Computer Vision HW 3

Jiyoon Park (jiyoonp)

November 2022

1 Representing the World with Visual Words

1.1 Planar Homographies as a Warp

We know that

$$\lambda x_1 = P_1 X$$
$$\lambda x_2 = P_2 X$$

We rewrite this as

$$P_1^{-1} \lambda x_1 = X$$
$$P_2^{-1} \lambda x_2 = X$$

$$P_1^{-1} x_1 \equiv P_2^{-1} x_2$$
$$x_1 \equiv P_1 P_2^{-1} x_2$$
$$H \equiv P_1 P_2^{-1}$$

We can see that there exists H that satisfies the equation

$$x_1 \equiv H x_2$$

1.2 The Direct Linear Transform

1) h has 8 degrees of freedom

2) h needs 4 point pairs (each point has x, y)

3)

$$x_1^i H x_2^i = 0$$

$$x_1^i = \begin{bmatrix} x_1^i \\ y_1^i \\ 1 \end{bmatrix}$$

$$x_2^i = \begin{bmatrix} x_2^i \\ y_2^i \\ 1 \end{bmatrix}$$

We rewrite H to be

$$H = \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix}$$

Then we can write

$$\begin{bmatrix} \lambda x_1 \\ \lambda y_1 \\ \lambda \end{bmatrix} = H \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

If we write out the equations,

$$h_{11}x_2 + h_{12}y_2 + h_{13} = (h_{31}x_2 + h_{32}y_2 + h_{33})x_1$$

$$h_{21}x_2 + h_{22}y_2 + h_{23} = (h_{31}x_2 + h_{32}y_2 + h_{33})y_1$$

To put into the form of:

$$A_i h = 0$$

We get,

$$A_i = \begin{bmatrix} -x_2^i & -y_2^i & -1 & 0 & 0 & 0 & x_1^i x_2^i & x_1^i y_2^i & x_1^i \\ 0 & 0 & 0 & -x_2^i & -y_2^i & -1 & y_1^i x_2^i & y_1^i y_2^i & y_1^i \end{bmatrix}$$

Where

$$h = \begin{bmatrix} h_{11} & h_{12} & h_{13} & h_{21} & h_{22} & h_{23} & h_{31} & h_{32} & h_{33} \end{bmatrix}^T$$

4) The trivial solution would be when h = 0. The matrix a is not full rank. It is rank(A) = 8. Since it is a 8x9 matrix, it is not full rank.

$$A^T A$$

is a 9 x 9 matrix. Since the rank(A) = 8, it means that the matrix will have zero eigenvalue with a corresponding eigen vector which we can use for h.

1.3 Using Matrix Decompositions to calculate the homography

1.4 Theory Questions

Q 1.4.1

We have

$$x_1 = K_1[I\ 0]X$$
$$x_2 = K_2[R\ 0]X$$

We can rewrite this as:

$$[I\ 0]^{-1}K_1^{-1}x_1 = X$$
$$[R\ 0]^{-1}K_2^{-1}x_2 = X$$

$$[I\ 0]^{-1}K_1^{-1}x_1 = [R\ 0]^{-1}K_2^{-1}x_2$$

$$x_1 = K_1[I\ 0][R\ 0]^{-1}K_2^{-1}x_2$$
$$x_1 = K_1R^{-1}K_2^{-1}x_2$$

So we can say that

$$H = K_1R^{-1}K_2^{-1}$$

Q 1.4.2

We know that

$$x_1 = Hx_2$$

and

$$H = K_1 R^{-1} k_2^{-1}$$

So,

$$H^2 = (K_1 R^{-1} k_2^{-1})(K_1 R^{-1} k_2^{-1})$$

Since K is constant, we can write

$$K_1, K_2$$

as K. Then,

$$H^2 = K R^{-1} R^{-1} K$$

We have the rotation matrix R,

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R^{-1} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R^{-1}R^{-1} = \begin{bmatrix} \cos(\theta)^2 - \sin(\theta)^2 & \cos(\theta)\sin(\theta) + \sin(\theta)\cos(\theta) & 0 \\ -\sin(\theta)\cos(\theta) - \cos(\theta)\sin(\theta) & -\sin(\theta)^2 + \cos(\theta)^2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Which brings us back to,

$$H^2 = K \begin{bmatrix} \cos(2\theta) & \sin(2\theta) & 0 \\ -\sin(2\theta) & \cos(2\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} K$$

which is the same as rotating it by 2θ.

