        S = np.array([
```

```
            [s[0] / s[2], s[1] / s[2]],
            [s[1] / s[2], 1]
        ])
    else:
        raise ValueError("The null space solution vector is too short to
construct the S matrix.")

    # Perform SVD on the S matrix
    U, Sigma, V_transpose = np.linalg.svd(S)

    # Rebuild the affine transformation matrix A from the singular values
    a_a = np.dot(U, np.dot(np.diag(np.sqrt(Sigma)), V_transpose))

    # Construct the affine homography matrix
    h_a = np.array([
        [a_a[0, 0], a_a[0, 1], 0],
        [a_a[1, 0], a_a[1, 1], 0],
        [0, 0, 1]
    ])

    return h_a

img1 = mpimg.imread('HW3_images/board_1.jpeg')

#projective homography
points =
[(70,420,1),(1222,139,1),(422,1759,1),(1356,1952,1),(422,1759,1),(70,420,
1),(1356,1952,1),(1222,139,1)]
homography_projective_distortion = projective_homography(points,img1)
plt.figure()
project(img1,homography_projective_distortion, new_img_dims=(5000, 5000,
-2000, -2000))

plt.figure()
points_o =
[(544,368,1),(591,583,1),(841,307,1),(895,647,1),(877,536,1),(602,687,1)]
h_a = affine_homography(points_o,img1,points,
                                                    homography_proje
ctive_distortion)


homography_affine_distortion =
np.matmul(homography_projective_distortion,h_a)
project(img1,np.linalg.inv(homography_affine_distortion),
new_img_dims=(5000, 5000, -1500, -3000))

######################
#ONE STEP APPROACH
######################
```

```
def one_step(points, image):
    # points should be a list of 15 3D points.

    # Compute cross products for 5 pairs of perpendicular lines
    lines = []
    for i in range(0, 15, 3):
        l = np.cross(points[i], points[i+1])
        m = np.cross(points[i], points[i+2])
        # Normalize by the third coordinate (homogeneous coordinates)
        l = l / l[2]
        m = m / m[2]
        lines.append((l, m))

    # Construct the A and B matrices
    A = []
    B = []
    for l, m in lines:
        A.append([l[0] * m[0], 0.5 * (l[1] * m[0] + l[0] * m[1]), l[1] *
m[1], 0.5 * (l[2] * m[0] + l[0] * m[2]), 0.5 * (l[2] * m[1] + l[1] *
m[2])])
        B.append([-l[2] * m[2]])

    A = np.matrix(A)
    B = np.matrix(B)

    # Solve for the conic coefficients
    conic_coefficients = np.linalg.inv(A) @ B
    conic_coefficients = conic_coefficients / np.amax(conic_coefficients)

    # Compute the conic matrix
    conic_matrix = np.matrix([[float(conic_coefficients[0]),
float(conic_coefficients[1]) / 2, float(conic_coefficients[3]) / 2],
                              [float(conic_coefficients[1]) / 2,
float(conic_coefficients[2]), float(conic_coefficients[4]) / 2],
                              [float(conic_coefficients[3]) / 2,
float(conic_coefficients[4]) / 2, 1]])

    # Compute A*A^T and Av for later use
    A_A_transpose_matrix = np.matrix([[float(conic_coefficients[0]),
float(conic_coefficients[1]) / 2],
                                      [float(conic_coefficients[1]) / 2,
float(conic_coefficients[2])]])
    A_A_transpose_matrix = A_A_transpose_matrix /
np.amax(A_A_transpose_matrix)
    A_v_matrix = np.matrix([[float(conic_coefficients[3]) / 2],
[float(conic_coefficients[4]) / 2]])

    # Derive A from A*A^T using SVD
```

```
    U_matrix, Sigma_matrix, V_transpose_matrix =
np.linalg.svd(A_A_transpose_matrix)
    A = U_matrix @ np.diag(np.sqrt(Sigma_matrix)) @ V_transpose_matrix

    # Derive v using A and Av
    v_vector = np.linalg.inv(A) @ A_v_matrix

    # Construct the homography matrix
    homography_matrix = np.matrix([[A[0, 0], A[0, 1], 0],
                                   [A[1, 0], A[1, 1], 0],
                                   [float(v_vector[0]),
float(v_vector[1]), 1]])

    return homography_matrix

#Example with first image
img = mpimg.imread('HW3_images/board_1.jpeg')
points = [(978,1335,1), (952,1159,1), (723,1321,1), (519,1474,1),
(748,1492,1), (483,1317,1), (1074,250,1), (730,325,1), (1103,619,1),
(1000,636,1), (1021,851,1), (1103,619,1), (451,386,1), (734,325,1),
(512,701,1)]
plt.figure()
h = one_step(points,img)
project(img, np.linalg.inv(h), new_img_dims=(6000, 6000, 0, -4000))
```