ECE 66100 Homework #3

by

Adrien Dubois (dubois6@purdue.edu)

October 23, 2025

1 Point-to-Point correspondences

It is known that the relationship between a point x in the physical space, and its corresponding pixel x' in the image plane can be written as $x' = Hx$ by representing the two points x and x' using homogeneous coordinates.

Expanding this out, we get:

$$\vec{\boldsymbol{x'}} = \boldsymbol{H}\vec{\boldsymbol{x}}$$

$$\begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{31} \\ h_{21} & h_{22} & h_{32} \\ h_{31} & h_{23} & h_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Which is the same as:

$$x'_1 = h_{11}x_1 + h_{12}x_2 + h_{13}x_3$$

$$x'_2 = h_{21}x_1 + h_{22}x_2 + h_{23}x_3$$

$$x'_3 = h_{31}x_1 + h_{32}x_2 + h_{33}x_3$$

However, since we are measure (x,y) pixel coordinates, we have to re-write this into the physical space context where $x = \frac{x_1}{x_3}$ and $y = \frac{x_2}{x_3}$

So the physical coordinates of the image pixel in terms of the parameters of the general projective homography are:

$$x' = \frac{h_{11}x_1 + h_{12}x_2 + h_{13}x_3}{h_{31}x_1 + h_{32}x_2 + h_{33}x_3} \text{and} y' = \frac{h_{21}x_1 + h_{22}x_2 + h_{23}x_3}{h_{31}x_1 + h_{32}x_2 + h_{33}x_3}$$

Since we are setting $x_3 = 1$ when converting our points from the physical space in $\mathbb{R}^2$ to the homogeneous representational space in $\mathbb{R}^3$, we can divide the RHS of the equation by x_3 and rewrite the above equation just in terms of the physical points:

$$x' = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}} \text{ and } y' = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}}$$

Therefore, each measurement in the physical $\mathbb{R}^2$ space, gives us two equations. Since, h_33 is set equal to 1, we need 4 point correspondences to evaluate the 8 remaining unknowns of the H matrix. The full expansion is as follows:

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1x_1 & -x'_1y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1x_1 & -y'_1y_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x'_2x_2 & -x'_2y_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -y'_2x_2 & -y'_2y_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x'_3x_3 & -x'_3y_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -y'_3x_3 & -y'_3y_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x'_4x_4 & -x'_4y_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -y'_4x_4 & -y'_4y_4 \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{bmatrix}$$

Lastly, to ensure that the mapped image has all pixels values filled in, we will be using the inverse homography and finding points from the target domain that mapped to the points in the source domain.

```python
class Homography():
    def __init__(self):
        pass
    def get_first_six_cols(self, x_points):
        zero_vec = np.zeros(3)
        eqn1_first6cols = np.vstack([np.hstack((point.get_hc(), zero_vec)) for point in x_points])
        eqn2_first6cols = np.vstack([np.hstack((zero_vec, point.get_hc())) for point in x_points])
        return np.vstack([[eqn1, eqn2] for eqn1, eqn2 in zip(eqn1_first6cols, eqn2_first6cols)]) # Stack the rows in an interleaved fashion

    def estimate_projective_homography(self, x_points, x_prime_points):
        # Lets build the first 6 columns of the matrix we are interested in:
        first6cols = self.get_first_six_cols(x_points)

        # Now we only need the last 2 columns
        # x' is in the shape: x1' y1' 1, x2' y2' 1 etc
        # x is: x1 y1 1, x2 y2 1, ...
        # So we take the outer product of the first two elements in the homogeneous form to get the remaining portion:
        last2cols_list = []
        for x, x_prime in zip(x_prime_points, x_points):
            last2cols_list.append(-np.outer(x.get_hc()[:2], x_prime.get_hc()[:2]))
        last2cols = np.vstack(last2cols_list)
        full_matrix = np.hstack((first6cols, last2cols))

        x_prime_matrix = np.vstack([np.vstack((x_prime.get_hc()[0], x_prime.get_hc()[1])) for x_prime in x_prime_points])
        H = np.matmul(np.linalg.inv(full_matrix), x_prime_matrix)
        H = np.vstack((H, np.array(1))).reshape((3,3))
        return H
```

1.1 Resizing the output to fit into a pre-determined range:

One issue with the applying the estimated homographies to images is that they warped image produced may be re-scaled to be outside of the target image plane. This causes issues since you may only display a certain portion of the image, or the image could have been positionally translated enough to be "off-screen". Therefore, it is necessary to apply a resizing homography that scales the image to a chosen width and height, and translates the image back into the image's coordinate frame.

The resizing homography can be calculated through the following steps:

1. Create a 4×3 matrix of the corners of your original image:

$$C = \begin{bmatrix} 0 & w & w & 0 \\ 0 & 0 & h & h \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

2. Calculate C' the transformed coordinates by multiple C by the estimated homography:

$$C' = HC$$

3. Normalize C' by dividing it by its third coordinate if it that coordinate is not 0

4. Find the width and height of the bounding box around the coordinates of C'

$$w_{out} = max_x(C') - min_x(C')$$

$$h_{out} = max_y(C') - min_y(C')$$

5. To translate the top left corner of C' back into the image plane, we can multiple the estimated homography by the following translational homography:

$$H_t = \begin{bmatrix} 1 & 0 & -min_x \\ 0 & 1 & -min_y \\ 0 & 0 & 1 \end{bmatrix}$$

6. On the other hand, the re-scale the transformed image to a user defined width= W_1 and height $= H_1$, we multiple the estimate homography by the following scaling homography:

$$H_s = \begin{bmatrix} \frac{W_1}{w_{out}} & 0 & 0 \\ 0 & \frac{H_1}{h_{out}} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```python
def get_scaling_and_translation_homography(corner_points, homography, target_shape):
    # Create the matrix of corner points
    corner_mat = np.zeros((3, 4))
    for i in range(4):
        corner_mat[:, i] = corner_points[i].hc

    # Apply the homography following C' = HC
    C_prime = homography @ corner_mat

    # C_pn = normalize_corner_points(C_prime)
    C_pn = C_prime / (C_prime[2] + 1e-6)
    y_min, x_min, y_max, x_max = np.min(C_pn[0]), np.min(C_pn[1]), np.max(C_pn[0]), np.max(C_pn[1])

    w_1 = x_max-x_min
    h_1 = y_max-y_min
    H_scale = np.array([[target_shape[0]/w_1, 0, 0], [0, target_shape[1]/h_1, 0], [0,0,1]])