Q 1.4.3

For planar homography to exist, it should satisfy one of two conditions. The points in the 3D world should lie on a plane or there is pure rotation between two views. If neither of the conditions are met, then there is no planar homography which means that planar homography is not sufficient to map any arbitrary scene to another viewpoint.

Q 1.4.4

Let

$$x = PX$$

If there are 3 points in the 3D world,

$$X_1, X_2, X_3$$

that are in a line, we can write

$$X_1 - X_2 = k(X_2 - X_3)$$

If we apply projection to the three points, we have

$$P(X_1 - X_2) = P(k(X_2 - X_3))$$

$$P(X_1) - P(X_2) = Pk(X_2) - Pk(X_3)$$

$$x_1 - x_2 = kx_2 - kx_3$$

$$x_1 - x_2 = k(x_2 - x_3)$$

Which also represents a line on the 2D projected points. Thus lines are preserved.

2 Computing Planar Homographies

2.1 Feature Detection and Matching

Q 2.1.1

The Harris Corner Detector looks at the horizontal and vertical derivatives of the image and look for areas that have high values for both. It looks at the window around a pixel to see if that pixel is unique. The FAST detector looks at 16 pixels around a pixel and sees if enough continuous pixels are not within range (in terms of intensity). To make it even faster, it checks four points first to see if it is worth searching more for. The computation of the FAST detector is very fast compared to the Harris Corner detector since it only checks four points in the beginning (at most only checks 16 pixels) and also because it does not calculate the gradient of the image.

Q 2.1.2

The BRIEF Descriptor converts image patches (the results of feature detectors) into binary vectors to represent the patch. Then compares the binary values of different patches to get the similarity between the patches. The descriptors in our lecture did so for the entire image (got the image orientation and did something with that as well). Which could be valuable if we wanted the descriptors to be rotation invariant or other cool features. However, compared to the descriptors we saw in class that does a lot of computation to find the description, the BRIEF Descriptor is much faster. We could use the ones we saw in class, but it would be a bit slow.

Q 2.1.3

The Hamming distance compares binary strings of equal length and returns the number of bit positions that have different bits(xor operation). The nearest Neighbour measures the distribution of points over a space and returns the distances in that space and is often used for classification and regression. They can both be used to match the interest points as they both give distances to one another. We use the Hamming Distance because it is faster than calculating the euclidean distance (just a bit-wise xor computation).

Q 2.1.4

```python
def matchPics(I1, I2, opts):
    """
    Match features across images

    Input
    -----
    I1, I2: Source images
    opts: Command line args

    Returns
    -------
    matches: List of indices of matched features across I1, I2 [p x 2]
    locs1, locs2: Pixel coordinates of matches [N x 2]
    """

    ratio = opts.ratio  # 'ratio for BRIEF feature descriptor'
    sigma = opts.sigma  # 'threshold for corner detection using FAST feature
    detector'

    # TODO: Convert Images to GrayScale
    # I1 = cv2.cvtColor(I1, cv2.COLOR_BGR2GRAY)
    # I2 = cv2.cvtColor(I2, cv2.COLOR_BGR2GRAY)
    I1 = skimage.color.rgb2gray(I1)
    I2 = skimage.color.rgb2gray(I2)

    # TODO: Detect Features in Both Images
    locs1 = corner_detection(I1, sigma)
    locs2 = corner_detection(I2, sigma)

    # TODO: Obtain descriptors for the computed feature locations
    desc1, locs1 = computeBrief(I1, locs1)
    desc2, locs2 = computeBrief(I2, locs2)

    # TODO: Match features using the descriptors
    matches = briefMatch(desc1, desc2, ratio)

    return matches, locs1, locs2
```

Q 2.1.5

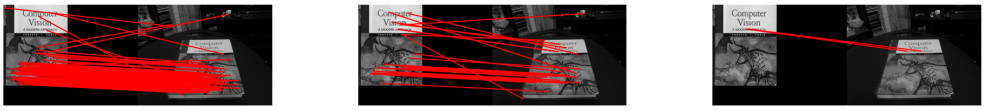


Figure 1: (a) 0.05 & 0.7(b)0.15 & 0.7(c)0.5 & 0.7

When I fix the ratio and vary the sigma, in theory I am changing the values I am giving to the threshold for corner detector using the FAST feature detector. Seeing by the way that detector works, if I increase the sigma, then it would have a harsher bar for something to be detected as a corner. So it will output less corners. And this is indeed what we see in Figure 1.

