ECE 66100 Homework #2

by

Adrien Dubois (dubois6@purdue.edu)

October 23, 2025

1 Task 1:

Please note that the points I used to obtain all the homographies are included on the last page of this document in the code printout section. These measurements were taken using a canvas element to record clicks on top of a matplotlib instance of the image.

1.1 Estimate a homography

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:

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1'x_1 & -x_1'y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y_1'x_1 & -y_1'y_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2'x_2 & -x_2'y_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -y_2'x_2 & -y_2'y_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3'x_3 & -x_3'y_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -y_3'x_3 & -y_3'y_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4'x_4 & -x_4'y_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -y_4'x_4 & -y_4'y_4 \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} x_1' \\ y_1' \\ x_2' \\ y_2' \\ x_3' \\ y_3' \\ x_4' \\ y_4' \end{bmatrix}$$

With this homography $\boldsymbol{H}$, we can map points from the image of Alex Honnold onto the picture frame's representational space through the following formula:

$$x'_{frame} = \boldsymbol{H} x_{alex}$$

However, this causes issues due to the scarcity of data in Alex's image: since we would be stretching the photo of Alex through the affine homography, the pixels in the warped image will be sparse with lots of empty spots in between. This can be seen through the following example:

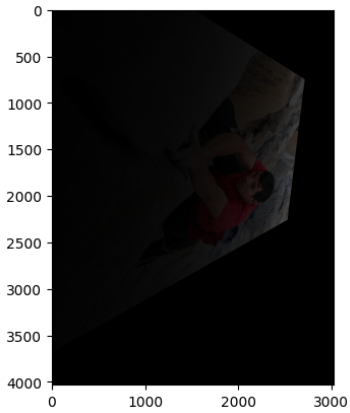

Figure 1: Forward mapping of Alex Honnold onto the image frame using the estimated homography

Therefore, 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.

Lastly, so that only pixels in the frame are replaced, I create a mask of the region of interest in the target domain. Portions within the masked area are replaced by the pixels in the warped source image (Alex Honnold's image in this case), and the background is filled in from the pixels in the target domain.

The following images are the results of that work:

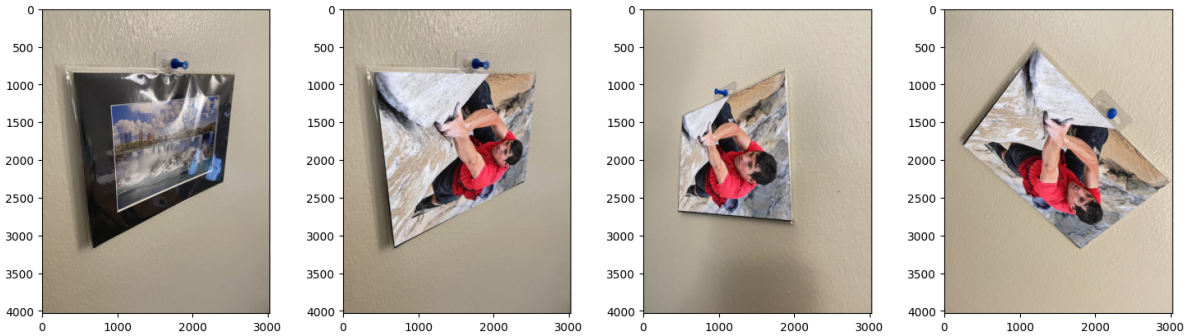

Figure 2: Example images of the original frame photo and the result of mapping Alex Honnold onto those frames through estimation of the projective homography

1.2 Recreating Image 1(c)

Next, by applying the product of the homographies between image 1(a) and 1(b) and image 1(b) and 1(c) to image 1(a), we can recreate image 1(c). The results are shown below:

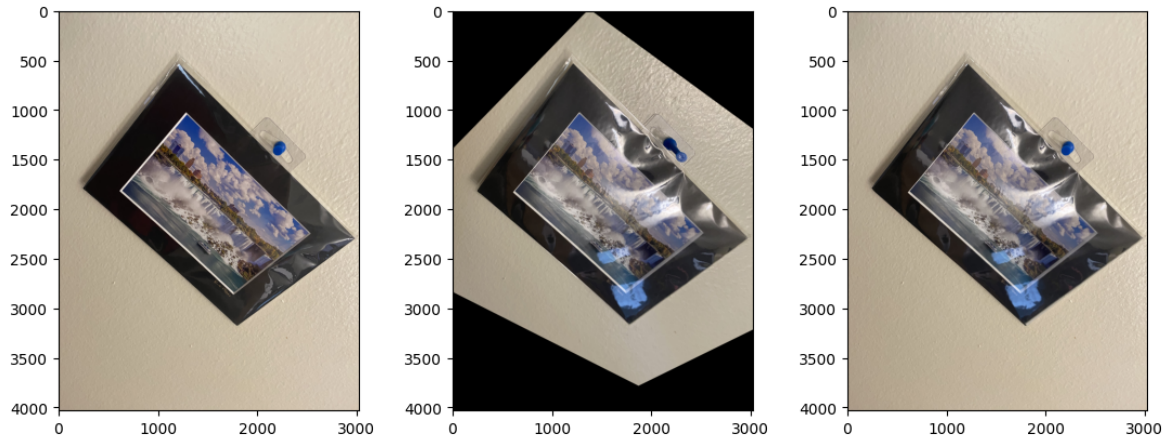

Figure 3: Recreating image 1(c) through a product of homographies. The images are included in the following order: **1)** the original image 1(c), **2)** the warped image with no masking, and **3)** the warped image's ROI masked over image 1(c).

1.3 Mapping Alex Honnold onto the photo-frames with only Affine Homographies

By definition, it is known that the H matrix for a general affine homography will be as follows:

$$\begin{bmatrix} x_1' \\ x_2' \\ x_3' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{13} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Therefore, since this step requires the use of purely affine homographies, we can update the equation in section 1.1 with additional constraints from the now known values for $\begin{bmatrix} h_{31} & h_{32} & h_{33} \end{bmatrix}$. This will result in 6 unknown variables, requiring the use of at least 3 points

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{12} \\ t_x \\ a_{21} \\ a_{22} \\ t_y \end{bmatrix} = \begin{bmatrix} x_1' \\ y_1' \\ x_2' \\ y_2' \\ x_3' \\ y_3' \end{bmatrix}$$

The resulting affine homographies are as follows (rounded to 2 decimal places):

$$H_{alex \to 1(a)} = \begin{bmatrix} 3.72 & -0.31 & 11.46 \\ 0.63 & 2.44 & 269.53 \\ 0. & 0. & 1. \end{bmatrix} H_{alex \to 1(b)} = \begin{bmatrix} 2.43 & 0.11 & 188.32 \\ -0.25 & 3.47 & 762.72 \\ 0. & 0. & 1. \end{bmatrix} H_{alex \to 1(c)} = \begin{bmatrix} 2.63 & -2.09 & 1306.06 \\ 3.49 & 1.48 & -200.31 \\ 0. & 0. & 1. \end{bmatrix}$$

Additionally, my resulting warped images are included below. It is important to note that, since I picked non-rectangular points for the image of Alex Honnold image 1(b) had the best results for the affine homography. This is caused by the fact that affine homographies will keep parallel lines parallel.