    H_translate = np.array([[1, 0, -x_min], [0, 1, -y_min], [0, 0, 1]])

    return H_translate @ H_scale
```

2 Two-Step Method

2.1 Step 1: Removing Projective Distortion

The purely projective distortion can be estimated by the homography $\boldsymbol{H}$ that takes the vanishing line to $\boldsymbol{l_\infty}$. The line at infinity $\boldsymbol{l_\infty}$ can be calculated through the use of two parallel lines pairs in the physical scene. However, due to projective distortions in the image plane, we can take the same lines in the image space and calculate the vanishing points using the cross product between their pairwise intersections. The vanishing line in the image space would then be the line that intersects both of these points. This vanishing line $\boldsymbol{l} = \begin{bmatrix} l_1 & l_2 & l_3 \end{bmatrix}^T$ can be brought back to infinity by multiplying the camera scene by the following inverse homography:

$$l_\infty = \boldsymbol{H}^{-T}\boldsymbol{l} = \begin{bmatrix} 1 & 0 & -\frac{l_1}{l_3} \\ 0 & 1 & -\frac{l_2}{l_3} \\ 0 & 0 & \frac{1}{l_3} \end{bmatrix} \begin{bmatrix} l_1 \\ l_2 \\ l_3 \end{bmatrix}$$

```python
def get_projective_homography_from_vanishing_line(horiz_lines, vert_lines):
    # Estimate vanihsing points
    vanishing_point1 = horiz_lines[0].get_intersection(horiz_lines[1])
    vanishing_point2 = vert_lines[0].get_intersection(vert_lines[1])

    # Estimate the vanishing line
    vanishing_line = Line(vanishing_point1, vanishing_point2)
    vanishing_line_hc = vanishing_line.hc / vanishing_line.hc[2]

    # Calculate the homography and normalize
    H = np.vstack((np.array([[1, 0, 0], [0, 1, 0]]), vanishing_line_hc))
    return H
```

2.2 Step 2: Removing affine distortion

2.2.1 Core Concept: Dual degenerate conic at infinity

It is known that a point degenerate conic is defined by two straight lines (l, m):

$$C = lm^T + ml^t$$

On the other hand, its dual is created by replacing the lines with points (P, Q):

$$C^* = PQ^T + QP^T$$

We can then create the dual degenerate conic at infinity by finding the intersection between the line at infinity $\boldsymbol{l_\infty}$ and any circle. This will require a pair of circular points.

It is known that the equation of a general conic, in homogeneous coordinates, is:

$$ax_1^2 + bx_1x_2 + cx_2^2 + dx_1x_3 + ex_2x_3 + fx_3^2 = 0$$

To get a circle, we can set $b = 0$ and $c = a = 1$ which gets us:

$$x_1^2 + x_2^2 + dx_1x_3 + ex_2x_3 + fx_3^2 = 0$$

Since it is known that the points at infinity must have the form: $\begin{bmatrix} x_1 & x_2 & 0 \end{bmatrix}^T$, we can conclude that the intersection between $\boldsymbol{l_\infty}$ and any circle must have $x_3 = 0$. Therefore, we get the following constraint on circular points:

$$x_1^2 + x_2^2 = 0$$

Which results in the following points:

$$\vec{\boldsymbol{I}} = \begin{bmatrix} 1 \\ i \\ 0 \end{bmatrix} \text{ and } \vec{\boldsymbol{J}} = \begin{bmatrix} 1 \\ -i \\ 0 \end{bmatrix}$$

Note that these points were calculated by setting $x_1 = 1$; however, since only the ratios matter this arbitrarily set value does not impact the accuracy result.

With the two circular points $\vec{\boldsymbol{I}}$ and $\vec{\boldsymbol{J}}$, we can now get the formula for the dual degenerate conic at infinity $\boldsymbol{C'^*_\infty}$:

$$\boldsymbol{C^*_\infty} = \vec{\boldsymbol{I}}\vec{\boldsymbol{J}}^T + \vec{\boldsymbol{J}}\vec{\boldsymbol{I}}^T$$

$$\boldsymbol{C'^*_\infty} = \begin{bmatrix} 1 & -i & 0 \\ i & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 1 & i & 0 \\ -i & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The important property of the dual degenerate conic at infinity is that it is invariant to a similarity homography. Additionally, you can estimate the dual degenerate conic at infinity from parallel lines. For the purposes of this homework, we will be estimating the dual degenerate conic at infinity by identifying at least 2 lines that are meant to be parallel, but are not in the image due to the affine distortion. This can be achieved through the following process:

2.2.2 Solution:

Lets assume that you have two orthogonal lines in the original scene $\boldsymbol{l}$ and $\boldsymbol{m}$ and that the camera image can be described by the affine homography $\boldsymbol{H}$ such that:

$$l' = H^{-T}l \text{ and } m' = H^{-T}m$$

Then, to calculate the angle between these two lines in terms of the homogeneous representation, we can use the following formula:

$$\cos(\theta) = \frac{l^T\boldsymbol{C^*_\infty}m}{\sqrt{\left(l^T\boldsymbol{C^*_\infty}l\right)\left(m^t\boldsymbol{C^*_\infty}m\right)}}$$

Since it is known that dual conics transform as: $C^{*'} = HC*H^T$, we can rewrite the previous formula in terms of the camera image. Additionally, by picking lines $\boldsymbol{l}$ and $\boldsymbol{m}$ such that they would be orthogonal in the physical scene we can rewrite the formula as such:

$$\cos(\theta) = \cos(\pi/2) = l^T\boldsymbol{H}\boldsymbol{C^*_\infty}\boldsymbol{H}^Tm = 0$$

where $\boldsymbol{H}$ is the affine homography that maps the physical scene to the image plane. The homography can be estimated through just two orthogonal line pairs since, for each line, we would get the following equations:

$$l^T\boldsymbol{H}\boldsymbol{C^*_\infty}\boldsymbol{H}^Tm = 0$$

$$\begin{bmatrix} l'_1 & l'_2 & l'_3 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} a_{11} & a_{21} & 0 \\ a_{12} & a_{22} & 0 \\ t_x & t_y & 1 \end{bmatrix} \begin{bmatrix} m'_1 \\ m'_2 \\ m'_3 \end{bmatrix} = 0$$

$$\begin{bmatrix} l'_1 & l'_2 & l'_3 \end{bmatrix} \begin{bmatrix} a_{11}^2 + a_{12}^2 & a_{11}a_{21} + a_{12}a_{22} & 0 \\ a_{21}a_{11} + a_{22}a_{12} & a_{21}^2 + a_{22}^2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} m'_1 \\ m'_2 \\ m'_3 \end{bmatrix} = 0$$

By letting $S = \boldsymbol{A}\boldsymbol{A}^T$ and knowing that $\boldsymbol{A}\boldsymbol{A}^T$ is symmetric we get:

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

Which finally results in:

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

This equation can be further simplified since all of the information is contained in the rations, which implies that s_{22} can be set to 1.

$$\begin{bmatrix} l'_1m'_1 & (l'_1m'_2 + l'_2m'_1) \end{bmatrix} \begin{bmatrix} s_{11} \\ s_{12} \end{bmatrix} = \begin{bmatrix} -l'_2m'_2 \end{bmatrix}$$

Since this equation has 2 degrees of freedom, two pairs of orthogonal lines would be enough to estimate the parameters of $\boldsymbol{S}$. We these additional lines, we can vectorize the equation above to $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ where:

$$\boldsymbol{S} = x = \left(A^T A\right)^{-1} A^T b = A^T A$$

Lastly, we can use singular value decomposition to extract the homography $\boldsymbol{H}$ (A in the context above) from $\boldsymbol{S}$. Doing so would get us:

$$S = U\Sigma V$$

$$H_{2\times 2} = V\sqrt{\Sigma}V^T$$

$$H = \begin{bmatrix} H_{2\times 2} & \vec{0}^T \\ \vec{0} & 0 \end{bmatrix}$$