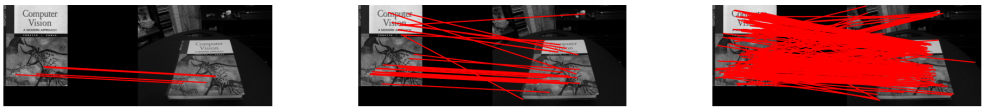
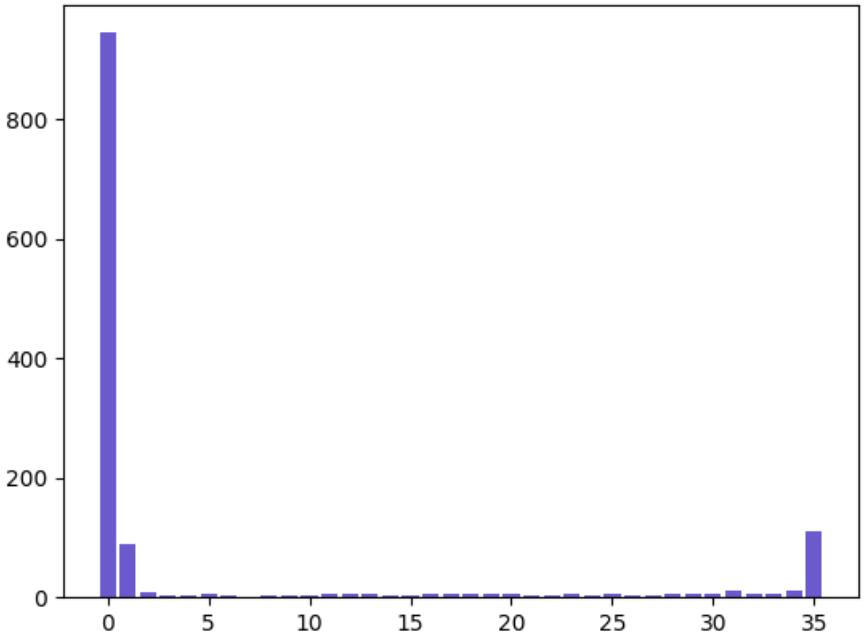


Figure 2: (a) 0.15 & 0.5(b)0.15 & 0.7(c)0.15 & 1.0

When I fix the sigma and vary the ratio, I am changing the ratio that is given to the BRIEF feature descriptor. If I give a bigger ratio, then it means Maximum ratio of distances between first and second closest descriptor in the second set of descriptors will increase. It will return more matches and that is what I am seeing as my result.