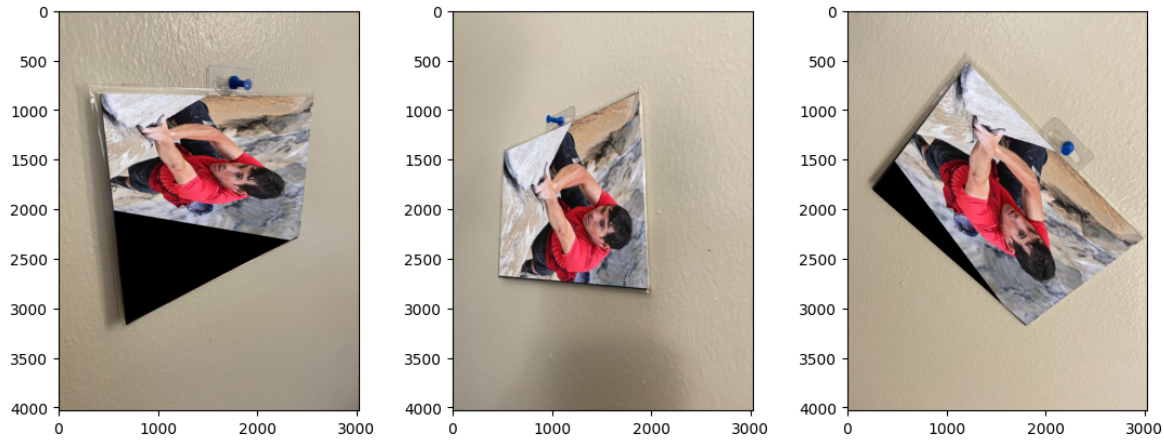

Figure 4: Caption

2 Task 2: Example with my own images

In this task, I took photos of my fuse box from multiple angles and have mapped a photo of a rock-climber climbing El-Capitan onto the fuse box. The results of repeating each step from task 1 are included below:

2.1 Ground truth images

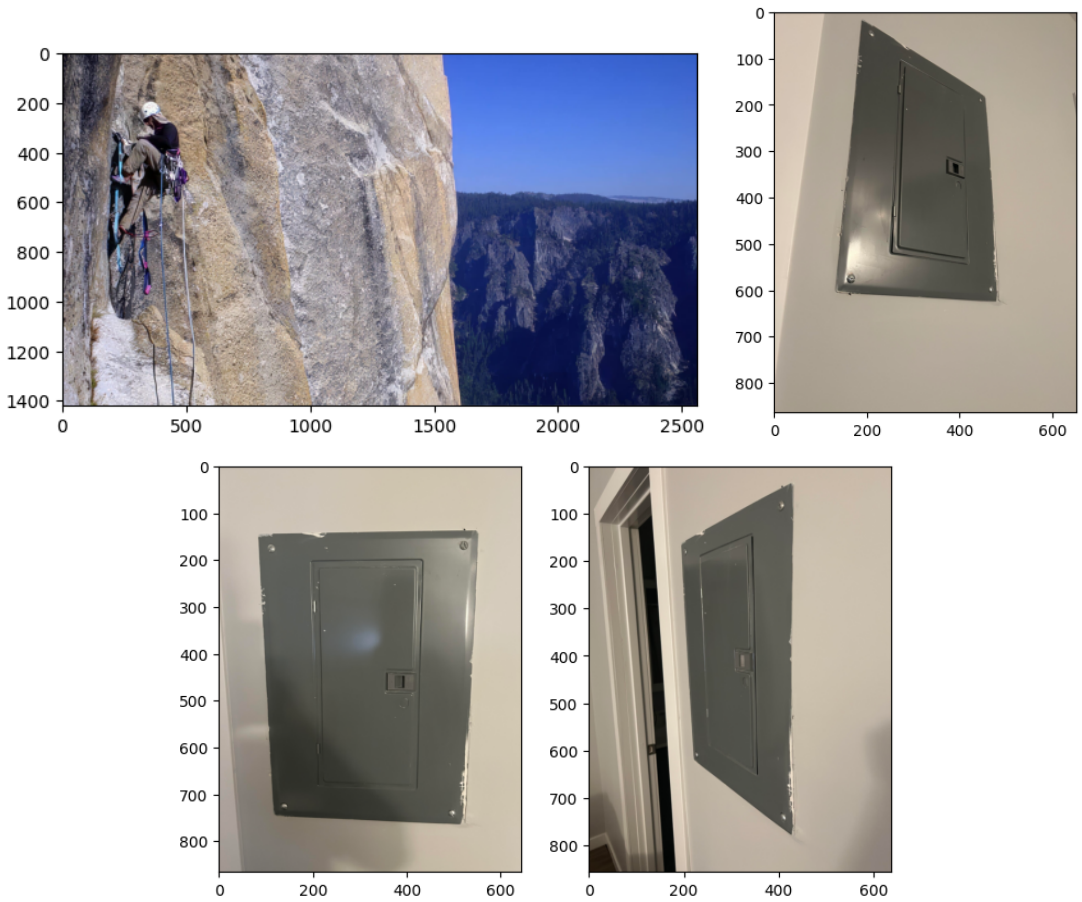

Figure 5: The ground truth images used for task 2. They include a climber on El-Capitan, and multiple views of my fuse-box.

