CMU Fall24 16820 Homework 1

Patrick Chen

Collaborators:

September 7, 2024

Q1.1 at page 2

Ans:

Given two camera projection matrices P_1 and P_2 corresponding to two cameras: camera 1 and camera 2, I have the following equations (1) and equations (2), where x_1 represents the projected 2D point, on the image plane of camera 1, of a 3D point, x_π, from a different 3D plane π, and x_2 represents the projected 2D point of the same 3D point, x_π, on the image plane of camera 2.

$$x_1 = P_1 x_\pi \tag{1}$$

$$x_2 = P_2 x_\pi \tag{2}$$

Both of equation (1) and (2) has the same 3D point x_π, it leads to the following equation (3):

$$P_1^{-1} x_1 = P_2^{-1} x_2 \tag{3}$$

Then I move P_1 to the right hand side by both multiply P_1 in equation (1) and (2), then I got equation (4).

$$x_1 = P_1 P_2^{-1} x_2 \tag{4}$$

It is the same form of the given equation (1) at page 2, which is the equation (5) here, where λ is a scale:

$$x_1 = \lambda H x_2 \tag{5}$$

So I get equation (6):

$$H = \lambda P_1 P_2^{-1} \tag{6}$$

Consequently, if I need to prove that H does exist, I need to prove that P_2^{-1} does exist. As x_π lies on a plane, it only has two degrees of freedom, and can be represented as a 2D point. So the projection matrix P_1 and P_2 can be reduced to simplified 3x3 Homography matrix H_1 and H_2, since the Z dimension in 3D world coordinate of x_π is no longer needed as the projection now is only 2D plane to 2D plane transformation. So it gives the following equation:

$$H = \lambda H_1 H_2^{-1} \tag{7}$$

Under this 2D plane to 2D plan transformation, the Homography between camera plane and 2D plans, π, can be simply reduced to KR, where K is the intrinsic matrix of the camera, and R is the rotation matrix. It is because that the translation is no longer needed as the camera plane can always be rotated to be parallel to the 2D plane, π, and do the further intrinsic mapping. So $H_1 = K_1 R_1$ and $H_2 = K_2 R_2$, $H_2^{-1} = R_2^{-1} K_2^{-1}$. Because K_2 and R_2 are invertible, it makes H_2 is also invertible, so H does exist: $H = \lambda H_1 H_2^{-1}$.

Q1.2 at page 3

Ans:

1. h has 9 variables, but it only has 8 degrees of freedom. Specifically, h is the transformation of two 2D homogeneous coordinates, and if it is multiplied by a scale, λ, then the transformation equation still holds because of homogeneous coordinates. As a result, I have to add a constraint like setting the last element of h to be 1 or setting $\|h_2\| = 1$, and it decreases the degree for freedom by 1. So, h has 8 degrees of freedom.

2. Since h has 8 unknowns, and each point pair can contribute to 2 equations for solving h, I need 4 pairs of points to solve h.

3. Substitute the given equation of $\mathbf{x}_1^i \equiv H\mathbf{x}_2^i \quad (i \in \{1 \dots N\})$ with variables:

$$\begin{bmatrix} x_1^i \\ y_1^i \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_2^i \\ y_2^i \\ 1 \end{bmatrix}$$

Then, I expand the equations to get:

$$\begin{aligned} x_1^i &= h_{11}x_2^i + h_{12}y_2^i + h_{13} \\ y_1^i &= h_{21}x_2^i + h_{22}y_2^i + h_{23} \\ 1 &= h_{31}x_2^i + h_{32}y_2^i + h_{33} \end{aligned}$$

By dividing the last element of homogeneous coordinates to get real plane coordinates, I get the following two equations:

$$x_1^i = \frac{h_{11}x_2^i + h_{12}y_2^i + h_{13}}{h_{31}x_2^i + h_{32}y_2^i + h_{33}}$$

$$y_1^i = \frac{h_{21}x_2^i + h_{22}y_2^i + h_{23}}{h_{31}x_2^i + h_{32}y_2^i + h_{33}}$$

Though multiplying through by denominator and then rearrange:

$$\begin{aligned} h_{11}x_2^i + h_{12}y_2^i + h_{13} - h_{31}x_2^i x_1^i - h_{32}y_2^i x_1^i - h_{33}x_1^i = 0 \\ h_{21}x_2^i + h_{22}y_2^i + h_{23} - h_{31}x_2^i y_1^i - h_{32}y_2^i y_1^i - h_{33}y_1^i = 0 \end{aligned}$$

Then I rearrange it to fit into the matrix form $A_i h = 0$

$$\begin{bmatrix} x_2^i & y_2^i & 1 & 0 & 0 & 0 & -x_1^i x_2^i & -x_1^i y_2^i & -x_1^i \\ 0 & 0 & 0 & x_2^i & y_2^i & 1 & -y_1^i x_2^i & -y_1^i y_2^i & -y_1^i \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Then I derive A_i:

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

4. (a) A trivial solution for h is all 0:

$$h = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

   (b) A is not full rank. In homogeneous system, if h is not trivial, then A must not be invertible, or the only solution of h would be a zero vector (trivial). It means that A should be singular, in other words, A must not be full rank.

   (c) The impact of A being not full rank is that the smallest singular values would be zero or near zero, allowing to find a non-trivial solution h.

Q1.4.1 at page 4

Ans:

Given two cameras separated by a pure rotation, I have the following two equations:

$$x_1 = K_1 I X \tag{8}$$

$$x_2 = K_2 R X \tag{9}$$

Since X is the same data point in 3D space, then I can substitute X in equation (7) with X in equation (8) and result in the following equation:

$$x_1 = K_1 R^{-1} K_2^{-1} x_2 \tag{10}$$

Since $x_1 = \lambda H x_2$, then I can know that:

$$H = \frac{1}{\lambda} K_1 R^{-1} K_2^{-1} \tag{11}$$

Since K_2 and R are invertible, $\det(K_2) \neq 0$ and $\det(R) \neq 0$, H does exist.

Q1.4.2 at page 4

Ans:

Recall from Q1.4.1, $H = K_1 R(\theta)^{-1} K_2^{-1}$, given that θ is the angle of rotation. Since $K_1 = K_2 = K$, I have:

$$H = KR(\theta)^{-1} K^{-1} \tag{12}$$

By multiplying two H together:

$$H^2 = (KR(\theta)^{-1} K^{-1})(KR(\theta)^{-1} K^{-1}) \tag{13}$$

By removing redundant identity matrix:

$$H^2 = KR(\theta)^{-1} R(\theta)^{-1} K^{-1} \tag{14}$$

According to the property of rotation matrix $R(\theta)^{-1} = R(\theta)^T = R(-\theta)$, and $R(-\theta)R(-\theta) = R(-2\theta)$, I have $R(\theta)^{-1} R(\theta)^{-1} = R(2\theta)^{-1}$, so I have:

$$H^2 = KR(2\theta)^{-1} K^{-1} \tag{15}$$

It demonstrates that H^2 corresponds to a rotation of 2θ.

Q1.4.3 at page 4

Ans:

It is because that either one of the following two conditions is needed to be held for the planar homography to exist:

1. The points in the 3D world lie on a 2D plane.
2. Only pure rotation between two views.

As a result, planar homography is not completely sufficient to map any scene image to another viewpoint.

Q1.4.4 at page 5

Ans:

The projection matrix P is a 3×4 matrix that can map a 3D point, X, to a 2D point, x:

$$x = PX \tag{16}$$

A line in 3D space can be represented as the following equation, where t is a parameter:

$$X(t) = X_1 + t(X_2 - X_1) \tag{17}$$

If I multiply the projection matrix, P, on both hand sides of equation (16), then I have:

$$PX(t) = PX_1 + tP(X_2 - X_1) \tag{18}$$

By moving P into parenthesis, then:

$$PX(t) = PX_1 + t(PX_2 - PX_1) \tag{19}$$

Then, I got the following equation:

$$x(t) = x_1 + t(x_2 - x_1) \tag{20}$$

Equation (19) is the equation of line on a 2D plane. As a result, the line can be preserved through perspective projection P.