```python
def get_purely_affine_homography_from_dual_degen_conic(horiz_lines, vert_lines):
    A = np.zeros((len(horiz_lines), 2))
    b = np.zeros((len(horiz_lines),))

    # Fill in the A and b matrices
    for i, lines in enumerate(zip(horiz_lines, vert_lines)):
        hline = lines[0].hc
        vline = lines[1].hc

        # Normalize line coordinates
        hline = hline / hline[2]
        vline = vline / vline[2]

        A[i] = np.array([hline[0]*vline[0], hline[0]*vline[1] + hline[1]*vline[0]])
        b[i] = -1 * np.array([hline[1] * vline[1]])

    S = (np.linalg.inv((A.T) @ A) @ A.T) @ b
    S = np.array([[S[0], S[1]], [S[1], 1]])
    U,Sigma,V = np.linalg.svd(S)

    H_2by2 = V @ np.diag(np.sqrt(Sigma)) @ V.T
    H = np.vstack((np.hstack((H_2by2, np.array([[0], [0]]))), np.array([0, 0, 0])))
    return H
```

3 One-Step Method

There exists a one-step method that will remove both the projective and affine distortions. In this way, instead of mapping the vanishing lines to the line at infinity $\boldsymbol{l}_{\boldsymbol{\infty}}$, we now map the projection of the dual degenerate conic at infinity $\boldsymbol{C}'^{*}_{\boldsymbol{\infty}}$ to the known $\boldsymbol{C}^{*}_{\boldsymbol{\infty}}$.

It is known from section 2.2.2 that

$$\boldsymbol{C}_{\infty}^{'*} = \boldsymbol{H}\boldsymbol{C}_{\infty}^{*}\boldsymbol{H}^{T}$$

and for two orthogonal lines:

$$\cos(\theta) = \cos(\pi/2) = l^{T}\boldsymbol{H}\boldsymbol{C}_{\infty}^{*}\boldsymbol{H}^{T}m = 0$$

Therefore, since we know that the general equation for a dual conic is:

$$\boldsymbol{C}'^{*} = \begin{bmatrix} a & b/2 & d/2 \\ b/2 & c & e/2 \\ d/2 & e/2 & f \end{bmatrix}$$

Since all of the information is contained in the ratios, we can set $f = 1$ and use user identified lines to estimate the remaining unknowns. Since the formula has 5 degree of freedom (a,b,c,d,e), 5 orthogonal pairs will need to be identified. Each pair would give the following equation:

$$\begin{bmatrix} l'_1m'_1 & l'_2m'_2 & l'_1m'_2 + l'_2m'_1 & l'_1m'_3 + l'_3m'_1 & l'_2m'_3 + l'_3m'_2 \end{bmatrix} \begin{bmatrix} a \\ c \\ b/2 \\ d/2 \\ e/2 \end{bmatrix} = \begin{bmatrix} -l'_3m'_3 \end{bmatrix}$$

Using the five line pairs, we can vectorize the equation above into $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ where:

$$\boldsymbol{C}_{\infty}^{'*} = x = \left(A^{T}A\right)^{-1}A^{T}b$$

We can then apply singular value decomposition on $\boldsymbol{C}_{\infty}^{'*}$ to estimate $\boldsymbol{H}$:

$$\boldsymbol{C}_{\infty}^{'*} = U\boldsymbol{C}_{\infty}^{*}U^{T} = H\boldsymbol{C}_{\infty}^{*}H^{T}$$

```python
def get_one_step_homography(horiz_lines, vert_lines):
    A = np.zeros((len(horiz_lines), 5))
    b = np.zeros((len(horiz_lines),))

    # Fill in the A and b matrices
    for i, lines in enumerate(zip(horiz_lines, vert_lines)):
        hline = lines[0].hc
        vline = lines[1].hc

        # Normalize line coordinates
        hline = hline / hline[2]
        vline = vline / vline[2]

        A[i] = np.array([hline[0]*vline[0], hline[1]*vline[1], hline[0]*vline[1] + hline[1]*vline[0], hline[0]*vline[2] + hline[2]*vline[0], hline[1]*vline[2] + hline[2]*vline[1]])
        b[i] = -1 * np.array([hline[2] * vline[2]])

    S = (np.linalg.inv((A.T) @ A) @ A.T) @ b
    S = np.array([[S[0], S[2], S[3]], [S[2], S[1], S[4]], [S[3], S[4], 1]])
    U,Sigma,V = np.linalg.svd(S)
    return U
```

4 Discussion of results:

The point to point correspondences worked the best for me; however, images such as the corridor were very reliant on choosing seemingly arbitrarily good points. The best performing technique was to pick the points furthest away from one another on the same lines as shown in the results sections. On the other hand, the resizing function that I implemented did not seem to consistently work accurately, as applying many homographies still resulted in all black images. I was able to fix some of these issues by hand, but not significantly enough to be included in this report. Lastly, my two-step and one step methods did not work accurately, even when picking more lines than required to help improve the solution. I was able to test my homographies which seemed to work correctly for specific line pairs, but never produced good results for either method. An example test for the one-step method is included below:

```
for i in range(len(horiz_board_lines)):
    print(f"Checking for line pair {i}: = {horiz_board_lines[i].hc.T @ H_onestep_board @ np.array([[1,0,0],[0,1,0],[0,0,0]]) @ H_onestep_board.T @ vert_board_lines[i].hc}")

# Output:
Checking for line pair 0: = -6.536945311954329e-06
Checking for line pair 1: = 6.132428161076016e-08
Checking for line pair 2: = 2.6170560341887243e-06
Checking for line pair 3: = 6.364545797177263e-08
Checking for line pair 4: = 3.2629004092565786e-07
Checking for line pair 5: = 1.1640604697071216e-07
```

5 Other Core code snippets

Included below are other core pieces of my logic that helped me solve the tasks in this assignment.

5.1 General class for operations with points in homogeneous coordinates

