CSC420 Assignment3

Student name: Yilin Qu

Student number: 1002995734

Utorid: quyilin

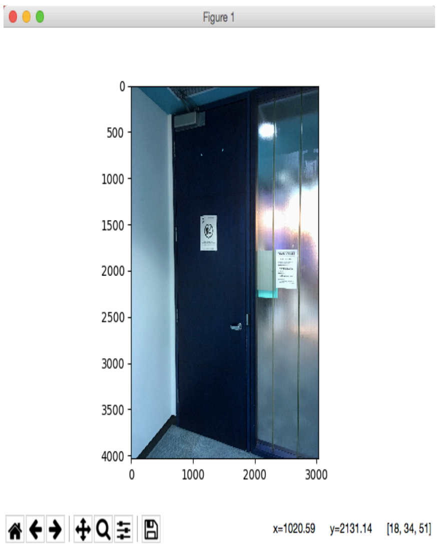
Part A

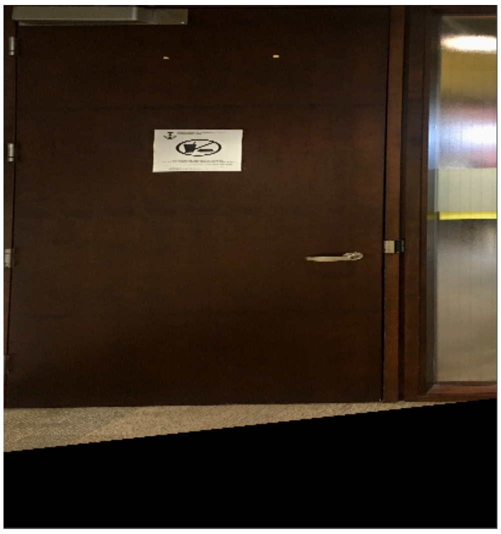
```
import numpy as np

import matplotlib

matplotlib.use('TkAgg')

import matplotlib.pyplot as plt

import cv2 as cv


img = cv.imread("door.jpg")

plt.imshow(img, cmap="gray")

plt.show()


# plot four corners on the door

x1, y1 = 671.5, 320.7

x3, y3 = 769.8, 3625.68

x2, y2 = 1904.13, 112.95

x4, y4 = 1926.05, 3952.95


M, N = img.shape[0], img.shape[1]


x_1, y_1 = 0, 0

x_2, y_2 = M-1, 0

x_3, y_3 = 0, N-1

x_4, y_4 = M-1, N-1


img = cv.imread("door.jpg")


A = np.array([[x1, y1, 1, 0, 0, 0, -x_1 * x1, -x_1 * y1, -x_1],

             [0, 0, 0, x1, y1, 1, -y_1 * x1, -y_1 * y1, -y_1],
             [x2, y2, 1, 0, 0, 0, -x_2 * x2, -x_2 * y2, -x_2],

             [0, 0, 0, x2, y2, 1, -y_2 * x2, -y_2 * y2, -y_2],


             [x3, y3, 1, 0, 0, 0, -x_3 * x3, -x_3 * y3, -x_3],

             [0, 0, 0, x3, y3, 1, -y_3 * x3, -y_3 * y3, -y_3],


             [x4, y4, 1, 0, 0, 0, -x_4 * x4, -x_4 * y4, -x_4],

             [0, 0, 0, x4, y4, 1, -y_4 * x4, -y_4 * y4, -y_4]])


# use numpy method to get h


ATA = np.matmul(A.T, A)

eigenvalues, eigenvector = np.linalg.eig(ATA)

h = eigenvector[:, np.argmin(eigenvalues)]

H = h.reshape((3, 3))


img1 = cv.warpPerspective(img, H, (5000, 4000))


plt.imshow(img1), plt.show()
```

Idea:

Stick the paper on the door and make the horizontal and vertical edges between the paper and the door parallel.

First we find four corners of the **door** on our captured image, and use these four points as the parameters to get the projective image. To get **h,** I first pass in four pairs of points so that I can solve the matrix equation to get h11-h22, and then us this **h** matrix to perform a warp operation. Then on the projective image, find four corner points of the paper and four corner points of the door, then calculate relative pixel-distance horizontal and vertical distance. Since we know the **actual size of paper** as well as the **ratio** of the pixel-distance between the paper and the door, we can use these two numbers to calculate the **actual size of the door.**, since homography does not change the ratio between parallel lines.





