16-720A — Spring 2021 — Homework 2

Jen-Hung Ho
jenhungh@andrew.cmu.edu

March 10, 2021

Question 1

Q1.1

The 3 x 4 camera projection matrices $\mathbf{P}_1$ and $\mathbf{P}_2$ project the homogeneous 3D point $\mathbf{X}_\pi = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$ in world frame to the homogeneous 2D point in each camera frame $\mathbf{x}_1 = \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$ and $\mathbf{x}_2 = \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$. The equations are as follows:

$$\lambda_1 \mathbf{x_1} = \mathbf{P_1 X_\pi} \tag{1}$$
$$\lambda_2 \mathbf{x_2} = \mathbf{P_2 X_\pi} \tag{2}$$

We also know that $\mathbf{X}_\pi$ is in plane π. Hence, we can project $\mathbf{X}_\pi$ to 2D homogeneous coordinates on plane π.

$$\lambda_1 \mathbf{x_1} = \mathbf{P_1 X_\pi} = \mathbf{P_1} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \mathbf{P_1} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & Z \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = \mathbf{P_1 (\pi x_\pi) = H_1 x_\pi} \tag{3}$$

$$\lambda_2 \mathbf{x_2} = \mathbf{P_2 X_\pi} = \mathbf{P_2} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \mathbf{P_2} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & Z \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = \mathbf{P_2 (\pi x_\pi) = H_2 x_\pi} \tag{4}$$

where $\mathbf{H}_1$ and $\mathbf{H}_2$ are 3 x 3 matrices. From equation 3 and 4, we know that:

$$\mathbf{x_\pi} = \lambda_1 \mathbf{H_1^{-1} x_1} = \lambda_2 \mathbf{H_2^{-1} x_2} \tag{5}$$

Rearrange equation 5 to get:

$$\mathbf{x_1} = \frac{\lambda_2}{\lambda_1} \mathbf{H_1 H_2^{-1} x_2} = \frac{\lambda_2}{\lambda_1} \mathbf{H x_2} \tag{6}$$

Since both $\mathbf{H}_1$ and $\mathbf{H}_2$ are 3 x 3 matrix and are invertible, we prove that the Homography Matrix $\mathbf{H}$ exists.

$$\boldsymbol{x_1} \equiv \mathbf{H x_2} \tag{7}$$

Q1.2

1.

The Homography Matrix **H** is a 3 x 3 matrix and thus there are 9 elements in **H**. However, since $\mathbf{x}_1$ and $\mathbf{x}_2$ are in homogeneous coordinates, **H** has no relation with the scale. Hence, **h** has **8 degrees of freedom**.

2.

Each point pair provides two constraints, and **h** has 8 degrees of freedom. Thus, we need **four** point pairs to solve **h**.

3.

$$\begin{bmatrix} x_1^i \\ y_1^i \\ 1 \end{bmatrix} \equiv \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & H_{33} \end{bmatrix} \begin{bmatrix} x_2^i \\ y_2^i \\ 1 \end{bmatrix} \tag{8}$$

From the third row, we know that:

$$1 = x_2^i H_{31} + y_2^i H_{32} + H_{33} \tag{9}$$

We can get two constraints from a corresponding point pair and thus derive $\mathbf{A}_i$

$$x_1^i = \frac{x_2^i H_{11} + y_2^i H_{12} + H_{13}}{x_2^i H_{31} + y_2^i H_{32} + H_{33}} \tag{10}$$

$$y_1^i = \frac{x_2^i H_{21} + y_2^i H_{22} + H_{23}}{x_2^i H_{31} + y_2^i H_{32} + H_{33}} \tag{11}$$

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

4.

The trivial solution for h is $\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$ and A matrix is NOT full-rank since the maximum rank of A is 8 (There exists a trivial solution for h). If we have more than four corresponding point pairs, then the system is over-determined (more constraints than variables).

Q1.3

For Camera 1 and 2, we know that

$$\mathbf{x_1} = \mathbf{K_1} \begin{bmatrix} \mathbf{I} & 0 \end{bmatrix} \mathbf{X} \tag{13}$$

$$\mathbf{x_2} = \mathbf{K_2} \begin{bmatrix} \mathbf{R} & 0 \end{bmatrix} \mathbf{X} \tag{14}$$

Move the intrinsic matrices to left side and change $\mathbf{X}$ into inhomogeneous coordinates.

$$\mathbf{K_1^{-1}x_1} = \mathbf{X} \tag{15}$$

$$\mathbf{K_2^{-1}x_2} = \mathbf{RX} \tag{16}$$

Put equation 18 into 19, we can get

$$\mathbf{K_2^{-1}x_2} = \mathbf{R}(\mathbf{K_1^{-1}x_1}) \tag{17}$$

$$\mathbf{x_1} = (\mathbf{K_1R^{-1}K_2^{-1}})\mathbf{x_2} = \mathbf{Hx_2} \tag{18}$$

The intrinsic matrix and the rotation matrix are invertible, so the Homography Matrix $\mathbf{H}$ eexists.