```
class Point():
    def __init__(self, x, y):
        """Defines a point using its physical space coordinates"""
        self.x = x
        self.y = y
        self.hc = self.get_hc()
    @classmethod
    def from_hc(cls, hc):
        """Defines a point from its representation in homogeneous coordinates"""
        if np.isclose(hc[2],0):
            x = hc[0]
            y = hc[1]
        else:
            x = hc[0] / hc[2]
            y = hc[1] / hc[2]
        return cls(x, y)
    def get_hc(self):
        """Returns the point in homogeneous coordinates"""
        return np.array([self.x, self.y, 1])
    def __repr__(self):
        """To string method for debugging"""
        return f"Point(x={self.x}, y={self.y}, hc={self.hc})"
```

5.2 General class for line operations in homogeneous coordinates

```
class Line():
    def __init__(self, point1, point2):
        """Defines a line that passes through 2 points in the physical space"""
        assert isinstance(point1, Point) and isinstance(point2, Point), "A line should be created by 2 Points, or by its angle to the x-axis"
        self.hc = self.get_hc(point1, point2)
    @classmethod
    def from_angle_to_x_axis(cls, angle, y_int):
        """Defines a line by its angle to the x-axis and y intercept"""
        intercept = Point(0, y_int) # Point1 is defined by the y-intercept
        point2 = Point(1, y_int + math.tan(math.radians(angle))) # Point 2 is the point on the line at x=1
        return cls(intercept, point2)
    def get_hc(self, point1, point2):
        """Returns the line in homogeneous coordinates"""
        line = np.cross(point1.hc, point2.hc)
        return line / (line[2] + 1e-6)
    def get_intersection(self, line2):
        intersection_point = np.cross(self.hc, line2.hc)
        return Point.from_hc(intersection_point)
    def __str__(self):
        return f"Line(algebraic={self.hc[1]} * y = {-self.hc[0]} * x + {-self.hc[2]}, hc={self.hc})"
```

5.3 Warp an image by a homography

```python
def reverseMapHomography(source_img, homography, target_size):
    new_image = np.zeros((target_size[0], target_size[1], source_img.shape[2]), dtype=source_img.dtype)
    H_inv = np.linalg.pinv(homography)

    # Create pixel coordinate list for physical space -> HC conversion
    x_range = np.arange(target_size[0])
    y_range = np.arange(target_size[1])
    x_1, x_2 = np.meshgrid(x_range, y_range)
    real_coords = np.vstack([x_2.ravel(), x_1.ravel()]).T
    x_3 = np.ones((real_coords.shape[0],1))
    hc_coord = np.hstack((real_coords, x_3))

    # Get normalized HC coordinates
    src_hc_coords = np.matmul(H_inv, hc_coord.T).T
    norm_mask = ~np.isclose(src_hc_coords[:, 2], 0)
    src_hc_coords[norm_mask] = src_hc_coords[norm_mask] / src_hc_coords[norm_mask, 2][:, np.newaxis]

    # Convert to integer coordinates for image coordinates
    src_x = src_hc_coords[:, 0].astype(int)
    src_y = src_hc_coords[:, 1].astype(int)


    # Check pixel locations vs bounds of the images
    valid_mask = (src_x >= 0) & (src_x < source_img.shape[1]) & (src_y >= 0) & (src_y < source_img.shape[0])
    valid_x = src_x[valid_mask]
    valid_y = src_y[valid_mask]

    # Map valid coordinates from source image to the new image
    new_image[x_1.flatten()[valid_mask], x_2.flatten()[valid_mask]] = source_img[valid_y, valid_x]

    return new_image
```

5.4 Warp one image by a homography and super-impose it on another image

```python
def map_image1_onto_roi_on_image2_inrgb(source_img, target_img, target_roi, homography):
    # This function is used to map one photo onto the ROI in a target image
    # Warp the image following the previously calculated homography
    warped_img = reverseMapHomography(source_img, homography, target_img.shape)

    # Create a mask of the polygon created by the image ROI that I want to cover
    # I will keep the area outside of the mask
    # # Note: cv2 polygons only accept integer pixels:
    polygon_coords = np.array([[int(x), int(y)] for x,y in target_roi], dtype=np.int32)
    mask = np.zeros_like(target_img, dtype=np.uint8)
    cv2.fillPoly(mask, [polygon_coords], (255, 255, 255))

    # Remove the frame portion from the image
    mask_inv = cv2.bitwise_not(mask)
    img_background = cv2.bitwise_and(target_img, mask_inv)

    # Add the warped photo in with projective transformation warping in
    final_image = cv2.add(img_background, cv2.bitwise_and(warped_img, mask))

    # Convert the image to RGB for matplotlib
    final_rgb_image = cv2.cvtColor(final_image, cv2.COLOR_BGR2RGB)
    return final_rgb_image
```

5.5 Example code to run point to point homography estimation

```python
# Define the points
board = [(73.64820002801525, 425.5899285614232), (1221.3719708642668, 139.16325815940627), (1356.5169491525423, 1954.5435635243032), (422.6046365037121, 1795.1935144978288)]
```

```
board_true = [(0,0), (board_img.shape[1], 0), (board_img.shape[1], board_img.shape[0]), (0, board_img.shape[0])]
board_points = [Point(coordinate[0], coordinate[1]) for coordinate in board]
board_true_points = [Point(coordinate[0], coordinate[1]) for coordinate in board_true]

# Get homography Estimate
H_board = Homography().estimate_projective_homography(board_points, corridor_true_points)

# Read the image
source = cv2.imread("/mnt/cloudNAS3/Adubois/Classes/ECE661/HW3/HW3_Images/board_1.jpeg")
target = cv2.imread("/mnt/cloudNAS3/Adubois/Classes/ECE661/HW3/HW3_Images/board_1.jpeg")

# Apply the homography
target_roi = board_true
homography = H_board
H_resize = get_scaling_and_translation_homography(board_points, homography, (target.shape[1], target.shape[0]))
warped_img = map_image1_onto_roi_on_image2_inrgb(source, target, target_roi, H_resize@homography)
```

5.5.1 Example code to run two step homography estimation

```
# Points that define the horizontal and vertical lines on the board
vert_line_coords_board = [(422.93241350329197, 1789.1422468132791), (73.97597702759504, 427.6070177896065), (1356.8447261521223, 1954.5435635243032), (1223.71683709203, 141.18034738758934), (487.6557641126207, 1609.6213055049727), (215.34871830788614, 451.81208852780514), (1129.0901386748844, 1756.8688191623478), (957.6375542793107, 280.35950413223145), (793.8538858617717, 1712.9637223974764), (544.0116145683962, 374.965443074276), (1051.9554057929447, 1807.945081732148), (847.5390020074556, 304.761829652997)]
horiz_line_coords_board = [(149.8692307692304, 464.73076923076906), (1078.3167832167833, 255.27622377622356), (423.77132867132855, 1594.5769230769229), (1213.253846153846, 1683.1923076923076), (73.18058968058949, 425.28308178308134), (1223.5631800631795, 142.23639873639831), (426.9889434889433, 1798.0594945594944), (1356.999473499473, 1953.7351702351702), (203.86923076923063, 744.674825174825), (1108.148951048951, 621.8216783216781), (336.79230769230753, 1244.1433566433566), (1172.5965034965034, 1254.2132867132866)]