Q 2.1.6



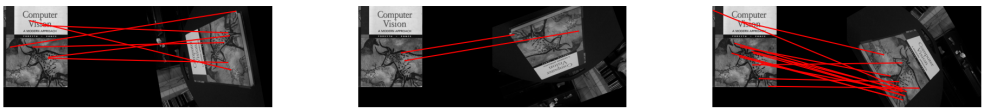


Figure 3: (a) 0.15 & 0.5(b)0.15 & 0.7(c)0.15 & 1.0

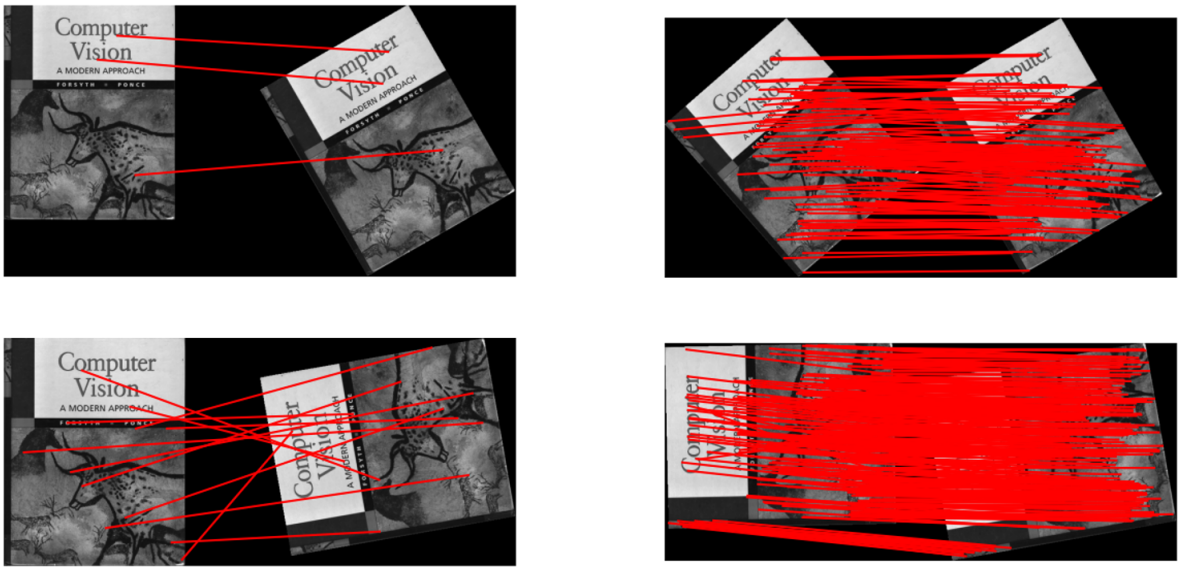
The matches it finds for each rotation angle varies a lot. This is because BREIF descriptor is not rotation invariant. Since it compares patches by pixel to pixel, if it rotates a bit, then the patch description becomes completely different.

```
def rotTest(opts):
    ratio = opts.ratio  # 'ratio for BRIEF feature descriptor'
    sigma = opts.sigma  # 'threshold for corner detection using FAST feature
    detector'

    # Read the image and convert to grayscale, if necessary
    I1 = cv2.imread('../data/cv_cover.jpg')
    I1 = cv2.cvtColor(I1, cv2.COLOR_BGR2GRAY)

```

```
    match_list = []
    for i in range(36):
        # Rotate Image
        I2 = ndimage.rotate(I1, 10 * i, reshape=False)

        # Compute features, descriptors and Match features
        locs1 = corner_detection(I1, sigma)
        locs2 = corner_detection(I2, sigma)

        # TODO: Obtain descriptors for the computed feature locations
        desc1, locs1 = computeBrief(I1, locs1)
        desc2, locs2 = computeBrief(I2, locs2)

        # TODO: Match features using the descriptors
        matches = briefMatch(desc1, desc2, ratio)
        # print(matches.shape)
        match_list.append(matches.shape[0])

        # plotMatches(im1, im2, matches, locs1, locs2)
    # Update histogram
    # match_hist = plt.hist(match_list)
    plt.bar([i for i in range(36)], match_list, color='slateblue')

    # Display histogram
    # plt.hist(match_list)
    plt.show()
```

Q 2.1.7

1)



The first row is when the image is rotated by 30 degrees. The second row is when the image is rotated by 100 degrees. The first column is when using not rotation invariant BRIEF. The Second column is when I use my rotation invariant implementation.