The left image is the picture captured at first, the right image is the image after homography. The parameters are as follows:

- **top left corner of the door: 248.418, 252.747**
- **bot left corner of the door: 248.418, 3099.07**
- **top right corner of the door: 3917.25, 252.747**
- **bot right corner of the door: 3917.25, 3099.07**
- **top left corner of the paper: 1590.41, 1042.79**
- **bot left corner of the paper: 1590.41, 1345.82**
- **top right corner of the paper: 2477.85, 1042.79**
- **bot right corner of the paper: 2477.85, 1345.82**

Using these data, we can calculate:

height of the door is:

297mm * (3099.07 – 252.747) / (1345.82 – 1042.79) = **2789.68396 mm**

width of the door is:

210mm * (3917.25 – 248.418) / (2477.85 – 1590.41) = **868.176688 mm**

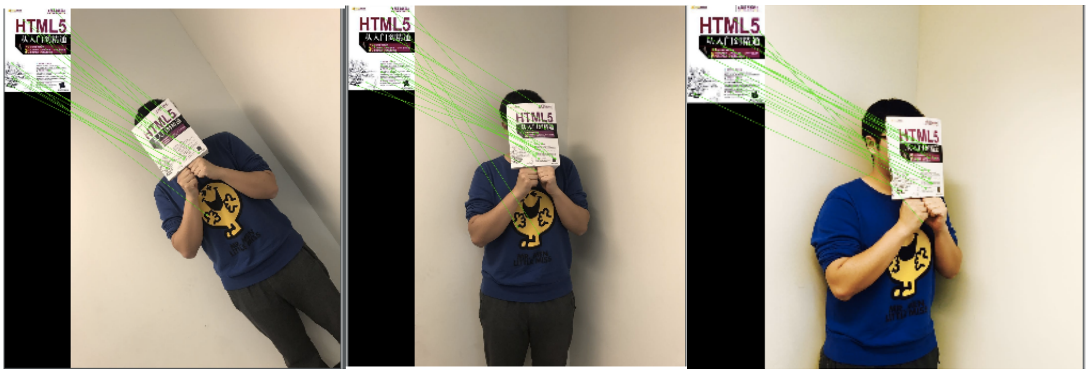
Part B

(a)

open source from:

```
# open source implementation and modification from following link:
#
https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_feature_homograph
y/py_feature_homography.html
```

```
def find_match(threshold):
    MIN_MATCH_COUNT = 200
    img2 = cv.imread('whitebook.jpg')          # queryImage
    img1 = cv.imread('cover.jpg') # trainImage
    # img1 = cv2.imread('img2.jpg', 0) # trainImage
    # img1 = cv2.imread('img3.jpg',0) # trainImage
    # Initiate SIFT detector
    sift = cv.xfeatures2d_SIFT.create()
    # find the keypoints and descriptors with SIFT
    kp1, des1 = sift.detectAndCompute(img1,None)
    kp2, des2 = sift.detectAndCompute(img2,None)
    FLANN_INDEX_KDTREE = 0
    index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
    search_params = dict(checks = 50)
    flann = cv.FlannBasedMatcher(index_params, search_params)
    matches = flann.knnMatch(des1,des2,k=2)
    # store all the good matches as per Lowe's ratio test.
    good = []
    for m,n in matches:
        if m.distance < threshold*n.distance:
            good.append(m)
    draw_params = dict(matchColor = (0,255,0), singlePointColor = None, flags = 2)
    img3 = cv.drawMatches(img1,kp1,img2,kp2,good,None,**draw_params)
    cv.imwrite("q2_a.jpg", img3)
    # plt.imshow(img3), plt.show()

    return kp1, kp2, good
```



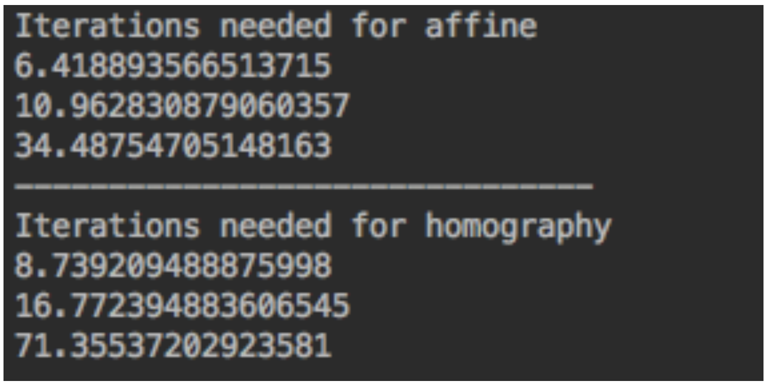
	img1	img2	img3
outlier =	4 / 20	6 / 20	10 / 20

(Some of the matches are right on the book but the relative positions do not match, so I count these as outliers as well)

(b)

```
def Q2_b_affine(P, p):

    return (np.log(1 - P)) / (np.log(1 - pow(p, 3)))


def Q2_b_homography(P, p):

    return (np.log(1 - P)) / (np.log(1 - pow(p, 4)))
```

Here P is 0.99, p is the visually estimated rate of inliers, and in affine transformation we need at least 3 pairs of keypoints, while in homography we need at least 4 pairs(according to the derivation of the transformation matrix).