# Conver the points into lines:
def get_vert_horiz_lines(vert_line_coords, horiz_line_coords):
    vert_line_points = [Point(coordinate[1], coordinate[0]) for coordinate in vert_line_coords]
    horiz_line_points = [Point(coordinate[1], coordinate[0]) for coordinate in horiz_line_coords]

    vert_lines = [Line(vert_line_points[i], vert_line_points[i+1]) for i in range(0, len(vert_line_points) - 1, 2)]
    horiz_lines = [Line(horiz_line_points[i], horiz_line_points[i+1]) for i in range(0, len(horiz_line_points) - 1, 2)]

    return vert_lines, horiz_lines
vert_board_lines, horiz_board_lines = get_vert_horiz_lines(vert_line_coords_board, horiz_line_coords_board)

# Get homography estimates from orthogonal lines
h_board_proj = get_projective_homography_from_vanishing_line(horiz_board_lines, vert_board_lines)
h_board_aff = get_purely_affine_homography_from_dual_degen_conic(horiz_board_lines, vert_board_lines)


# Apply the estimated homographies onto the board image:
source = cv2.imread("/mnt/cloudNAS3/Adubois/Classes/ECE661/HW3/HW3_Images/board_1.jpeg")
target = cv2.imread("/mnt/cloudNAS3/Adubois/Classes/ECE661/HW3/HW3_Images/board_1.jpeg")
target_roi = board_true

# Step 1:
H_resize = get_scaling_and_translation_homography(board_true_points, h_board_proj, (target.shape[1], target.shape[0]))
warped_img_1 = map_image1_onto_roi_on_image2_inrgb(source, target, target_roi, H_resize@h_board_proj)
```

```
plt.imshow(warped_img_1)
plt.show()

# Step 2
H_resize2 = get_scaling_and_translation_homography(board_true_points, h_board_aff, (
    target.shape[1], target.shape[0]))
warped_img_2 = map_image1_onto_roi_on_image2_inrgb(warped_img_1, target, target_roi,
    H_resize2@h_board_aff)
warped_img_2 = cv2.cvtColor(warped_img_2, cv2.COLOR_BGR2RGB)
plt.imshow(warped_img_2)
plt.show()
```

5.6 Example code to run one step homography estimation

```
# Get one step homography estimate
H_onestep_board = get_one_step_homography(horiz_board_lines, vert_board_lines)

# Open images and load points
source = cv2.imread("/mnt/cloudNAS3/Adubois/Classes/ECE661/HW3/HW3_Images/board_1.jpeg")
target = cv2.imread("/mnt/cloudNAS3/Adubois/Classes/ECE661/HW3/HW3_Images/board_1.jpeg")
target_roi = board_true
homography = H_onestep_board