To make the BRIEF descriptor rotation invarient, I did some googling and found out that there is a ORB algorithm that appearently uses the FAST and BRIEF and makes it rotation invariant. I read its paper $http://www.gwylab.com/download/ORB_2012.pdf$ and in section [3.2. Orientation by Intensity Centroid], it introduces how we can try to calculate the orientation of the image and use it to rotate it before using the BRIEF descriptor. I tried implementing it the same way as the paper and it didn't really work :(. So I thought of some other ways I could calculate the orientation of the image. I came up with a really constraint algorithm we can use if we have the same image in different rotations. I take the mean of the locations that the feature extractor gives and calculate the orientation of the image based on that. I get the orientation of both images and rotate the image to have the same alignment. It works well for when the image is rotated.

```
def matchPics(I1, I2, opts):
    w, h, c = I1.shape

    ratio = opts.ratio  # 'ratio for BRIEF feature descriptor'
    sigma = opts.sigma  # 'threshold for corner detection using FAST feature
    detector'

    # TODO: Convert Images to GrayScale
    I1 = skimage.color.rgb2gray(I1)
    I2 = skimage.color.rgb2gray(I2)

    # TODO: Detect Features in Both Images
    locs1 = corner_detection(I1, sigma)
    locs2 = corner_detection(I2, sigma)
```

```
14
15      # EXTRA CREDIT try 2 (works!!!!!)
16      mean_1 = np.mean(locs1, axis=0).reshape(2, 1)
17      mean_2 = np.mean(locs2, axis=0).reshape(2, 1)
18
19      or_1 = math.atan2(mean_1[0] - w / 2, mean_1[1] - h / 2)
20      or_2 = math.atan2(mean_2[0] - w / 2, mean_2[1] - h / 2)
21
22      # rotate I1 to match I2 rotation
23      rad = or_1 - or_2
24      deg = rad * 180 / math.pi
25      I1_r = ndimage.rotate(I1, deg, reshape=True)
26
27      locs1_r = corner_detection(I1_r, sigma)
28      # TODO: Obtain descriptors for the computed feature locations
29      desc1, locs1 = computeBrief(I1_r, locs1_r)
30      desc2, locs2 = computeBrief(I2, locs2)
31
32      # TODO: Match features using the descriptors
33      matches = briefMatch(desc1, desc2, ratio)
34
35      return matches, locs1, locs2, deg
```

2.2 Homography Computation

Q 2.2.1

```
def computeH(x1, x2):
    # Q2.2.1
    N = x1.shape[0]
    # Compute the homography between two sets of points
    A = [[[-x2[i, 0], -x2[i, 1], -1, 0, 0, 0, x1[i, 0] * x2[i, 0], x1[i, 0] * x2
    [i, 1], x1[i, 0]],
          [0, 0, 0, -x2[i, 0], -x2[i, 1], -1, x1[i, 1] * x2[i, 0], x1[i, 1] * x2
    [i, 1], x1[i, 1]]]
         for i in range(N)
         ]
    A = np.array(A).reshape(2 * N, -1)
    u, s, vh = np.linalg.svd(A)
    H2to1 = vh[-1, :].reshape(3, 3)

    return H2to1
```

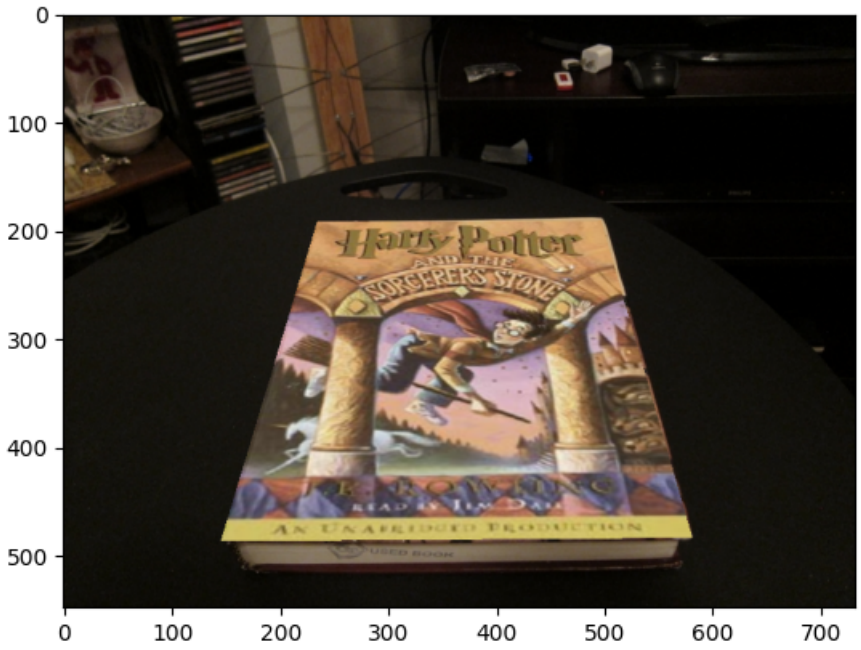
Q 2.2.2

```python
def computeH_norm(x1, x2):
    # Q2.2.2
    # Compute the centroid of the points
    x1_x = np.mean(x1[:, 0])
    x1_y = np.mean(x1[:, 1])

    x2_x = np.mean(x2[:, 0])
    x2_y = np.mean(x2[:, 1])

    # Shift the origin of the points to the centroid
    x1_n, x2_n = np.copy(x1), np.copy(x2)
    x1_n[:, 0] = x1_n[:, 0] - x1_x
    x1_n[:, 1] = x1_n[:, 1] - x1_y

    x2_n[:, 0] = x2_n[:, 0] - x2_x
    x2_n[:, 1] = x2_n[:, 1] - x2_y

    # Normalize the points so that the largest distance from the origin is equal
     to sqrt(2)
    dist1 = [np.linalg.norm(x1_n[i, :]) for i in range(x1.shape[0])]
    dist2 = [np.linalg.norm(x2_n[i, :]) for i in range(x2.shape[0])]
    s1 = math.sqrt(2) / max(dist1)
    s2 = math.sqrt(2) / max(dist2)
    x1_n = x1_n * s1
    x2_n = x2_n * s2

    # Similarity transform 1
    T1 = np.array([[s1, 0, -x1_x * s1],
                   [0, s1, -x1_y * s1],
                   [0, 0, 1]])

    # Similarity transform 2
    T2 = np.array([[s2, 0, -x2_x * s2],
                   [0, s2, -x2_y * s2],
                   [0, 0, 1]])
    # Compute homography
    H = computeH(x1_n, x2_n)

    # Denormalization
    H2to1 = np.linalg.inv(T1) @ H @ T2
    return H2to1
```