Q2.1.1 at page 5

Ans:

FAST detector has the following differences when comparing with Harris corner detector:

1. FAST detector is not rotation-invariant, while Harris corner detector can be rotation-invariant with some tricky implementation details.
2. FAST detector only compare intensities with neighbors in a circular range around a candidate point, while Harris corner detector uses much more expansive computations like first order derivative (gradient), summation of a window of gradients, and solving eigenvalue problems, leading to the result that FAST detector is much more faster than Harris corner detector.
3. FAST detector is more sensitive to noise and the change of lighting, while Harris corner detector is more robust to noise and lighting change because Harris corner detector gets more information from gradients in multiple directions of a window plus solving eigenvalue problems to get optimized candidates of corners.
4. FAST detector is faster but less accurate in the localizing features, while Harris corner detector is slower but more accurate in localizing feature points.

Q2.1.2 at page 5

Ans:

BRIEF descriptor has the following differences when comparing with filterbanks descriptor:

1. BRIEF descriptor chooses n pairs of points within a predefined window to form a n-bit binary descriptor by setting each bit 1 or 0 according to the comparison of each pair of selected points, while filterbanks convolve the image with a series of predefined filters to form an array of responses in each pixel as the descriptor.

2. BREIF descriptor is faster than filterbanks descriptor as BRIEF descriptor only use sampling and comparison of intensity, while filterbanks descriptor applys convolution of different filters to produce different response as the descriptor.

3. As for the robustness, filterbanks descriptor is more robust to rotation and scale as it can apply different filters to mitigate the impact from scaling and rotating, while BRIEF descriptor only compares the intensity within a window and it is not rotation-invariant and scale-invariant.

Q2.1.3 at page 5

Ans:

The Hamming distance is to calculate the number of different bits in two binary descriptors. For example, let x_1 be a d-dimensional binary descriptor: $x_1 = (x_1^0, x_1^1, ..., x_1^{(d-1)})$, and x_2 is another d-dimensional binary descriptor: $x_2 = (x_2^0, x_2^1, ..., x_2^{(d-1)})$, the Hamming distance is defined as the following:

$$d_H(\mathbf{x_1}, \mathbf{x_2}) = \sum_{i=1}^{d} \delta(x_1^i, x_2^i), where\ \delta(x_1^i, x_2^i) = \begin{cases} 0 & \text{if } x_1^i = x_2^i, \\ 1 & \text{if } x_1^i \neq x_2^i. \end{cases} \tag{21}$$

Hamming distance has the following benefits over conventional Euclidean distance measure:

1. Hamming distance is faster than Euclidean distance because Hamming distance only needs to compute the number of different bits among two binary vectors, and it can be computed by efficient XOR and counter operations, while Euclidean distance needs to compute several complex operations, like vector substraction, summation of square and square root operations, making Euclidean distance slow and expensive for computation.

2. In our setting, I use BREIF descriptor on our detected feature points, resulting in vectors in binary form, and it is definitely suitable to use Hamming distance to compare two binary feature descriptor vectors.

Q2.1.4 at page 5

Ans:

The best output is as the following Figure 1, and the corresponding code snippets are as the following Figure 2. The sigma is set to 0.08, and ration is set to 0.61.

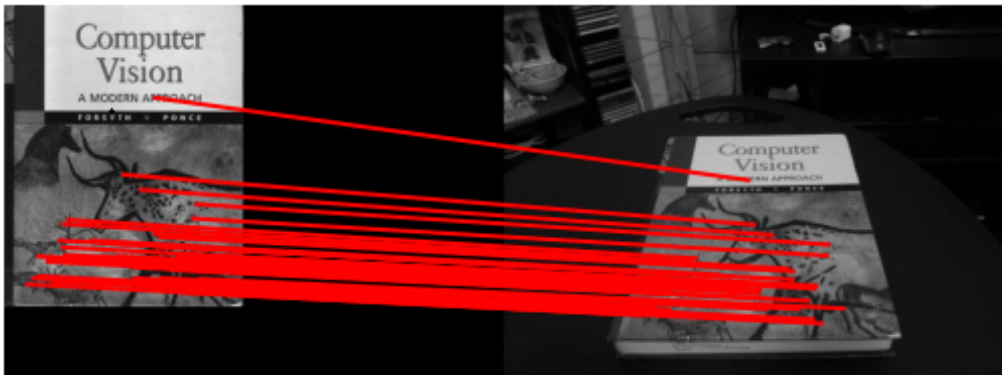


Figure 1: Best Output

```python
# Q2.1.4

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
        # Convert BGR order (from cv2.imread()) to RGB order
        image1_rgb = I1[:, :, [2, 1, 0]]
        image2_rgb = I2[:, :, [2, 1, 0]]
        gray_I1 = skimage.color.rgb2gray(image1_rgb)
        gray_I2 = skimage.color.rgb2gray(image2_rgb)

        # TODO: Detect Features in Both Images
        locs1 = corner_detection(gray_I1, sigma);
        locs2 = corner_detection(gray_I2, sigma);


        # TODO: Obtain descriptors for the computed feature locations
        desc1, locs1 = computeBrief(gray_I1, locs1)
        desc2, locs2 = computeBrief(gray_I2, locs2)


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

        return matches, locs1, locs2

```

Figure 2: Q2.1.4 Code Snippet

Q2.1.5 at page 6

Ans:

According to skimage API documentation, the sigma value is used in FAST detector to decide whether the pixels on the circle are brighter, darker, or similar with respect to the test pixel. The lower value of sigma would produce more detected feature points, while the higher value would produce less detected feature points. On the other hand, the ratio value is used in $skimage.feature.match\_descriptors()$ API. It is the ratio between the distance of the first closest descriptors to the second closest descriptors. Thus, the higher the value would cause more matched correspondence as the matched condition is more easy, while the lower the value would cause less matched correspondence. The following experiments listed in Table 1 is started with the given values of sigma=0.15 and ratio=0.7. As I increase sigma, it results in more false positive matched points, such as the read line near the top of the image in row 1 and row 2 of Table 1. So I decrease sigma to 0.10, and it shows up more unmatched points. Thus, I also decrease ratio to 0.65, and it shows less mismatched correspondence. As I decrease sigma to 0.08, and ratio to 0.61, I think I found the optimized result, as there is no mismatched correspondence. As I keep decreasing sigma to 0.7, the mismatched correspondence near the right border of the book in the right image shows up. As I decrease ratio from 0.61 to 0.55, the mismatched correspondence does not disappear. As the result, I think sigma=0.08 and ratio=0.61 is my best result.

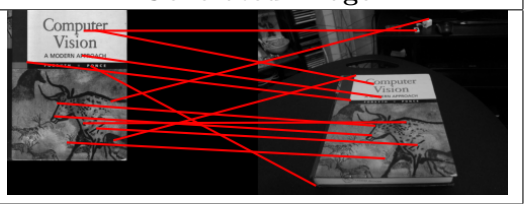
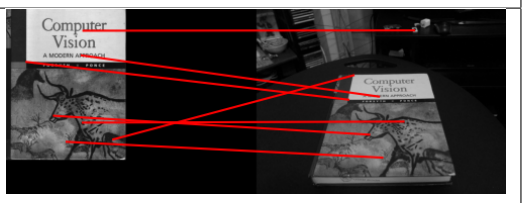
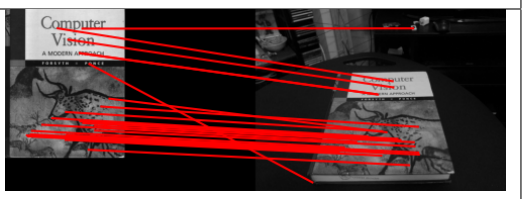
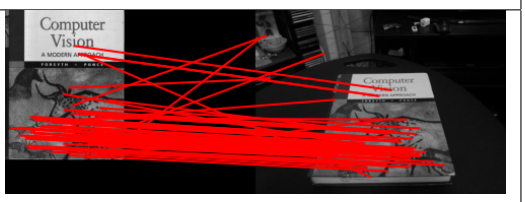
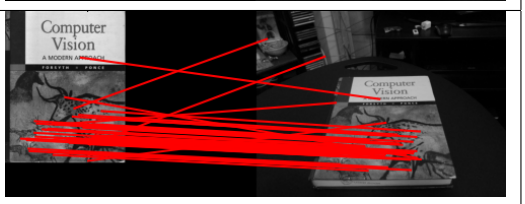
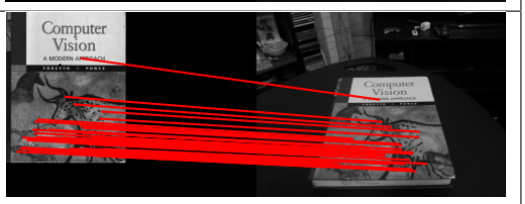
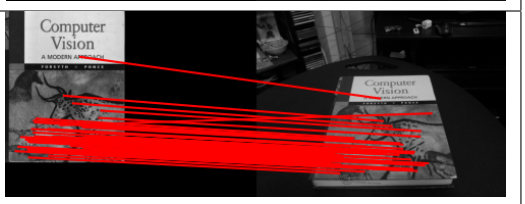
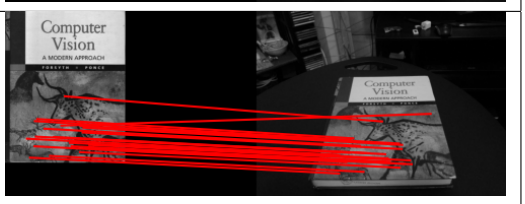
Parameter sigma	Parameter ratio	Generated Image
0.20	0.7	
0.20	0.65	
0.15	0.7	
0.10	0.7	
0.10	0.65	
0.08	0.61	
0.07	0.61	
0.06	0.55	