# Apply the homographies
# It is important to note that I had to apply a 180deg rotational homography to get this
    step to work
H_resize = get_scaling_and_translation_homography(board_points, h_rotate@homography, (
    target.shape[1], target.shape[0]))
img = map_image1_onto_roi_on_image2_inrgb(source, target, target_roi,
    h_rotate@H_resize@homography)
plt.imshow(img)
plt.show()
```

6 Results

6.1 Source images with chosen points

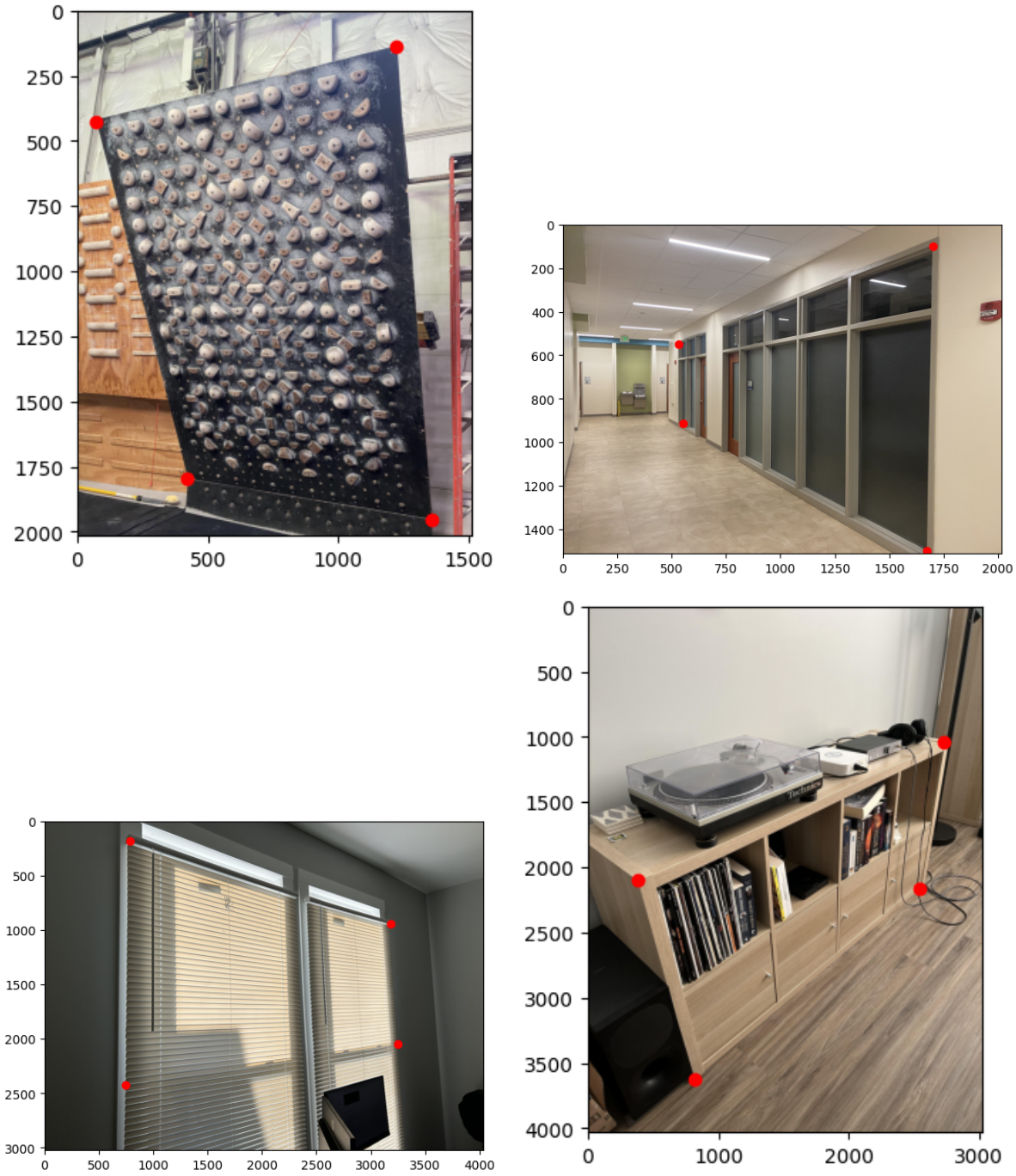