Q 2.2.3

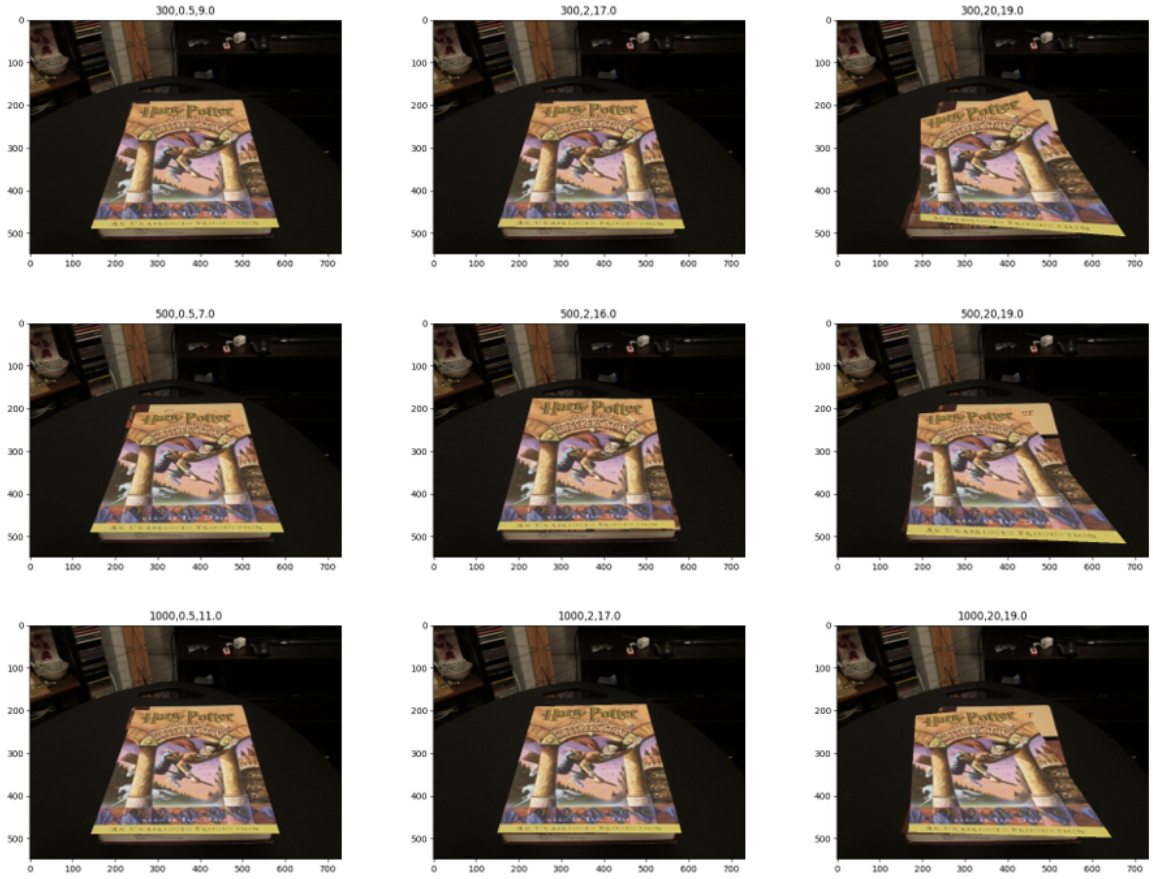
```python
def computeH_ransac(locs1, locs2, opts):
    # Q2.2.3
    # Compute the best fitting homography given a list of matching points

    max_iters = opts.max_iters  # the number of iterations to run RANSAC for
    inlier_tol = opts.inlier_tol  # the tolerance value for considering a point to be an inlier

    locsf1 = np.fliplr(locs1)
    locsf2 = np.fliplr(locs2)

    iters = 0
    max_right = 0

    rand_len = locs1.shape[0]

    while iters < max_iters:
        iters += 1
        # print(iters, end=" ")

        # rand_points = np.random.choice(rand_len, 4)
        rand_points = np.array(rd.sample(range(rand_len), 4))

        H2to1 = computeH_norm(locsf1[rand_points, :], locsf2[rand_points, :])
        img2_warped = np.matmul(H2to1,
                                np.concatenate((locsf2, np.ones((locsf2.shape[0], 1))), axis=1).transpose())
        img2_warped = img2_warped / img2_warped[2, :]
        img2_warped = img2_warped[:2, :].T

        right = np.zeros((rand_len, 1))
        for i in range(rand_len):
            if np.linalg.norm(img2_warped[i, :] - locsf1[i, :]) < inlier_tol:
                right[i] = 1
            else:
                right[i] = 0
        if np.sum(right) > max_right:
            bestH2to1 = H2to1
            inliers = right
            max_right = np.sum(right)
    print("::", np.sum(max_right))

    bestH2to1 = bestH2to1.reshape(3, 3)
    return bestH2to1, inliers
```