Table 1: Parameter Changes and Corresponding Generated Images

Q2.1.6 at page 6

Ans:

As shown in the Figure 3, the maximum of matches occurs at rotation = 0 degrees, and the rotation makes the number of matches decrease drastically. I think it is because the BRIEF descriptor and the FAST detector using in the matchPics() function is not rotation-invariant.

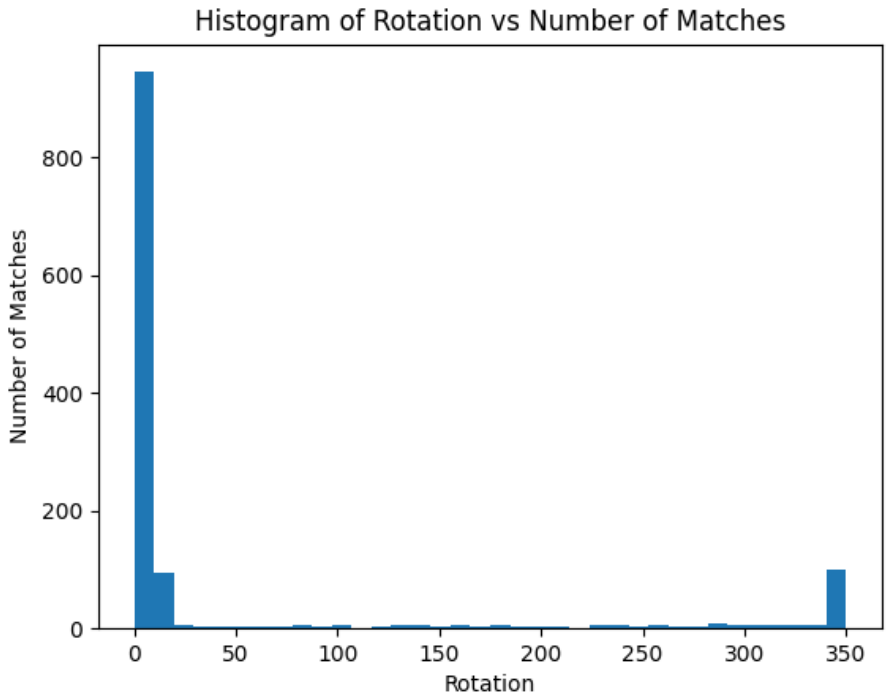
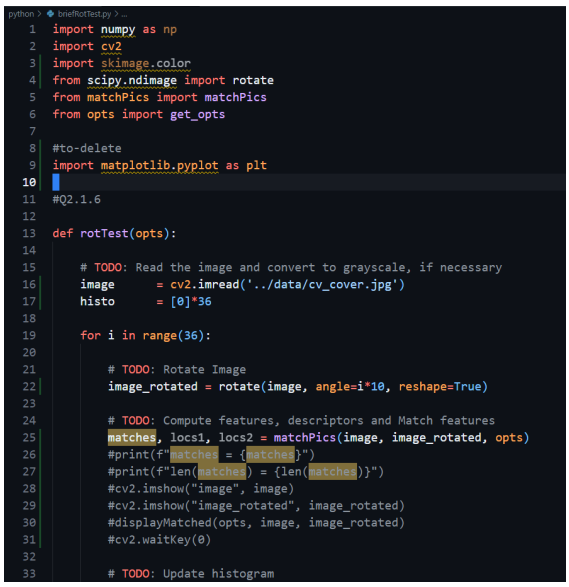
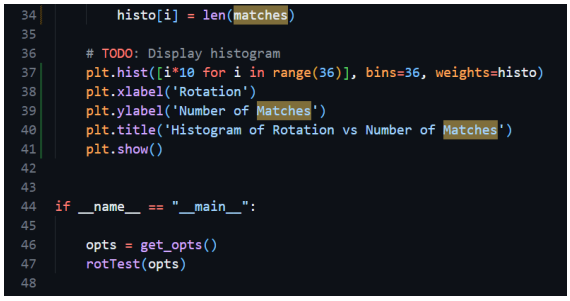


Figure 3: Histogram Result



```python
import numpy as np
import cv2
import skimage.color
from scipy.ndimage import rotate
from matchPics import matchPics
from opts import get_opts

#to-delete
import matplotlib.pyplot as plt

#Q2.1.6

def rotTest(opts):

    # TODO: Read the image and convert to grayscale, if necessary
    image      = cv2.imread('../data/cv_cover.jpg')
    histo      = [0]*36

    for i in range(36):

        # TODO: Rotate Image
        image_rotated = rotate(image, angle=i*10, reshape=True)

        # TODO: Compute features, descriptors and Match features
        matches, locs1, locs2 = matchPics(image, image_rotated, opts)
        #print(f"matches = {matches}")
        #print(f"len(matches) = {len(matches)}")
        #cv2.imshow("image", image)
        #cv2.imshow("image_rotated", image_rotated)
        #displayMatched(opts, image, image_rotated)
        #cv2.waitKey(0)

        # TODO: Update histogram
```

Figure 4: Q2.1.6 Code Snippet 1



```python
        histo[i] = len(matches)

    # TODO: Display histogram
    plt.hist([i*10 for i in range(36)], bins=36, weights=histo)
    plt.xlabel('Rotation')
    plt.ylabel('Number of Matches')
    plt.title('Histogram of Rotation vs Number of Matches')
    plt.show()


if __name__ == "__main__":

    opts = get_opts()
    rotTest(opts)
```

Figure 5: Q2.1.6 Code Snippet 2

Q2.1.7 at page 6

Ans:
//to-do

Q2.2.1 at page 7

Ans:
Figure 6 shows the code snippet.

```python
def computeH(x1, x2):
    #Q2.2.1
    # TODO: Compute the homography between two sets of points
    # Construct A matrix
    A = np.zeros((2*x1.shape[0], 9))
    for i in range(x1.shape[0]):
        xi1 = x1[i][0]
        yi1 = x1[i][1]
        xi2 = x2[i][0]
        yi2 = x2[i][1]
        A[2*i+0] = [xi2, yi2, 1,   0,   0, 0,  -xi1*xi2, -xi1*yi2, -xi1]
        A[2*i+1] = [  0,   0, 0, xi2, yi2, 1,  -yi1*xi2, -yi1*yi2, -yi1]

    # Construct ATA
    ATA = A.T@A

    # Calculate eigenvalues, eigenvectors
    eigenval, eigenvec = np.linalg.eig(ATA)

    # Choose the eigenvector w/ smallest eigenvalues
    smallest_eigenvec = eigenvec[:, np.argsort(eigenval)[0]]
    H2to1 = np.reshape(smallest_eigenvec, (3, 3))
    H2to1 = np.array(H2to1, dtype=np.float32)

    #test
    '''
    x2_homo = np.hstack([x2, np.ones((x2.shape[0], 1))]) #Nx3
    x1_recover_all = H2to1@x2_homo.T #3xN
    x1_recover_all = x1_recover_all.T #Nx3
    x1_col1 = x1_recover_all[:, 0]/x1_recover_all[:, 2]
    x1_col2 = x1_recover_all[:, 1]/x1_recover_all[:, 2]
    x1_recover_all = np.column_stack((x1_col1, x1_col2)) #Nx2
    print(f"original, x1_recover_all = {x1_recover_all}")
    '''

    return H2to1
```