Figure 1: Source images used for this assignment with chosen points. The first two images were given by the instructor, while the following two images were my own. The same image ordering will be used for all results reported.

6.2 Source images with chosen horizontal and vertical lines

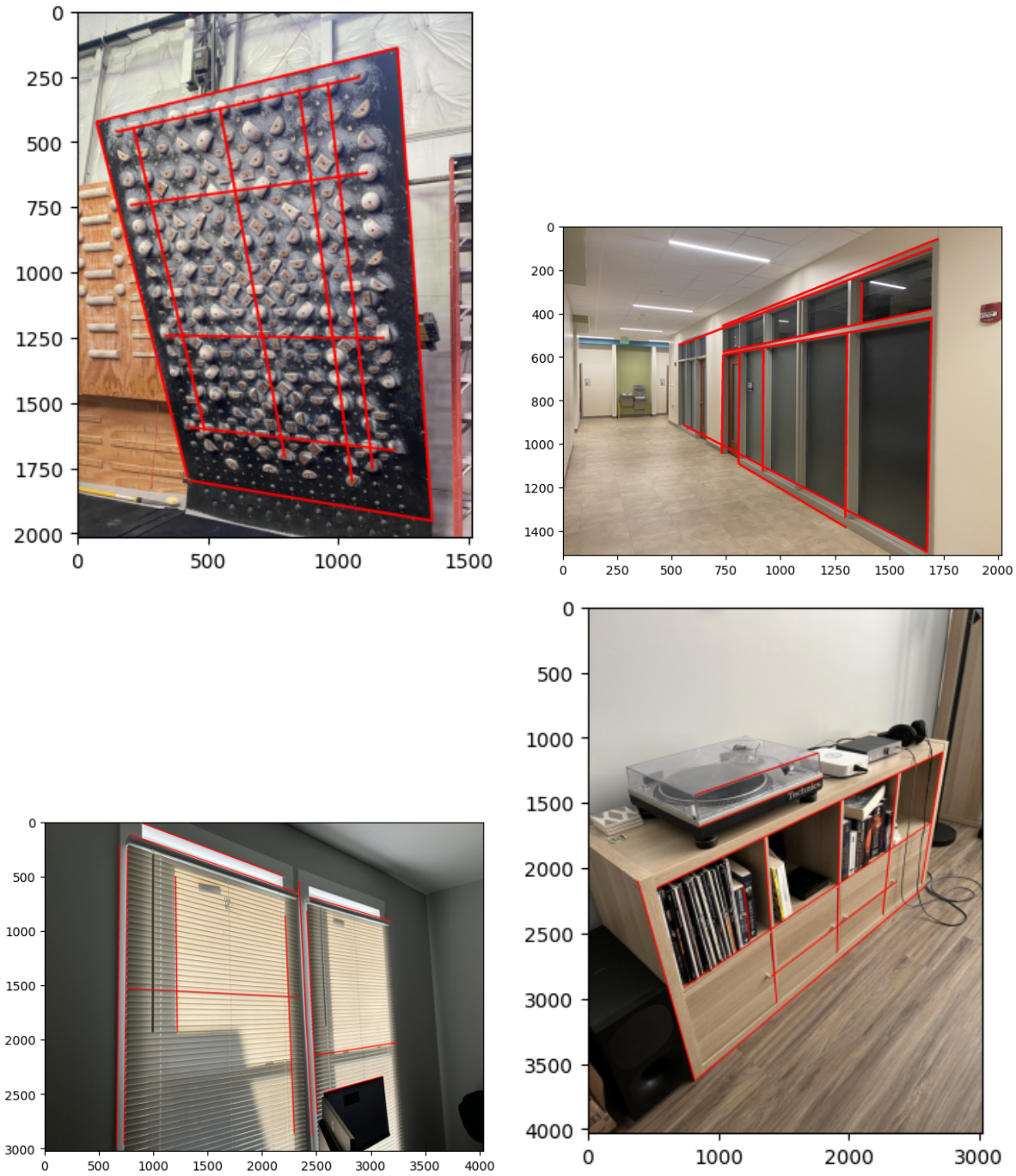

Figure 2: Images with the chosen horizontal and vertical lines used for the one step and two step methods.

6.3 Point to point Correspondences

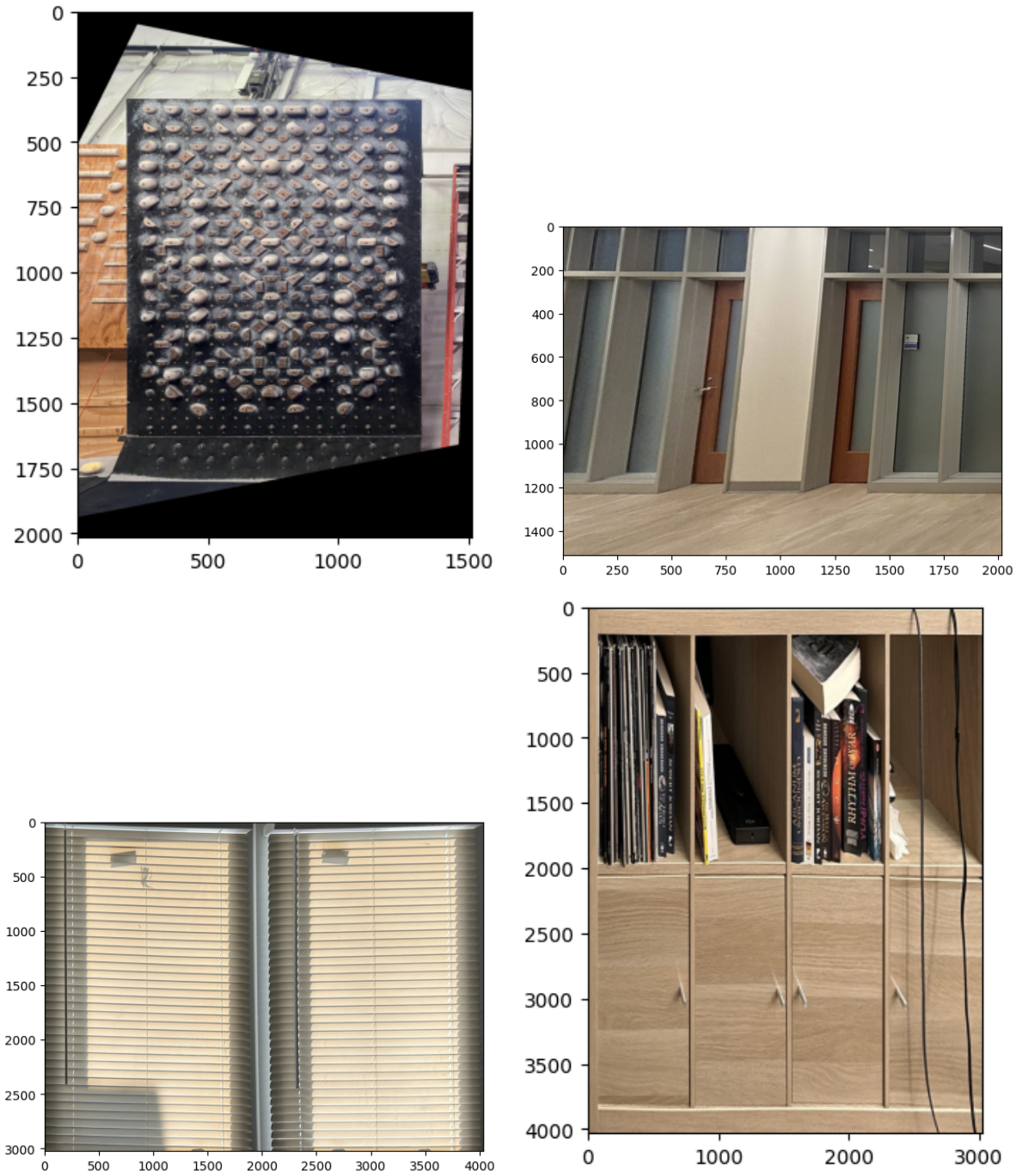

Figure 3: Images after removing the transformation using point to point correspondences.