Q 2.2.4




```python
def warpImage(opts):
    # Read the image and convert to grayscale, if necessary
    I1 = cv2.imread('../data/cv_cover.jpg')

    I2 = cv2.imread('../data/cv_desk.png')

    I3 = cv2.imread('../data/hp_cover.jpg')
    I3 = cv2.resize(I3, (I1.shape[1], I1.shape[0]))

    matches, locs1, locs2 = matchPics(I1, I2, opts)

    plotMatches(I1, I2, matches, locs1, locs2)
    loc1_, loc2_ = match_maker(matches, locs1, locs2)

    bestH2to1, inliers = computeH_ransac(loc1_, loc2_, opts)

    composite_img = compositeH(bestH2to1, I3, I2)
    composite_img = composite_img[:, :, [2, 1, 0]]

    plt.imshow(composite_img)
    plt.show()
```

Q 2.2.5



Naturally, we can assume that the results of H will get better when we increase the max-iter since it is randomly sampling points and the more tries we do, the better the values will be. For the inlier-tols, we will get better H's when we make the number of inliers less. However, if we make it too small, there will not be any inliers left for us to use.

In the image, each row represents iterations of 300, 500, 1000. And the columns represent inliers of 0.5, 2, 20. So it is natural to see that the results are best at the bottom left corner (but if I make the inlier-tol too small, this won't be true) and gets worse as it goes to the right top corner.