2.2 Mapping using projective homographies

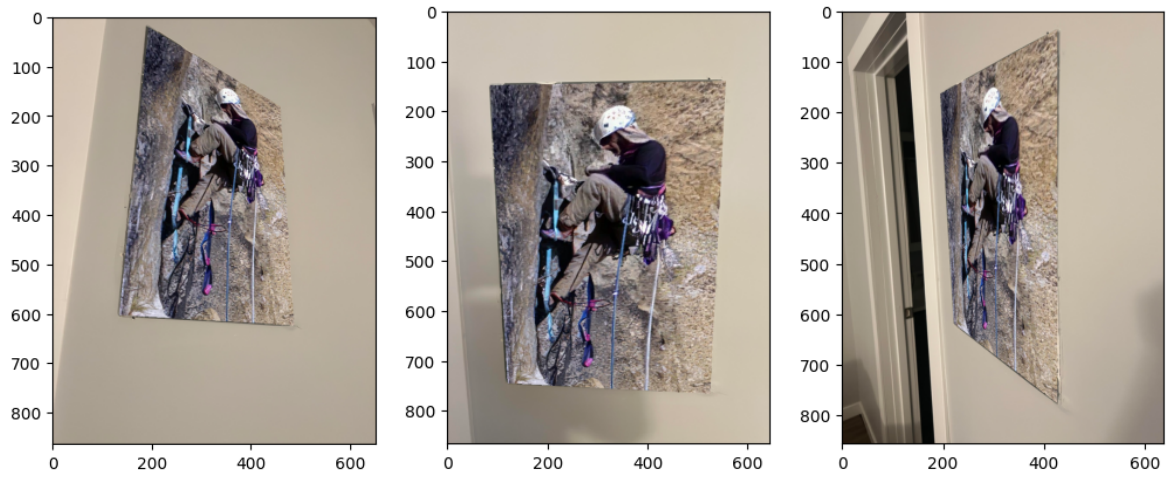

Figure 6: The results of mapping the rock climber onto my fuse-box using projective homographies.

2.3 Mapping homographies between images

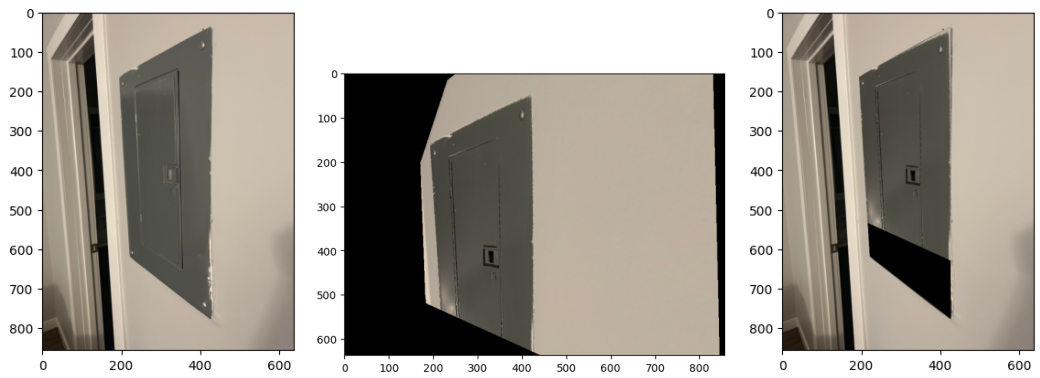

Figure 7: The target image and warped images created by applying the product between the homographies of subsequent ground truth images

It is important to note that since the shift in perspective included in my images was so large, and the angle from the camera to the subject so small, there was some loss when applying these subsequent homographies as the result mapped to an area outside of the image. This issue is shown in the middle image in 7, and resulted in the black area on the right image in the same figure. This issue did not occur when testing my algorithm for more reasonable angle changes as shown in task 1.