6.4 Two step method

6.5 Step 1: Removing the purely projective distortions

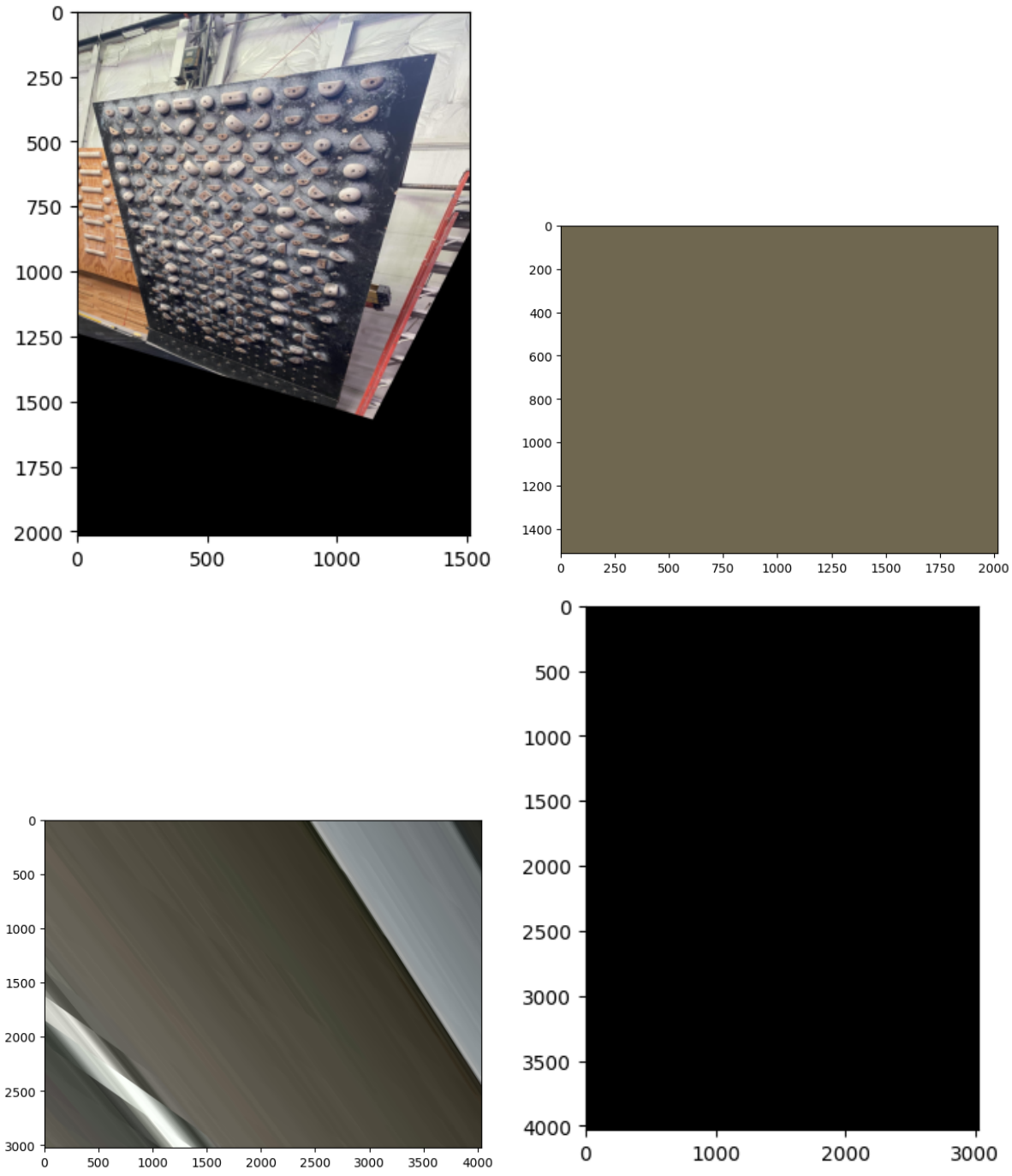

Figure 4: Images after removing the purely projective transformations using the two step method.

6.6 Step 2: Removing the purely affine distortions

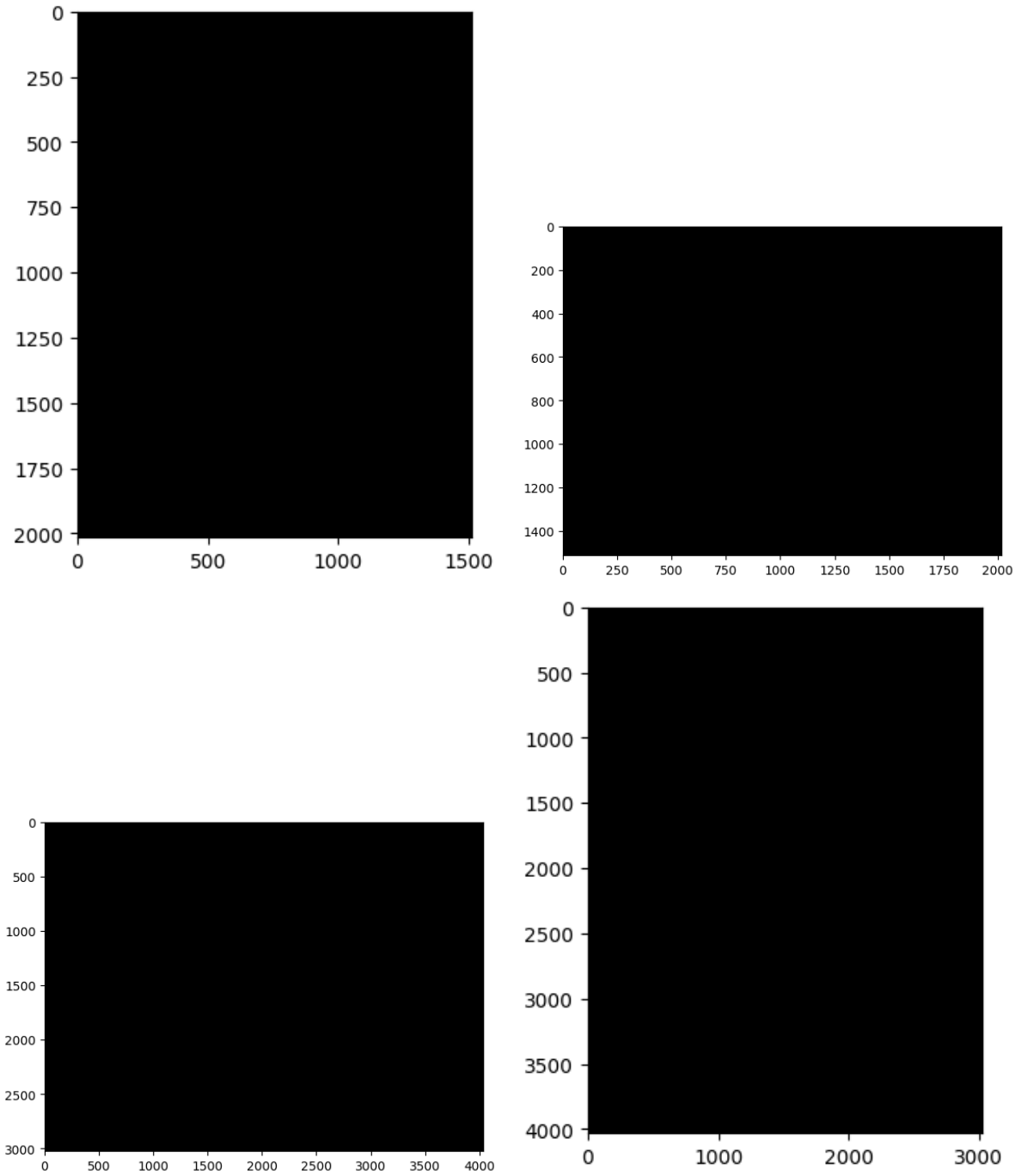

Figure 5: Images after removing the purely affine transformations using the two step method.

6.7 One step method

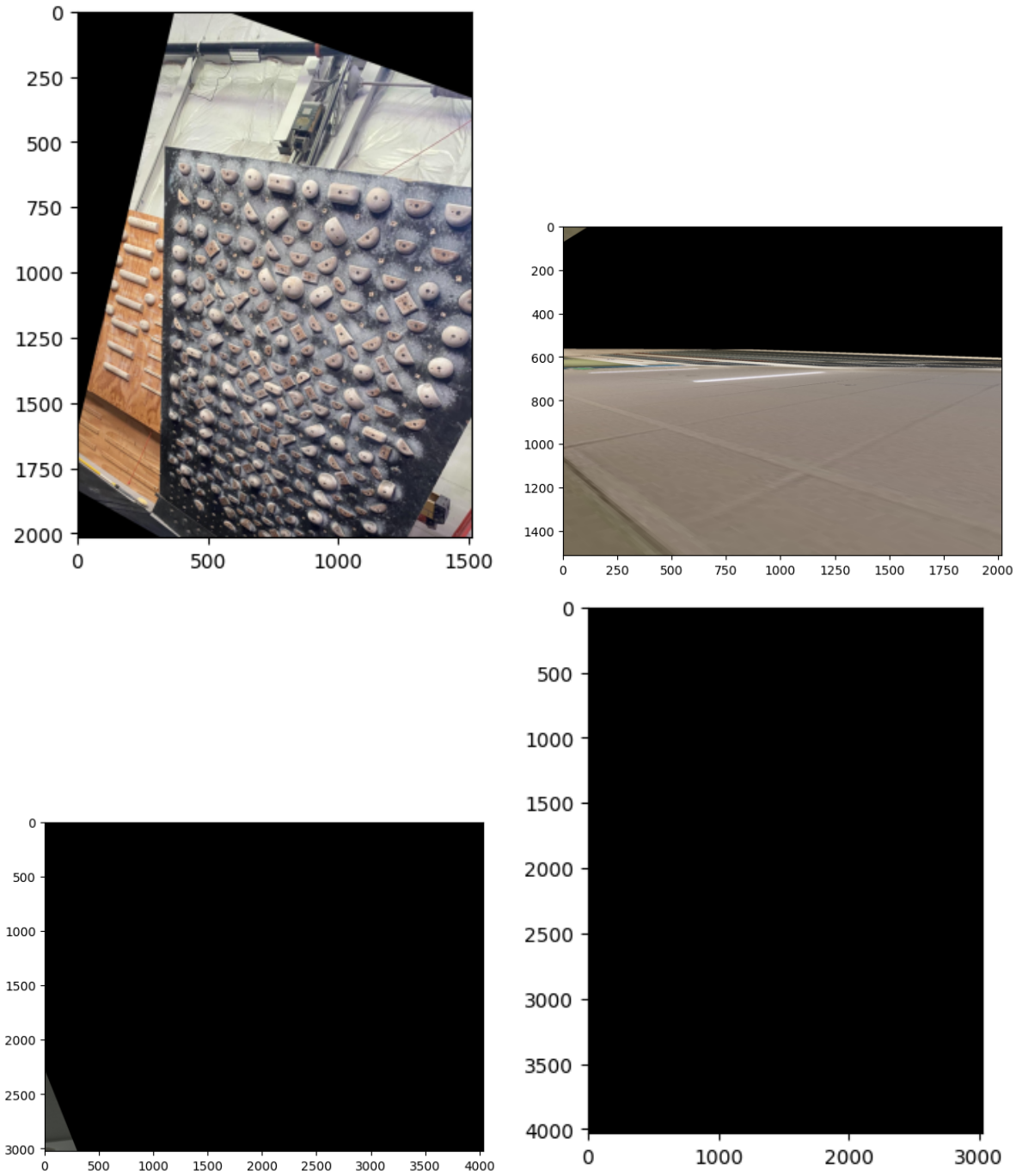

Figure 6: Images after removing the transformation using the one step method.