Q1.4

From Q1.3, we know that

$$\mathbf{x_1} = \mathbf{H_\theta x_2} = (\mathbf{KR_\theta^{-1}K^{-1}})\mathbf{x_2} \tag{19}$$

Change the rotation from θ to 2θ, then

$$\mathbf{x_1} = \mathbf{H_{2\theta}x_2} = (\mathbf{KR_{2\theta}^{-1}K^{-1}})\mathbf{x_2} \tag{20}$$

$$\mathbf{H_{2\theta}} = \mathbf{KR_{2\theta}K^{-1}} = (\mathbf{KR_\theta K^{-1}})(\mathbf{KR_\theta K^{-1}}) = \mathbf{H_\theta H_\theta} = \mathbf{H_\theta^2} \tag{21}$$

Q1.5

Planar Homography can handle the transformation of two viewpoints if the camera motion is a pure rotation. However, if there is a translation between two viewpoints, Planar Homography might not be sufficient to solve the mapping between them. For example, if you not only rotate the camera but also change the positions of the camera, it is impossible to map the viewpoints by using Planar Homography.

Q1.6

The perspective projection equation is

$$\mathbf{x} = \mathbf{PX} \tag{22}$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \frac{1}{Z} \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{23}$$

where f is the focal length. Assume that a line in 3D coordinate is

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} X_0 \\ Y_0 \\ Z_0 \end{bmatrix} + \begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix} t \tag{24}$$

Project the 3D line into a 2D plane:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \frac{1}{Z_0 + V_z t} \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_0 + V_x t \\ Y_0 + V_y t \\ Z_0 + V_z t \end{bmatrix} \tag{25}$$

$$u = \frac{f(X_0 + V_x t)}{Z_0 + V_z t} \tag{26}$$

$$v = \frac{f(Y_0 + V_y t)}{Z_0 + V_z t} \tag{27}$$

Modify equation 26 and 27, we can get:

$$u = \frac{fV_x}{V_z} + \frac{f(XV_z - ZV_x)}{V_z^2}\frac{V_z}{Z_0 + V_z t} = u_0 + V_u s \tag{28}$$

$$v = \frac{fV_y}{V_z} + \frac{f(YV_z - ZV_y)}{V_z^2}\frac{V_z}{Z_0 + V_z t} = v_0 + V_v s \tag{29}$$

From equation 28 and 29, we know that the projection of a 3D line is a 2D line, which verify that the Projection Matrix $\mathbf{P}$ preserves lines.

Question 2

Q2.1.1

The Harris Corner Detector uses sliding windows to find the corners by applying the first-order approximation (Matrix **A**), while the FAST (Features from Accelerated Segment Test) Detector finds the corners by comparing the difference of brightness of the surrounding pixels (16 pixels from a circle with radius = 3). The computational performance of the FAST Detector is better than the Harris Corner Detector since it doesn't need to compute the derivative with respect to x and y. Moreover, in FAST Detector, we could also implement the high-speed test (only check pixel 1, 5, 9, and 13) to reduce the pixel we need to check and hence increase the computational efficiency.

Q2.1.2

BRIEF (Binary Robust Independent Elementary Features) Descriptor describes a feature point by randomly selecting point pairs around it, compare their pixel values, and transform a point pair into a boolean value to form a binary array. We can also use filter banks such as Gabor Filters that Professor mentioned in the lectures to describe feature points (ex. GIFT Descriptor). The filter responses of each filter in the filter bank can detect edges and thus form a descriptor that describes the feature points.

Q2.1.3

The definition of Hamming distance is the number of different elements between two equal-length strings. It is mainly used in the binary array to compute the bit difference. Computing the Hamming distance of two descriptors is extremely fast by using XOR or bit count, which provides computational benefits over Euclidean distance. Nearest Neighbor is a brute force matching method that matches the corresponding point based on the distance between two points. The closer two points are (They are nearest neighbors), the more likely they are matching points.