3 Creating your Augmented Reality application

3.1 Incorporating video

Figure 4: Images at different time frame

VIDEO LINK

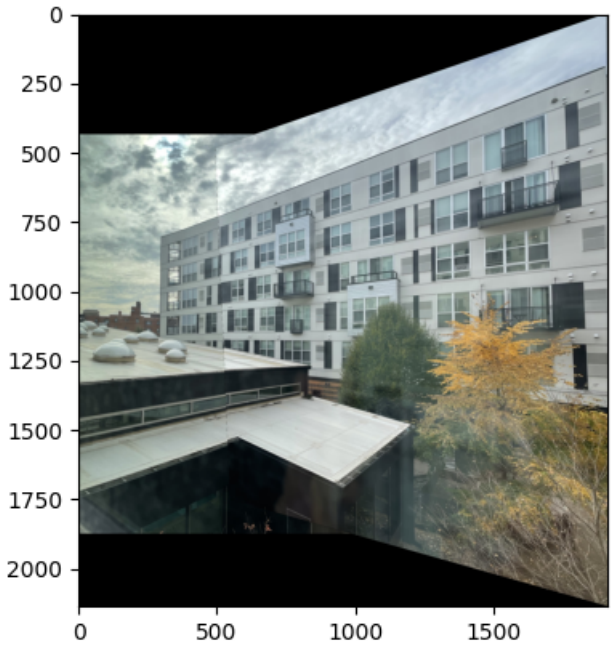
```
import cv2
import numpy as np
from PIL import Image

from displayMatch import plotMatches
from helper import loadVid
from matchPics import matchPics
from opts import get_opts
from planarH import computeH_ransac, compositeH, match_maker

# Import necessary functions

opts = get_opts()
opts.ratio = 0.7  # 'ratio for BRIEF feature descriptor'
opts.sigma = 0.05  # 'threshold for corner detection using FAST feature detector'

path1 = "../data/ar_source.mov"
path2 = "../data/book.mov"

ar_source = loadVid(path1)
book = loadVid(path2)

book_cover = cv2.imread('../data/cv_cover.jpg')

ratio = book_cover.shape[0] / book_cover.shape[1]
crop_y = ar_source.shape[1] - 100
crop_x = int(crop_y / ratio)

cropped = np.array(
    [a[50:crop_y, ar_source.shape[0] // 2 - crop_x // 2:ar_source.shape[0] // 2 + crop_x // 2, :] for a in
     ar_source])
cropped_scaled = np.array(
    [cv2.resize(c, (book_cover.shape[1], book_cover.shape[0])) for c in
     cropped])

N = min(ar_source.shape[0], book.shape[0])
```

```
37
38 cap = cropped_scaled
39 count = 0
40 composite_img_list = []
41 while count < N:
42
43     frame = cap[count]
44
45     matches, locs1, locs2 = matchPics(book_cover, book[count], opts)
46
47     plotMatches(book_cover, book[count], matches, locs1, locs2)
48     loc1_, loc2_ = match_maker(matches, locs1, locs2)
49     bestH2to1, inliers = computeH_ransac(loc1_, loc2_, opts)
50
51     composite_img = compositeH(bestH2to1, cropped_scaled[count], book[count])
52     composite_img = composite_img[:, :, [2, 1, 0]]
53     cv2.imshow('plz work', composite_img)
54     cv2.imwrite("frame%d.jpg" % count, composite_img)
55     composite_img_list.append(composite_img)
56
57     im = Image.fromarray(composite_img)
58     im.save("../result/vid5/" + str(count) + ".jpg")
59     count = count + 1
60
61     if cv2.waitKey(10) & 0xFF == ord('q'):
62         break
63
64 cv2.destroyAllWindows()  # destroy all opened windows
```

3.2 Make Your AR Real Time

4 Create a Simple Panorama

Figure 5: Images left and Image right




```
path1 = '../data/left.jpg'
path2 = '../data/right.jpg'
opts = get_opts()
ratio = opts.ratio  # 'ratio for BRIEF feature descriptor'
```

```
sigma = opts.sigma  # 'threshold for corner detection using FAST feature
    detector'

# Read the image and convert to grayscale, if necessary
I1 = cv2.imread(path1)

I2 = cv2.imread(path2)

ctrpts = cpselect(path1, path2)

locs1, locs2 = [], []

for i in ctrpts:
    id = i['point_id']
    x1 = i['img1_x']
    y1 = i['img1_y']
    x2 = i['img2_x']
    y2 = i['img2_y']
    locs1.append([y1, x1])
    locs2.append([y2, x2])
loc1_, loc2_ = np.array(locs1), np.array(locs2)

bestH2to1, inliers = computeH_ransac(loc1_, loc2_, opts)

composite_img = compositeH(bestH2to1, I1, I2)
composite_img = composite_img[:, :, [2, 1, 0]]

plt.imshow(composite_img)
plt.show()
```