Figure 6: Q2.2.1 Code Snippet

Q2.2.2 at page 7

Ans:
Figure 7 - Figure 9 show the code snippet.

```python
def computeDist(x, cx):
    # x : np.array: (2,)
    # cx: np.array: (2,)
    return np.linalg.norm(x-cx)

def computeLongestDist(x, cx):
    # x : np.array: (N, 2)
    # cx: np.array: (2,)
    '''
    dist = -1
    cand = np.array([0, 0])
    for xi in x:
        computed_dist = computeDist(xi, cx)
        if computed_dist > dist:
            dist = computed_dist
            cand[0] = xi[0]
            cand[1] = xi[1]
    return dist, cand
    '''
    return np.max(np.sqrt(np.sum((x-cx)**2, axis=1)))

def computeH_norm(x1, x2):
    #Q2.2.2
    # TODO: Compute the centroid of the points
    cx1, cy1 = np.mean(x1[:, 0]), np.mean(x1[:, 1])
    cx2, cy2 = np.mean(x2[:, 0]), np.mean(x2[:, 1])
    c1 = np.array([cx1, cy1])
    c2 = np.array([cx2, cy2])

    #test
    '''
    image = np.zeros((600, 600, 3), dtype = np.uint8)
    for point in x1:
        cv2.circle(image, tuple(map(int, point)), radius=5, color=(0, 255, 0), thickness=-1)
    for point in x2:
        cv2.circle(image, tuple(map(int, point)), radius=5, color=(255, 0, 0), thickness=-1)
    cv2.circle(image, tuple(map(int, c1)), radius=5, color=(0, 0, 255), thickness=-1)
    cv2.circle(image, tuple(map(int, c2)), radius=5, color=(0, 0, 255), thickness=-1)
    '''

    # TODO: Shift the origin of the points to the centroid
    # TODO: Normalize the points so that the largest distance from the origin is equal to sqrt(2)
    # Merge these two TODO to form T1 and T2
    # Calculate the largest distance from the center (cx1, cy1), (cx2, cy2)
    longest_dist1 = computeLongestDist(x1, c1)
    longest_dist2 = computeLongestDist(x2, c2)
    norm_factor1 = np.sqrt(2)/longest_dist1
    norm_factor2 = np.sqrt(2)/longest_dist2

    #test
    '''
    cv2.circle(image, tuple(map(int, cand1)), radius=5, color=(20, 140, 255), thickness=-1)
    cv2.circle(image, tuple(map(int, cand2)), radius=5, color=(20, 140, 255), thickness=-1)
    cv2.line(image, tuple(map(int, c1)), cand1, color=(255, 255, 255), thickness=2)
    cv2.line(image, tuple(map(int, c2)), cand2, color=(255, 255, 255), thickness=2)
    cv2.imshow("Points on Blank Image", image)
```

Figure 7: Q2.2.2 Code Snippet 1

```python
    cv2.waitKey(0)
    '''

    # TODO: Similarity transform 1
    T1 = np.zeros((3, 3))
    T1[0][0] = norm_factor1
    T1[0][2] = -norm_factor1*c1[0]
    T1[1][1] = norm_factor1
    T1[1][2] = -norm_factor1*c1[1]
    T1[2][2] = 1
    x1_homo = np.hstack([x1, np.ones((x1.shape[0], 1))]) #Nx3
    x1_norm = T1@x1_homo.T #3xN
    x1_norm = x1_norm.T #Nx3
    x1_col1 = x1_norm[:, 0]/x1_norm[:, 2]
    x1_col2 = x1_norm[:, 1]/x1_norm[:, 2]
    x1_norm = np.column_stack((x1_col1, x1_col2)) #Nx2

    #test
    '''
    x1_norm_homo = np.hstack([x1_norm, np.ones((x1_norm.shape[0], 1))])
    x1_recover = np.linalg.inv(T1)@x1_norm_homo.T
    x1_recover = x1_recover.T
    x1_recover = np.column_stack((x1_recover[:, 0]/x1_recover[:, 2], x1_recover[:, 1]/x1_recover[:, 2]))
    print(f"x1_recover = {x1_recover}")
    '''


    # TODO: Similarity transform 2
    T2 = np.zeros((3, 3))
    T2[0][0] = norm_factor2
    T2[0][2] = -norm_factor2*c2[0]
    T2[1][1] = norm_factor2
    T2[1][2] = -norm_factor2*c2[1]
    T2[2][2] = 1
    x2_homo = np.hstack([x2, np.ones((x2.shape[0], 1))]) #Nx3
    x2_norm = T2@x2_homo.T #3xN
    x2_norm = x2_norm.T #Nx3
    x2_col1 = x2_norm[:, 0]/x2_norm[:, 2]
    x2_col2 = x2_norm[:, 1]/x2_norm[:, 2]
    x2_norm = np.column_stack((x2_col1, x2_col2)) #Nx2

    #test
    '''
    x2_norm_homo = np.hstack([x2_norm, np.ones((x2_norm.shape[0], 1))])
    x2_recover   = np.linalg.inv(T2)@x2_norm_homo.T
    x2_recover   = x2_recover.T
    x2_recover   = np.column_stack((x2_recover[:, 0]/x2_recover[:, 2], x2_recover[:, 1]/x2_recover[:, 2]))
    print(f"x2_recover = {x2_recover}")
    '''

    # TODO: Compute homography
    H2to1_norm = computeH(x1_norm, x2_norm)

    # TODO: Denormalization
    H2to1 = np.linalg.inv(T1)@H2to1_norm@T2

    #test
```

Figure 8: Q2.2.2. Code Snippet 2

```python
    '''
    x2_homo = np.hstack([x2, np.ones((x2.shape[0], 1))]) #Nx3
    x1_recover_all = H2to1@x2_homo.T #3xN
    x1_recover_all = x1_recover_all.T #Nx3
    x1_col1 = x1_recover_all[:, 0]/x1_recover_all[:, 2]
    x1_col2 = x1_recover_all[:, 1]/x1_recover_all[:, 2]
    x1_recover_all = np.column_stack((x1_col1, x1_col2)) #Nx2
    print(f"norm, x1_recover_all = {x1_recover_all}")
    '''

    return H2to1
```

Figure 9: Q2.2.2 Code Snippet3

Q2.2.3 at page 7

Ans:
Figure 10 - Figure 11 show the code snippet.

```python
def computeH_ransac(locs1, locs2, opts, fit_inlier_last: bool = False, fit_inlier_last_num: int = 4):
    #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

    if locs1.shape[0] < 4:
        raise ValueError(f"Error: The number of match point is {len(locs1)}, smaller than 4, not sufficient to calculate a Homography.")

    max_inlier_num = -1
    max_inliers = np.zeros(locs1.shape[0], dtype = int)
    for i in range(max_iters):
        # Randomly sample 4 points from locs1, locs2
        random_index = random.sample(range(0, locs1.shape[0]), 4)
        locs_sample1 = locs1[random_index, :]
        locs_sample2 = locs2[random_index, :]

        # Calculate the Homography
        H2to1 = computeH_norm(locs_sample1, locs_sample2)

        # Calculate the inline number by recovering x1 points
        locs2_homo = np.hstack([locs2, np.ones((locs2.shape[0], 1))]) #Nx3
        locs1_recover_all = H2to1@locs2_homo.T #3x6
        locs1_recover_all = locs1_recover_all.T #Nx3
        locs1_col1 = locs1_recover_all[:, 0]/locs1_recover_all[:, 2]
        locs1_col2 = locs1_recover_all[:, 1]/locs1_recover_all[:, 2]
        locs1_recover_all = np.column_stack((locs1_col1, locs1_col2)) #Nx2

        distance_sq = np.sum((locs1_recover_all-locs1)**2, axis=1)
        inliers = distance_sq <= inlier_tol**2
        '''
        for i in range(locs1_recover_all.shape[0]):
            recovered_pt = locs1_recover_all[i]
            original_pt  = locs1[i]
            dist = computeDist(recovered_pt, original_pt)
            if dist < inlier_tol:
                inliers[i] = 1
        '''

        inlier_num = np.sum(inliers)
        if inlier_num > max_inlier_num:
            max_inlier_num = inlier_num
            max_inliers    = inliers
            bestH2to1 = H2to1

    inliers = max_inliers

    #Fit the inlier again
    if fit_inlier_last:
        indices = [i for i, x in enumerate(inliers) if x == 1]
        if len(indices) >= fit_inlier_last_num:
            locs1_opt = locs1[indices, :]
            locs2_opt = locs2[indices, :]
            bestH2to1 = computeH_norm(locs1_opt, locs2_opt)

    #test
```