2.4 Mapping using only affine homographies

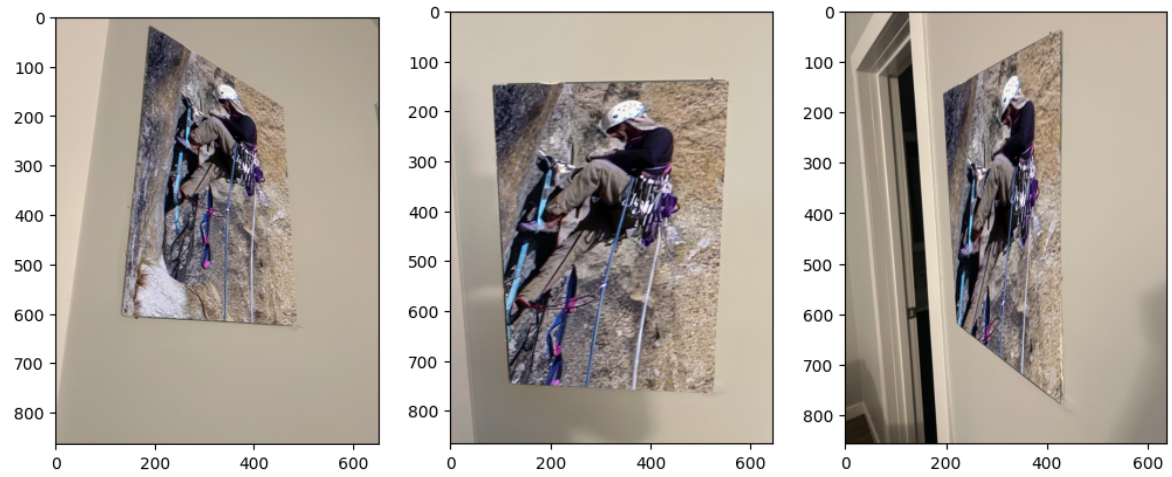

Figure 8: Result of mapping the climber onto my fuse-box using only affine projections.

3 Bonus Section

3.1 Rotational Homography

Since we are only rotating the image, we know that we are dealing with a similarity transformation. Chapter 2.4.2 of Multiple View Geometry by Richard Hartley and Andrew Zisserman gives the following formula for a general similarity transformation:

$$H_{rot} = \begin{bmatrix} s \times cos(\alpha) & -s \times sin(\alpha) & t_x \\ s \times sin(\alpha) & s \times cos(\alpha) & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

Therefore, the homography needed for part one will be a similarity transformation with no scaling factor ($s = 1$), and not translation ($t_x, t_y = 0, 0$).

3.2 Vertical Tilting:

We know that the vertical homography will stretch the coordinates of points along the vertical axis. We also know that it is not translating points: $t_x, t_y = 0, 0$, nor is it scaling them which implies that top-left 2by2 matrix is the identity matrix. I considered this through the use of a unit circle: a point (x,y) will then be stretched by its y-value multiplied by the sine of the angle of rotation. For the homography, this stretching would therefore occur along the x_3 direction for a point $x = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^T$

Therefore the homography for vertical tilting would be:

$$H_{verticle\ tilt} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & sin(\alpha) & 1 \end{bmatrix}$$

3.3 Horizontal Tilting:

The horizontal tilting will keep vertical lines straight, and only affect the direction of horizontal lines. This can again be done by a stretch in the x-axis using the x_3 value. Following the idea from the unit-circle, I will instead apply the cosine of the angle (cosine will give the x-translation of a point by an angle α).

$$H_{horizontal\ tilt} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ cos(\alpha) & 0 & 1 \end{bmatrix}$$