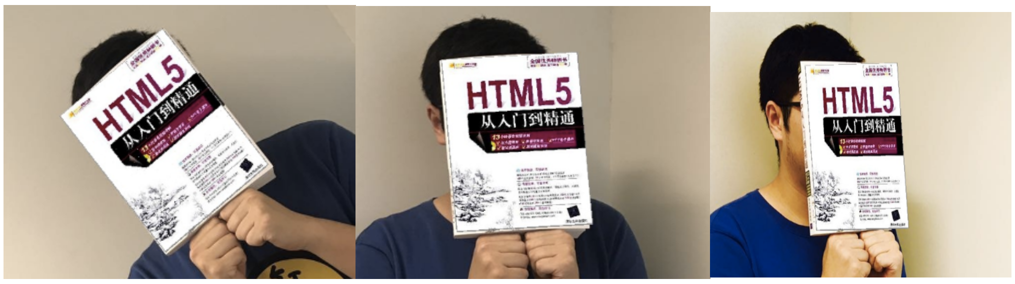


```
Iterations needed for affine
6.418893566513715
10.962830879060357
34.48754705148163
--------------------------------
Iterations needed for homography
8.739209488875998
16.772394883606545
71.35537202923581
```

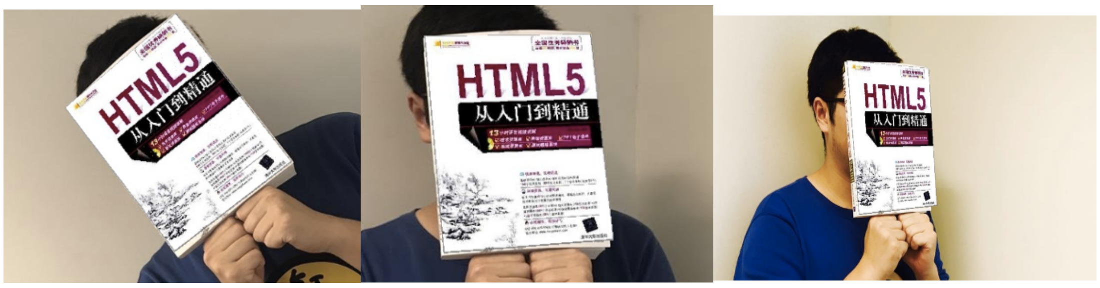
(c) and (d)

```
def ransac_transformation(mode, img1, img2, outimage):
    kp1, kp2, good = find_match(img2, 0.55)

    img2 = cv.imread(img2)          # queryImage
    img1 = cv.imread(img1)                      # trainImage

    src = np.array([kp1[m.queryIdx].pt for m in good])
    dst = np.array([kp2[m.trainIdx].pt for m in good])
    if mode == "affine":
        model_robust, inliers = sk.measure.ransac((src, dst),
AffineTransform, min_samples=5, residual_threshold=2,
                                          max_trials=100)
        h = model_robust.params
        np.delete(h, 2, axis=0)
        for i in range(img1.shape[0]):
            for j in range(img1.shape[1]):
                src = np.array([[j],
                               [i],
                               [1]])
                dst = np.matmul(h, src)

                y = int(dst[0, 0])
                x = int(dst[1, 0])

                img2[x, y] = img1[i, j]
        cv.imwrite(outimage, img2)

    elif mode == "homography":
        model_robust, inliers = sk.measure.ransac((src, dst),
ProjectiveTransform, min_samples=5, residual_threshold=2,
                                              max_trials=100)
        h = model_robust.params
        for i in range(img1.shape[0]):
            for j in range(img1.shape[1]):
                src = np.array([[j],
                               [i],
                               [1]])
                dst = np.matmul(h, src)

                y = int((dst[0, 0] / dst[2, 0]))
                x = int((dst[1, 0] / dst[2, 0]))

                img2[x, y] = img1[i, j]
        cv.imwrite(outimage, img2)
    else:
        return False
```

Mode == "affine":



Mode == "homography":



In practice my algorithm will fail when:

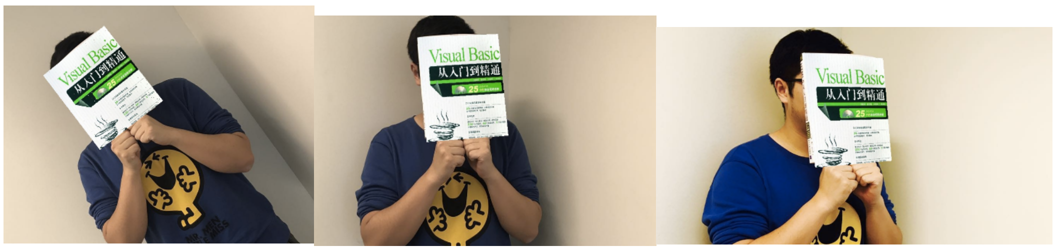
- **I change the parameter "min_sample" to be less than 3 for affine or less than 4 for homography**
- **Sometimes there might be some "NaN" values so I cannot round them**
- **When I try some other images, it will fail when there are some other "corners" or "edges", for example when I put the book near a black**

- **board, the corners and edges of the black board will do affect the matches, so that cannot recover a good transformation.**
- **In the above case, sometimes it will cause the index to be out of bound, which is also a failure mode.**
- **I change the lighting condition too much.**
- **If I warp the book too much then affine will not be accurate.**

Comparing affine and homography:

- **Affine transformation will preserve the planar property**
- **Homography will map a projective plane**
- **In the above results homography are more accurate than affine.**

(e)



This looks a little bit wired because the size of another book cover is not perfectly the same as the previous book cover. Also, different keypoints should be detected. The positions for the book are mapped almost correctly.

Part C

Method1: camera calibration.

Taking a picture in a fixed length(d = 50cm) without out-of-plane rotation and record six different in our real-world coordinates, and then pass these six points to a matrix A, then find the null-space of A and compute the q-r factorization to find K. **(But this is probably wrong because it does not give the correct K matrix, another method is below:)**

```
def find_K_2d_to_3d():
    img = cv.imread("q3.jpg")
    plt.imshow(img), plt.show()
    M, N = img.shape[0], img.shape[1]
    x1, y1, X1, Y1, Z1 = 79.75, 281.57, -150 / M, 120 / N, 500
    x2, y2, X2, Y2, Z2 = 114.8, 565.9, -150 / M, 0, 500
    x3, y3, X3, Y3, Z3 = 149.9, 854.3, -150 / M, -120 / N, 500
    x4, y4, X4, Y4, Z4 = 527.8, 262.1, 0, 120 / N, 500
    x5, y5, X5, Y5, Z5 = 531.7, 581.5, 0, 0, 500
    x6, y6, X6, Y6, Z6 = 539.5, 854.3, 0, -120 / N, 500

    A = np.array([[X1, Y1, Z1, 1, 0, 0, 0, 0, -x1*X1, -x1*Y1, -x1*Z1, -x1],
                 [0, 0, 0, 0, X1, Y1, Z1, 1, -y1*X1, -y1*Y1, -y1*Z1, -y1],

                 [X2, Y2, Z2, 1, 0, 0, 0, 0, -x2 * X2, -x2 * Y2, -x2 * Z2, -x2],
                 [0, 0, 0, 0, X2, Y2, Z2, 1, -y2 * X2, -y2 * Y2, -y2 * Z2, -y2],

                 [X3, Y3, Z3, 1, 0, 0, 0, 0, -x3 * X3, -x3 * Y3, -x3 * Z3, -x3],
                 [0, 0, 0, 0, X3, Y3, Z3, 1, -y3 * X3, -y3 * Y3, -y3 * Z3, -y3],

                 [X4, Y4, Z4, 1, 0, 0, 0, 0, -x4 * X4, -x4 * Y4, -x4 * Z4, -x4],
                 [0, 0, 0, 0, X4, Y4, Z4, 1, -y4 * X4, -y4 * Y4, -y4 * Z4, -y4],

                 [X5, Y5, Z5, 1, 0, 0, 0, 0, -x5 * X5, -x5 * Y5, -x5 * Z5, -x5],
                [0, 0, 0, 0, X5, Y5, Z5, 1, -y5 * X5, -y5 * Y5, -y5 * Z5, -y5],


                [X6, Y6, Z6, 1, 0, 0, 0, 0, -x6 * X6, -x6 * Y6, -x6 * Z6, -x6],
                [0, 0, 0, 0, X6, Y6, Z6, 1, -y6 * X6, -y6 * Y6, -y6 * Z6, -y6]])


    ATA = np.matmul(A.transpose(), A)
    v = np.linalg.svd(ATA)[2].T[-1].reshape((3, 4))
    v = v[0:3, 0:3]
    result = np.linalg.qr(v)[1]
    print(result)
```

Method2 : Using "vanishing point" method, find 3 vanishing points and record the coordinates. Here I zoom in the image and show its 3 vanishing points. They are the cross points of each two green lines.