Figure 10: Q2.2.3 Code Snippet 1

```python
    '''
    x2 = locs2
    x2_homo = np.hstack([x2, np.ones((x2.shape[0], 1))]) #Nx3
    x1_recover_all = bestH2to1@x2_homo.T #3xN
    x1_recover_all = x1_recover_all.T #Nx3
    x1_col1 = x1_recover_all[:, 0]/x1_recover_all[:, 2]
    x1_col2 = x1_recover_all[:, 1]/x1_recover_all[:, 2]
    x1_recover_all = np.column_stack((x1_col1, x1_col2)) #Nx2
    print(f"ransac, x1_recover_all = {x1_recover_all}")
    '''

    return bestH2to1, inliers
```

Figure 11: Q2.2.3. Code Snippet 2

Q2.2.4 at page 8

Ans:

Figure 12 shows the result image, and Figure 13 - Figure 14 show the code snippet. Besides, in the step4, the reason that the book space in cv_desk.png is not filled up is that the size of hp_cover.jpg is not the same as cv_cover.jpg, and the homography warping cause some blank pixels without any mapping intensity values. The solution is to resize the hp_cover.jpg to the size of cv_cover.jpg as shown in Figure 13 line 32.

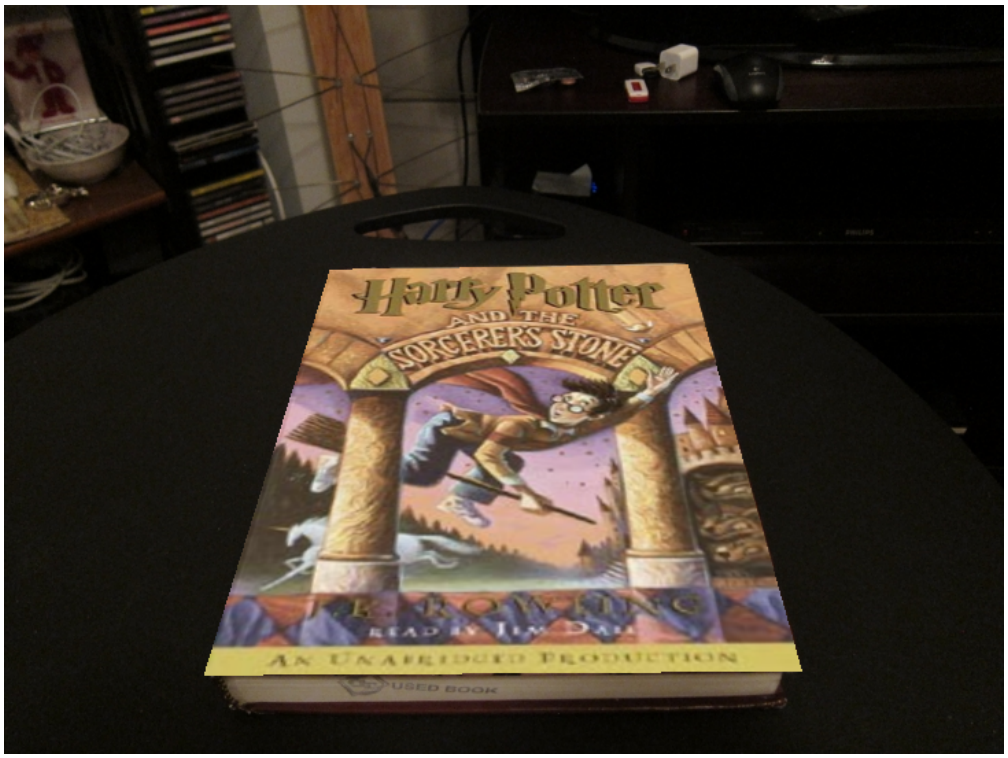


Figure 12: Q2.2.4 Result Image

```python
# Q2.2.4

def warpImage(opts):
    # Read Input Image
    image_cv_cover      = cv2.imread('../data/cv_cover.jpg')
    image_cv_desk       = cv2.imread('../data/cv_desk.png')
    image_hp_cover      = cv2.imread('../data/hp_cover.jpg')

    # Compute Homography
    matches, locs1, locs2 = matchPics(image_cv_desk, image_cv_cover, opts)
    x1 = locs1[matches[:, 0], :]
    x2 = locs2[matches[:, 1], :]
    x1_correct_pt = np.flip(x1, axis=1)
    x2_correct_pt = np.flip(x2, axis=1)
    H2to1_ransac, inliers = computeH_ransac(x1_correct_pt, x2_correct_pt, opts, fit_inlier_last=True, fit_inlier_last_num=4)

    # Resize the image_hp_cover to the size of image_cv_cover
    resize_image_hp_cover = cv2.resize(image_hp_cover, (image_cv_cover.shape[1], image_cv_cover.shape[0]))

    # Warping hp_cover.jpg to cv_desk.png
    warped_hp_image = cv2.warpPerspective(resize_image_hp_cover, H2to1_ransac, (image_cv_desk.shape[1], image_cv_desk.shape[0]))

    # Composite the warped hp_cover.jpg with cv_desk.png
    composite_img = compositeH(H2to1_ransac, resize_image_hp_cover, image_cv_desk)

    cv2.imshow('composite_img', composite_img)
    cv2.waitKey(0)
```

Figure 13: Q2.2.4 warpImage() Code Snippet

```python
def compositeH(H2to1, template, img):

    #Create a composite image after warping the template image on top
    #of the image using the homography

    #Note that the homography we compute is from the image to the template;
    #x_template = H2to1*x_photo
    #For warping the template to the image, we need to invert it.


    # TODO: Create mask of same size as template
    mask = np.full((template.shape[0], template.shape[1], 1), 255, dtype=np.uint8)
    composite_img = np.zeros((img.shape[0], img.shape[1], 3), dtype=np.uint8)

    # TODO: Warp mask by appropriate homography
    warped_mask = cv2.warpPerspective(mask, H2to1, (img.shape[1], img.shape[0]))

    # TODO: Warp template by appropriate homography
    warped_template = cv2.warpPerspective(template, H2to1, (img.shape[1], img.shape[0]))

    # TODO: Use mask to combine the warped template and the image
    inverse_mask    = cv2.bitwise_not(warped_mask)
    masked_template = cv2.bitwise_and(warped_template, warped_template, mask=warped_mask)
    masked_img      = cv2.bitwise_and(img, img, mask=inverse_mask)
    composite_img   = cv2.add(masked_template, masked_img)
    '''
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            if warped_mask[i, j] == 255:
                composite_img[i, j] = warped_template[i, j]
            else:
                composite_img[i, j] = img[i, j]
    '''

    return composite_img
```

Figure 14: Q2.2.4. CompositeH() Code Snippet

Q2.2.5 at page 8

Ans:

Table 2 shows the result of different setting among parameters 'max_iters' and 'inlier_tol'. According to equation (22) specified in the slides of lecture, as the inlier_tol decreases, it makes the feature points harder to be considered as an inlier, so the inlier rate w would decrease, and it makes the k (max_iters), the number of trials that can gaurantee a return of a good model, increased too. On the opposite, if inlier_tol increases, w would increase, and thus k (max_iters) would decrease. However, in Table 2, 2^{nd} row increase the inlier_tol from 2.0 to 10.0, and the result seems not quite good. The reason is that at the end of computeH_ransac() function, I would use all inliers again to fit an optimized homography. If the inlier_tol is too large, then some outlier match points may be deemed as inlier, and they would be included into the last step calculating of optimized homography, making the result worse. On the other hand, from 4^{th} row to 5^{th} row, the inlier_tol is decreased from 1 to 0.001, and the result seems worse in a great amount. The reason is that when iter_tol is too low, then most of the time the homography is calculated using outliers. Sometimes the homography is truly computed by true inliers, but the number of inlier just equals to those fitting using outliers because of the very low value of iter_tol. Accordingly, the RANSAC model selection process through inliers would become ineffective. In this problem, according to my experiments, the appropriate range for inlier_tol should be [0.5, 8], and the max_iters set to 500 is enough.

$$k = \frac{\log(1-p)}{\log(1-w^n)} \tag{22}$$

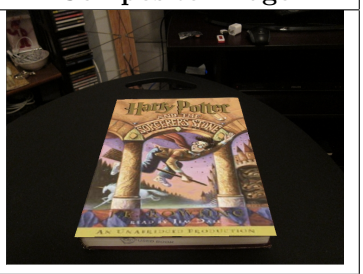
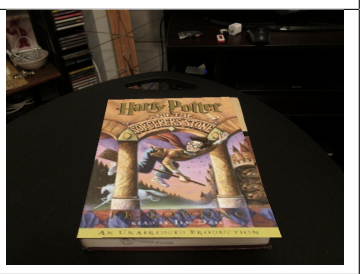
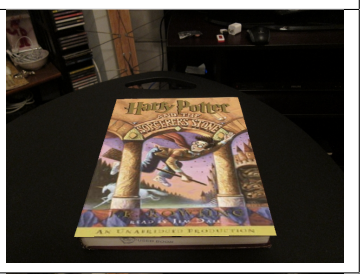
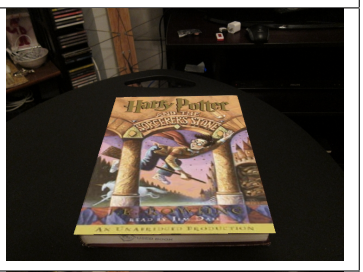
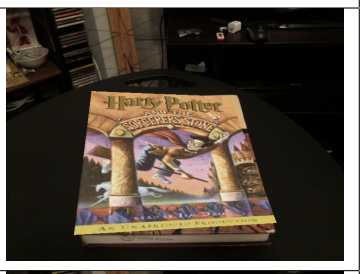
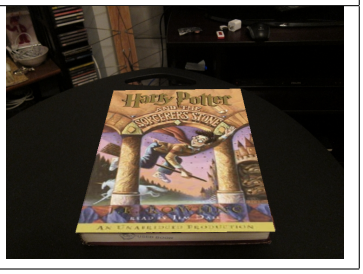
Parameter max_iters	Parameter inlier_tol	Composite Image
500	2.0	
500	10.0	
5000	2.0	
500	1.0	
500	0.001	
5000	0.001	

Table 2: Parameter Changes and Corresponding Composite Images

Q3.1 at page 9

Ans:

Figure 15 - Figure 17 shows the results of composite video where the timestamps displaying overlays at left, right, and center of the video frame. The resulted video is here: ar.avi, and the code snippets are as the following Figure 18 - Figure 20.

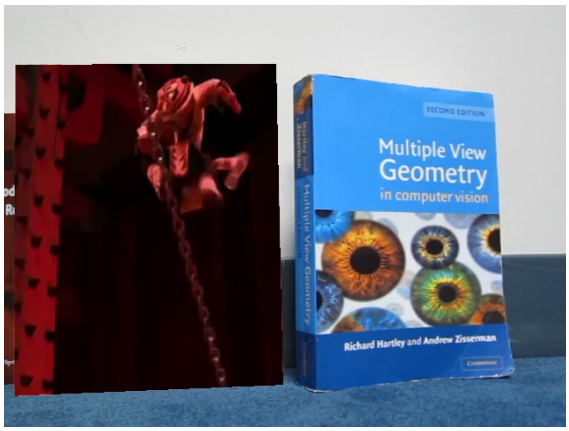


Figure 15: Q3.1 Screenshot of Result at Left Timestamp

Figure 16: Q3.1 Screenshot of Result at Right Timestamp

Figure 17: Q3.1 Screenshot of Result at Center Timestamp

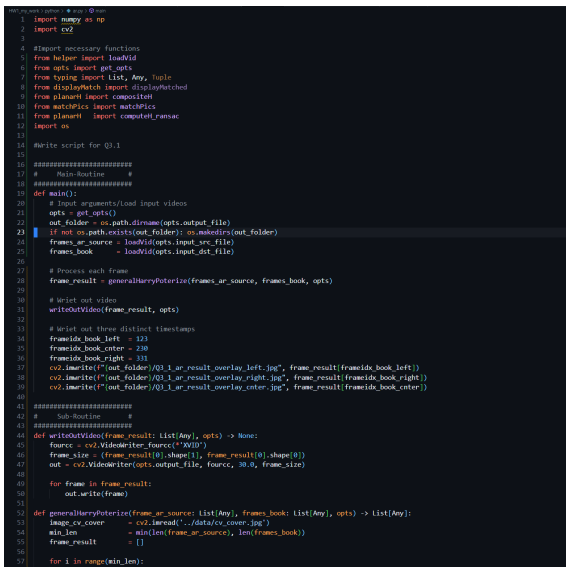


Figure 18: Q3.1 Code Snippet 1

Figure 19: Q3.1 Code Snippet 2

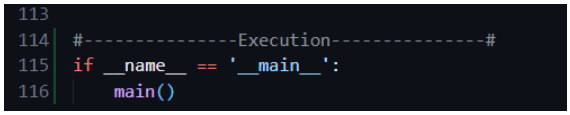


Figure 20: Q3.1 Code Snippet 3

Q3.2 at page 10

Ans:

In order to accelerate the whole program to approach the real-time performance, I firstly need to know which part of the program takes long time to process. To get the 30 FPS result, each for loop to process a frame should use less than 1/30 seconds, nearly 0.03 seconds. Figure 21 shows the code I added to do the profiling, and Figure 22 shows the result. As we can see from the result, both the time used to compute BRIEF descriptor and the match of the BRIEF descriptors take longer than 0.03 seconds, that is, original BRIEF descriptor takes 7.23 seconds to complete, and BRIEF matching takes 0.341 seconds to complete. One can reproduce the profiling result by referring to the command specified in python/README.md. When it comes to improving the whole process, my first thought is to down-sample the input frame, and use the smaller images (frames) to calculate feature detection, description, and matching, and then calculate the corresponding homography and rescale it to the original scale. The benefit is obvious: calculation on lower resolution reduces the processing time greatly. Hence, I did the following three changes:

1. Include a down_sample_factor input argument to specify the factor of down-sampling. Down-sample each of the frames before calculating feature detection, descriptor, matching, and homography. After calculating the homography, rescale it back to the original scale.

2. Since I have the change of scaling, and the original implementation of FAST detector and BRIEF descriptor are not scale-invariant, I choose to replace them with Opencv built-in ORB detector and descriptor, which has both scale-invariance and rotation-invariance.

3. As I have changed to the ORB detector and descriptor, the matching method should also be changed. According to this discussion and OpenCV website, I would choose FLANN based Matcher over BF-Matcher.