3.4 Bonus Question Results

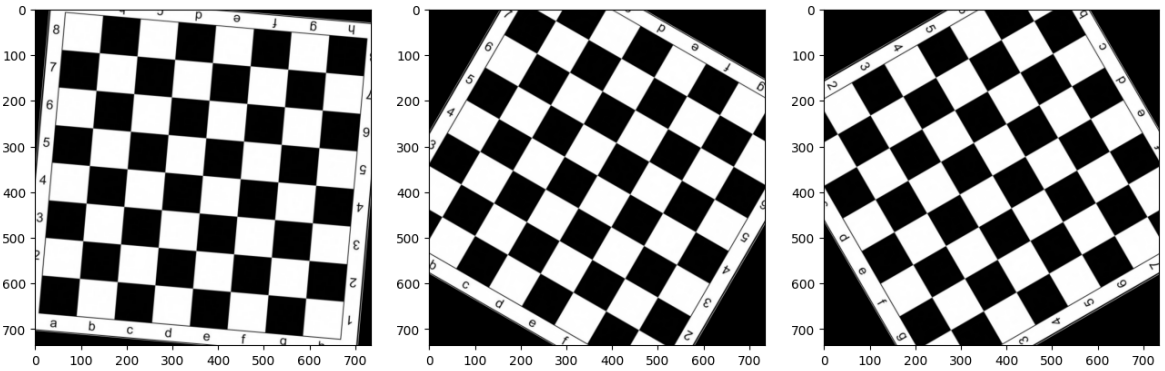

Figure 9: Example images of applying a rotational homography onto an image of a chessboard. The angles used were 5°, 30°and 60°respectively.

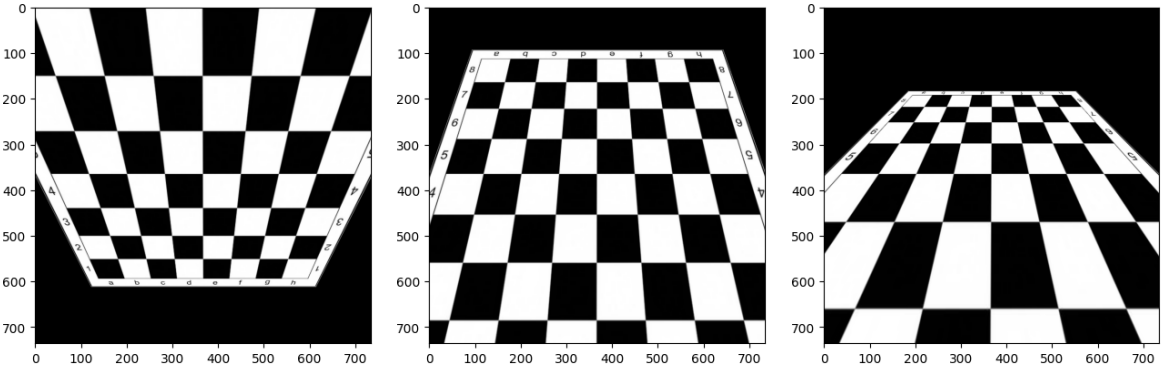

Figure 10: Example images of applying a vertical tilt homography onto an image of a chess board. The angles used were 30°, 200°and 270°respectively.

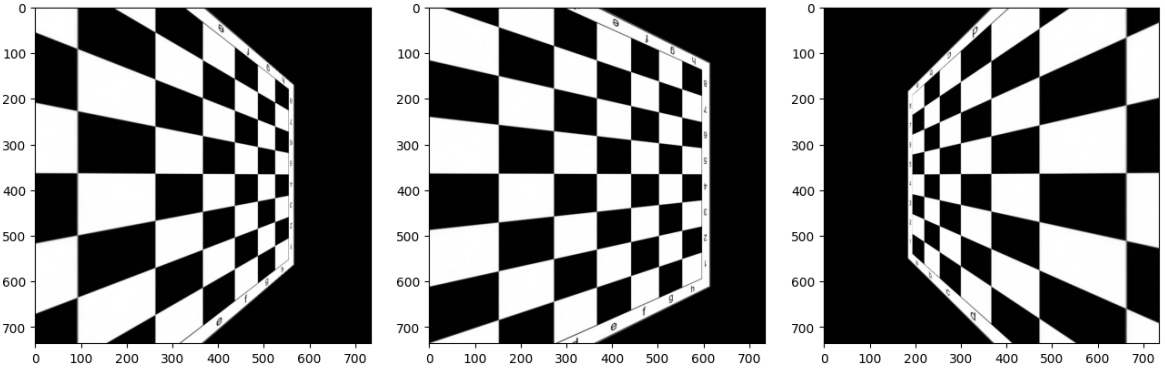

Figure 11: Example images of applying a horizontal tilt homography onto an image of a chess board. The angles used were 30°, 60°and 180°respectively.