Derivations:

For example:
We have three matrix equations

$$V_1^T w V_2 = 0$$
$$V_1^T w V_3 = 0$$
$$V_3^T w V_2 = 0$$

where $w = \begin{bmatrix} w1 \\ w2 \\ w3 \\ w4 \end{bmatrix}$

and V_1, V_2, V_3 are vanishing points

Solve this equation system to have w':

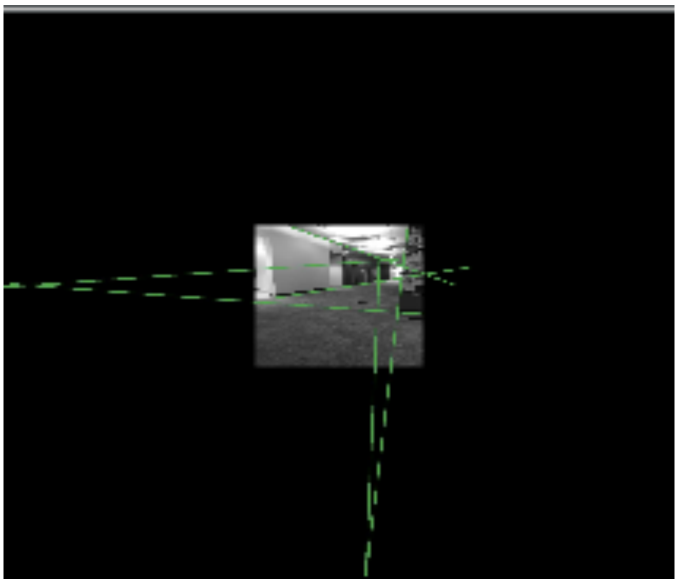
$$\begin{bmatrix} w_1 & 0 & w_2 \\ 0 & w_1 & w_3 \\ w_2 & w_3 & w_4 \end{bmatrix}$$

Then K is given by:

$$K = inv(chol(w'))$$

where "chol" is the cholesky transformation and inv is the inverse

It might be necessary to normalize K according to the $K[2, 2]$ value as well.

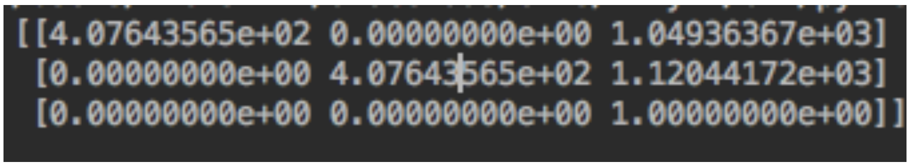


```
    def find_K_vanishing_point():
    # img = cv.imread("vanishingPoint.jpg")
```

```
# plt.imshow(img), plt.show()

x1, y1, z1 = 79.364, 962.338, 1

x2, y2, z2 = 1253.1, 921.516, 1

x3, y3, z3 = 1079.49, 1986.65, 1


# x1, y1, z1 = 212, 2138, 1

# x2, y2, z2 = -49, 42, 1

# x3, y3, z3 = 1105, 146, 1


A = np.array([[x1*x2 + y1*y2, x2*z1 + x1*z2, z1*y2 + y1*z2, z1*z2],

             [x1*x3 + y1*y3, x3*z1 + x1*z3, z3*y1 + y3*z1, z1*z3],

             [x2*x3 + y2*y3, x3*z2 + x2*z3, z2*y3 + z3*y2, z2*z3]])

# print(A)

ATA = np.matmul(A.T, A)

#print(ATA)

# find null space of A

eigenvalues, eigenvector = np.linalg.eig(ATA)

h = eigenvector[:, np.argmin(eigenvalues)]

# print(h)

h = np.linalg.svd(A)[2][-1]

#print(h)

w1 = h[0]

w2 = h[1]

w3 = h[2]

w4 = h[3]


#print(w1, w2, w3, w4)

w = np.array([[w1, 0, w2],

             [0, w1, w3],

             [w2, w3, w4]])

# print(w)

# K = inv(chol(W))

K = np.linalg.inv(np.linalg.cholesky(w).T)
```

```
print(K / K[2, 2])
```

And the K matrix I got for iPhone7 is as follows:



```
[[4.07643565e+02 0.00000000e+00 1.04936367e+03]
 [0.00000000e+00 4.07643565e+02 1.12044172e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
```

Part D