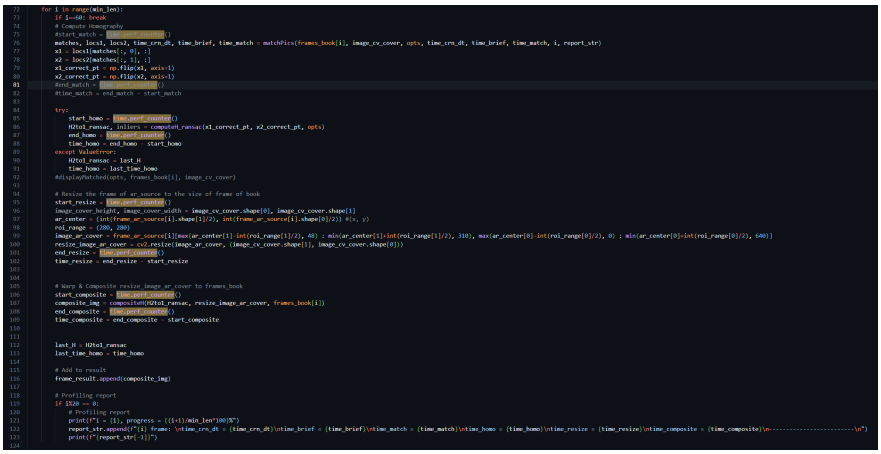


Figure 21: Q3.2 Profiling Code Snippet

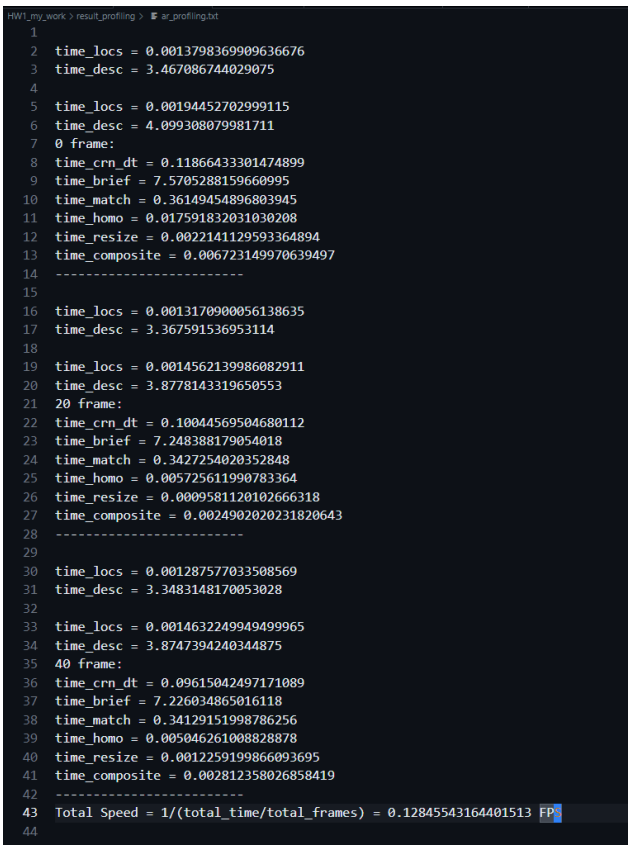


Figure 22: Q3.2 Profiling Result

Figure 23 shows the code snippet of change 1 occuring in ec/matchPics.py matchPics(), which I down-sample the two gray images before calculating the feature points. Figure 24 shows the code snippet of change 1 occuring in ec/planarH.py computeH_ransac(), which I scale back the calculated homography to the original scale by multiplying the down_sample_factor on the translation part of the homography, and leaves other parts unchanged as other parts do not change due to down-sampling.

```
# Scale down the image
gray_I1 = cv2.resize(gray_I1, (gray_I1.shape[1] // down_sample_factor, gray_I1.shape[0] // down_sample_factor))
gray_I2 = cv2.resize(gray_I2, (gray_I2.shape[1] // down_sample_factor, gray_I2.shape[0] // down_sample_factor))


#----------------------------Corner Detection--------------------------------#
```

Figure 23: Q3.2 Change 1 Code Snippet 1

```
    if down_sample_factor > 1:
        bestH2to1[0, 2] *= down_sample_factor
        bestH2to1[1, 2] *= down_sample_factor

    #test
```

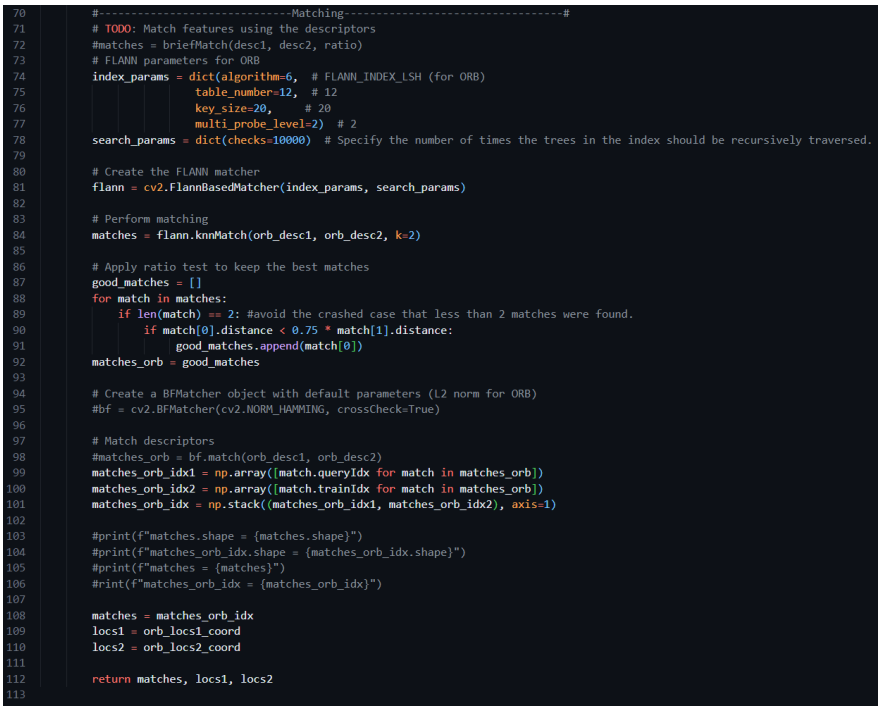
Figure 24: Q3.2 Change 1 Code Snippet 2

Figure 25 shows the change 2 occuring in ec/matchPics.py matchPics(), where I call the Opencv built-in ORB function to find the corresponding detectors and descriptors.

```
    #-----------------------------Corner Detection--------------------------------#

    # TODO: Detect Features in Both Images
    #locs1 = corner_detection(gray_I1, sigma);
    #locs2 = corner_detection(gray_I2, sigma);

    orb = cv2.ORB_create(nfeatures=20000)
    orb_locs1, orb_desc1 = orb.detectAndCompute(gray_I1, None);
    orb_locs2, orb_desc2 = orb.detectAndCompute(gray_I2, None);

    orb_locs1_coord = np.array([(int(x.pt[1]), int(x.pt[0])) for x in orb_locs1])
    orb_locs2_coord = np.array([(int(x.pt[1]), int(x.pt[0])) for x in orb_locs2])

    #print(f"locs1 = {locs1}")
    #print(f"locs2 = {locs2}")
    #print(f"locs1.shape = {locs1.shape}")
    #print(f"locs2.shape = {locs2.shape}")

    #print(f"orb_locs1_coord = {orb_locs1_coord}")
    #print(f"orb_locs2_coord = {orb_locs2_coord}")
    #print(f"orb_locs1_coord.shape = {orb_locs1_coord.shape}")
    #print(f"orb_locs2_coord.shape = {orb_locs2_coord.shape}")

    #-----------------------------Descriptor---------------------------------#
    # TODO: Obtain descriptors for the computed feature locations
    #desc1, locs1 = computeBrief(gray_I1, locs1)
    #desc2, locs2 = computeBrief(gray_I2, locs2)
```