3.5 Proof of linear coherence:

It is clear from the images that horizontal tilting will warp the horizontal lines, while keeping vertical lines parallel. On the other hand, vertical tilting will warp the vertical lines and keep horizontal lines parallel. This can be demonstrated by applying the composite homography on some arbitrary horizontal

and vertical lines through the following formula:

$$\vec{l'} = \boldsymbol{H}^{-T}\vec{l}$$

The results of that are included below:

```
Input Horizontal Lines:
[[ 0  0  0]
 [ 5  5  5]
 [-10  0 -10]]
Affect of horizontal tilt on horiztonal lines:
[[  4.96603265   4.99320656   4.96603265]
 [  5.           5.           5.        ]
 [-1837.5       -1837.5      -1837.5     ]]
Input Vertical Lines:
[[ 5  5  5]
 [ 0  0  0]
 [-10  0 -10]]
Affect of horizontal tilt on vertical lines:
[[  9.97282609  10.          9.97282609]
 [  0.          0.          0.        ]
 [-1840.       -1840.      -1840.     ]]
```

```
Input Horizontal Lines:
[[ 0  0  0]
 [ 5  5  5]
 [-10  0 -10]]
Affect of vertical tilt on horiztonal lines:
[[  0.          0.          0.        ]
 [  0.02721088  0.          0.02721088]
 [1817.5       1837.5      1817.5     ]]
Input Vertical Lines:
[[ 5  5  5]
 [ 0  0  0]
 [-10  0 -10]]
Affect of vertical tilt on vertical lines:
[[  5.          5.          5.        ]
 [ -4.97959179 -5.00680268 -4.97959179]
 [1820.        1840.       1820.     ]]
```

Figure 12: The effect on horizontal and vertical lines of applying horizontal tile and vertical tilt respectively.

4 Code Printout

The printout of my code in included in the following pages.

Source code

Important Notice:

To keep the size of the document low, the following code only includes the functions that I implemented and the point measurements. I have included my complete (non-cleaned up) python notebook with all outputs in my submission as well for your convenience.

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

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})"

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

    def estimate_affine_homography(self, x_points, x_prime_points):
        # ONly 3 points are needed
        x_points = x_points[:3]
        x_prime_points = x_prime_points[:3]

        # Get first 6 columns similar to the general projective
homography
        # However, in the case of an affine matrix, we only need those
6 columns
        full_matrix = self.get_first_six_cols(x_points)

        # Create x' matrix: [x_1', y_1', x_2', y_2', ...]
        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)
        last_row = np.array((0, 0, 1)).reshape(-1, 1)
        H = np.vstack((H, last_row)).reshape((3,3))
        return H

    # Used for the bonus. Explanations of the code are included in my
report
    def get_normalizing_homography(self, image):
```

```
        h, w = image.shape[:2]
        print(h, w)
        H_norm = [2/w, 0, -1, 0, 2/h, -1, 0, 0, 1]
        H_norm = np.array(H_norm, dtype=np.float32).reshape((3,3))
        return H_norm
    def get_denormalizing_homography(self, image):
        h, w = image.shape[:2]
        H_norm = [w/2, 0, w/2, 0, h/2, h/2, 0, 0, 1]
        H_norm = np.array(H_norm, dtype=np.float32).reshape((3,3))
        return H_norm
    def get_rotational_homography(self, alpha):
        H_rot = [math.cos(alpha), -math.sin(alpha), 0,
math.sin(alpha), math.cos(alpha), 0, 0, 0, 1]
        H_rot = np.array(H_rot, dtype=np.float32).reshape((3,3))
        return H_rot
    def get_horizontal_tilt_homography(self, alpha):
        H = [1,0,0,0,1,0, math.cos(alpha),0,1]
        H = np.array(H, dtype=np.float32).reshape((3,3))
        return H
    def get_vertical_tilt_homography(self, alpha):
        H = [1,0,0,0,1,0,0,math.sin(alpha), 1]
        H = np.array(H, dtype=np.float32).reshape((3,3))
        return H