Figure 25: Q3.2 Change 2 Code Snippet

Figure 26 shows the change 3 occuring in ec/matchPics.py matchPics(), where I call the Opencv built-in ORB matching function, called FLANN based Matcher, to find the corresponding matching feature points.



```
    #-----------------------------Matching--------------------------------#
    # TODO: Match features using the descriptors
    #matches = briefMatch(desc1, desc2, ratio)
    # FLANN parameters for ORB
    index_params = dict(algorithm=6,  # FLANN_INDEX_LSH (for ORB)
                        table_number=12,  # 12
                        key_size=20,     # 20
                        multi_probe_level=2)  # 2
    search_params = dict(checks=10000)  # Specify the number of times the trees in the index should be recursively traversed.

    # Create the FLANN matcher
    flann = cv2.FlannBasedMatcher(index_params, search_params)

    # Perform matching
    matches = flann.knnMatch(orb_desc1, orb_desc2, k=2)

    # Apply ratio test to keep the best matches
    good_matches = []
    for match in matches:
        if len(match) == 2: #avoid the crashed case that less than 2 matches were found.
            if match[0].distance < 0.75 * match[1].distance:
                good_matches.append(match[0])
    matches_orb = good_matches

    # Create a BFMatcher object with default parameters (L2 norm for ORB)
    #bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

    # Match descriptors
    #matches_orb = bf.match(orb_desc1, orb_desc2)
    matches_orb_idx1 = np.array([match.queryIdx for match in matches_orb])
    matches_orb_idx2 = np.array([match.trainIdx for match in matches_orb])
    matches_orb_idx = np.stack((matches_orb_idx1, matches_orb_idx2), axis=1)

    #print(f"matches.shape = {matches.shape}")
    #print(f"matches_orb_idx.shape = {matches_orb_idx.shape}")
    #print(f"matches = {matches}")
    #rint(f"matches_orb_idx = {matches_orb_idx}")

    matches = matches_orb_idx
    locs1 = orb_locs1_coord
    locs2 = orb_locs2_coord

    return matches, locs1, locs2
```

Figure 26: Q3.2 Change 3 Code Snippet

Figure 28 - Figure 30 show the code snippet of the main function in ec/ar_ec.py. One can refer to ec/README.md for the command to run the real-time AR application. The final FPS will be automatically reported on the standard output and the report willl be stored under the folder specified by –output_file argument. The resulting video will also be stored in the same output folder. The final performance un-

der my environment is nearly 31 FPS as shown in Figure 27, which is 242 times faster over the original implementation with 0.128 FPS shown in Figure 22.

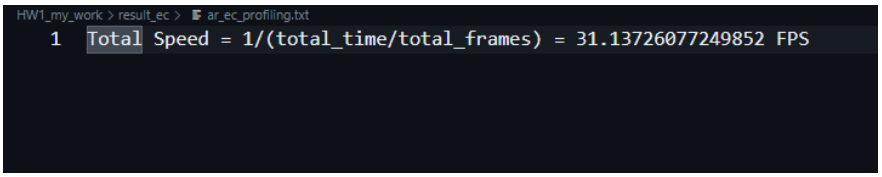


Figure 27: Q3.2 ar_ec.py Final Speed

```python
import numpy as np
import cv2

# Import necessary functions

from loadVid import loadVid


# Q3.2

#Import necessary functions
from opts import get_opts
from typing import List, Any, Tuple
from displayMatch import displayMatched
from planarH import compositeH
from matchPics import matchPics
from planarH   import computeH_ransac
import os
import time
import shutil

#Write script for Q3.1

########################
#     Main-Routine     #
########################
def main():
    # Input arguments/Load input videos
    opts = get_opts()
    out_folder = os.path.dirname(opts.output_file)
    if not os.path.exists(out_folder):
        os.makedirs(out_folder)
    else:
        shutil.rmtree(out_folder)
        os.makedirs(out_folder)

    frames_ar_source = loadVid(opts.input_src_file)
    frames_book      = loadVid(opts.input_dst_file)

    # Process each frame
    frame_result = generalHarryPoterize(frames_ar_source, frames_book, opts, out_folder)

    # Wriet out video
    writeOutVideo(frame_result, opts)

    # Wriet out three distinct timestamps
    #frameidx_book_left  = 123
    #frameidx_book_cnter = 230
    #frameidx_book_right = 331
    #cv2.imwrite(f"{out_folder}/Q3_1_ar_result_overlay_left.jpg", frame_result[frameidx_book_left])
    #cv2.imwrite(f"{out_folder}/Q3_1_ar_result_overlay_right.jpg", frame_result[frameidx_book_right])
    #cv2.imwrite(f"{out_folder}/Q3_1_ar_result_overlay_cnter.jpg", frame_result[frameidx_book_cnter])

########################
#     Sub-Routine      #
########################
```

Figure 28: Q3.2 ar_ec.py Code Snippet 1

```python
def writeOutVideo(frame_result: List[Any], opts) -> None:
    fourcc = cv2.VideoWriter_fourcc(*'XVID')
    frame_size = (frame_result[0].shape[1], frame_result[0].shape[0])
    out = cv2.VideoWriter(opts.output_file, fourcc, 30.0, frame_size)

    for frame in frame_result:
        out.write(frame)

def generalHarryPoterize(frame_ar_source: List[Any], frames_book: List[Any], opts, out_folder: str) -> List[Any]:
    image_cv_cover      = cv2.imread('../data/cv_cover.jpg')
    min_len             = min(len(frame_ar_source), len(frames_book))
    frame_result        = []
    down_sample_factor  = opts.down_sample_factor
    frame_interval_same_H = opts.frame_interval_same
    total_time          = 0

    for i in range(min_len):
        time_loop_start = time.perf_counter()
        # Start of the Computation
        # Compute Homography
        if i%frame_interval_same_H == 0:
            matches, locs1, locs2 = matchPics(frames_book[i], image_cv_cover, opts, down_sample_factor)
            x1 = locs1[matches[:, 0], :]
            x2 = locs2[matches[:, 1], :]
            x1_correct_pt = np.flip(x1, axis=1)
            x2_correct_pt = np.flip(x2, axis=1)

            try:
                H2to1_ransac, inliers = computeH_ransac(x1_correct_pt, x2_correct_pt, opts, True, 10, down_sample_factor)
            except ValueError:
                H2to1_ransac = last_H
        else:
            H2to1_ransac = last_H

        # Resize the frame of ar_source to the size of frame of book
        image_cover_height, image_cover_width = image_cv_cover.shape[0], image_cv_cover.shape[1]
        ar_center = (int(frame_ar_source[i].shape[1]/2), int(frame_ar_source[i].shape[0]/2)) #(x, y)
        roi_range = (280, 280)

        image_ar_cover = frame_ar_source[i][max(ar_center[1]-int(roi_range[1]/2), 48) : min(ar_center[1]+int(roi_range[1]/2), 310), max(ar_center[0]-int(roi_range[0]/2), 0) : min(ar_center[0]+int(roi_range[0]/2), 640)]
        resize_image_ar_cover = cv2.resize(image_ar_cover, (image_cv_cover.shape[1], image_cv_cover.shape[0]))

        # Warp & Composite resize_image_ar_cover to frames_book
        composite_img = compositeH(H2to1_ransac, resize_image_ar_cover, frames_book[i])

        # End of the computation
        time_loop_end = time.perf_counter()
        total_time += (time_loop_end-time_loop_start)


        # Add to result
        frame_result.append(composite_img)
        last_H = H2to1_ransac

        cv2.imshow("composite_img", composite_img)
        cv2.waitKey(1)
        #cv2.destroyAllWindows()
```

Figure 29: Q3.2 ar_ec.py Code Snippet 2

```python
            # Stop if 'q' is pressed
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break

    result_fps = f"Total Speed = 1/(total_time/total_frames) = {1/(total_time/min_len)} FPS"
    print(result_fps)
    with open(f"{out_folder}/ar_ec_profiling.txt", "w") as f:
        f.write(result_fps)
    return frame_result

#---------------Execution---------------#
if __name__ == '__main__':
    main()
```

Figure 30: Q3.2 ar_ec.py Code Snippet 3

There is another input argument that I originally developed to further speedup, but I currently do not use. It is the –frame_interval_same. It is used to bypass some frames that can use the same homography as previous frame. The reason that one can do it is to assume that the targeted frames will not have large movement over 'frame_interval_same' frames. Under this assumption, the mapping of homography can be deemed as nearly unchanged. The default value of this argument is set to 2 as I think that data/book.mov does not have big motion between continuous frames. Although I can use it to further speedup my performance, which may be up to 50 FPS, I do not turn it on by setting this argument to 1 , like the one specified in ec/README.md, because I have already reached the performance over 30 FPS without the usage of this argument.

Q4 at page 10

Ans:
In