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.inv(homography)

    # cv2 is in (y,x) format
    for i in range(target_size[0]):
        for j in range(target_size[1]):
            target_hccoord = np.array((j,i,1))

            # Find the corresponding coordinate from the image I want
to map over
            src_hccoord = np.matmul(H_inv, target_hccoord)
            src_hccoord = (src_hccoord/src_hccoord[2]).astype(int) #
ensure x_3 = 1

            if 0 <= src_hccoord[0] < source_img.shape[1] and 0 <=
src_hccoord[1] < source_img.shape[0]:
                new_image[i,j] = source_img[src_hccoord[1],
src_hccoord[0]]
    return new_image

def mapHomography(source_img, homography, target_size):
    new_image = np.zeros((target_size[0], target_size[1],
source_img.shape[2]), dtype=source_img.dtype)
```

```
    for i in range(source_img.shape[0]):
        for j in range(source_img.shape[1]):
            curr_hc_coord = np.array((j, i, 1))

            new_hc_coord = np.dot(homography, curr_hc_coord)
            new_hc_coord = (new_hc_coord/new_hc_coord[2]).astype(int)
# Ensure x_3 = 1

            # Check x and y bounds:
            if 0 <= new_hc_coord[0] < target_size[1] and 0 <=
new_hc_coord[1] < target_size[0]:
                new_image[new_hc_coord[1]][new_hc_coord[0]] =
source_img[i][j]

    return new_image

def map_image1_onto_roi_on_image2_inrgb(source_img, target_img,
target_roi, homography):
    # This function is used to map the photo of Alex onto the ROI in
the frame-photo
    # Warp the alex image following the previously calculated
homography
    warped_alex = reverseMapHomography(source_img, homography,
target_img.shape)

    # Create a mask of the polygon created by the image frame 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 Alex photo in with projective transformation warping in
    final_image = cv2.add(img_background, cv2.bitwise_and(warped_alex,
mask))

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

Point measurements for given images

```
img1_pixel_coords = [(420.590909090909, 841.6818181818182),
(2558.772727272728, 874.409090909091), (2416.954545454546,
2270.7727272727275), (682.409090909091, 3154.4090909090914)]
img2_pixel_coords = [(518.772727272727, 1430.7727272727275),
(1871.5000000000005, 830.7727272727275), (1958.7727272727275,
2805.318181818182), (464.2272727272725, 2674.409090909091)]
img3_pixel_coords = [(1216.9545454545455, 538.4090909090905),
(2973.318181818182, 2305.681818181818), (1806.045454545455,
3178.409090909091), (256.9545454545455, 1814.772727272727)]
alex_pixel_coords = [(126.51136363636363, 201.73484848484838),
(689.6742424242425, 68.59090909090901), (698.6704545454545,
638.9507575757575), (133.70833333333331, 629.9545454545455)]
```

Point measurements for my images

```
myimg1_pixel_coords = [(86.69264069264068, 150.39393939393938),
(555.3073593073593, 141.02164502164499), (524.8474025974026,
766.6222943722944), (121.83874458874459, 745.5346320346321)]
myimg2_pixel_coords = [(194.09090909090907, 23.811688311688158),
(458.24675324675337, 182.77272727272714), (486.29870129870136,
624.590909090909), (135.64935064935065, 603.551948051948)]
myimg3_pixel_coords = [(31.458874458874448, 368.29978354978357),
(603.168831168831, 431.56277056277054), (389.94913419913416,
787.7099567099567), (47.86038961038963, 799.4253246753248)]
myimg4_pixel_coords = [(195.10768398268402, 165.05681818181802),
(426.9799783549784, 37.52705627705609), (426.9799783549784,
777.1996753246754), (222.93235930735938, 619.5265151515152)]
el_cap_pixel_coords = [(76.9193548387097, 114.38709677419342),
(546.5967741935484, 181.48387096774195), (670.4677419354838,
1027.9354838709678), (92.40322580645164, 1089.8709677419356)]
```