Q2.1.4

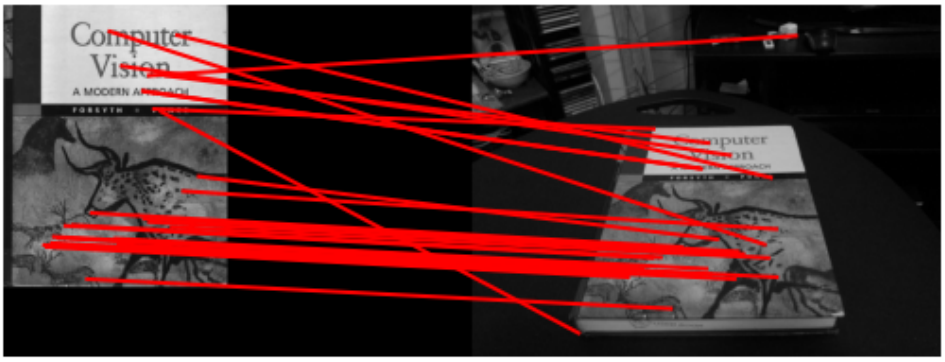

Figure 1: Feature Matching of cv_cover.png and cv_desk.png (ratio = 0.7, sigma = 0.15)

Q2.1.5

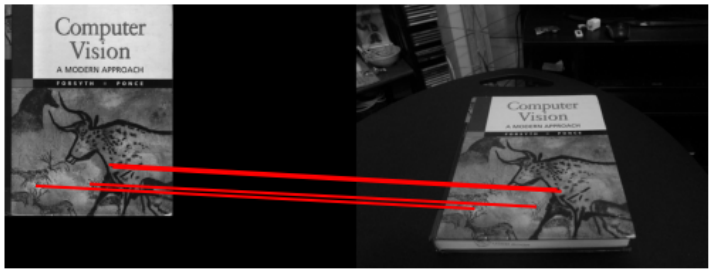

Figure 2: Feature Matching of cv_cover.png and cv_desk.png with ratio = 0.5

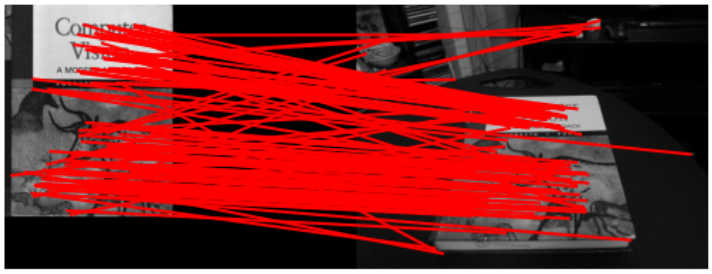

Figure 3: Feature Matching of cv_cover.png and cv_desk.png with ratio = 0.9

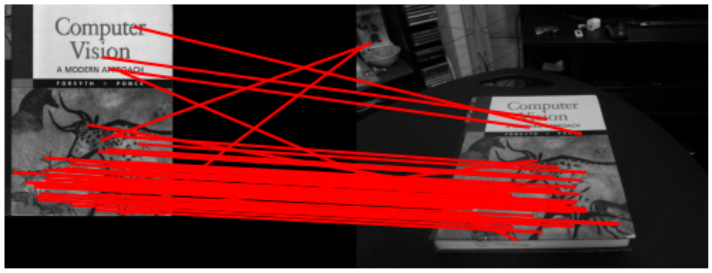

Figure 4: Feature Matching of cv_cover.png and cv_desk.png with sigma = 0.1

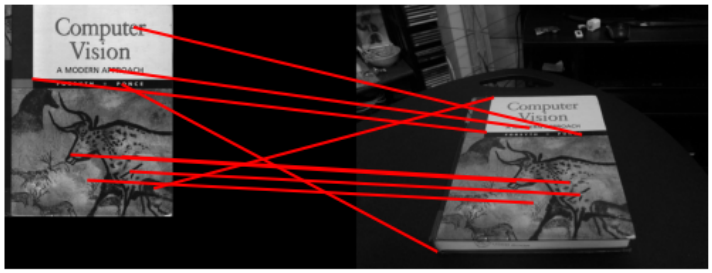

Figure 5: Feature Matching of cv_cover.png and cv_desk.png with sigma = 0.2

Figure 2 and 3 show the ablation study of **ratio** in BRIEF descriptor matching. The definition of the max ratio is the division of the nearest neighbor distance to the second nearest neighbor distance. If the ratio of the matching pair is lower than the max ratio, then it is accepted. Hence, lower down the max ratio means that there should be fewer corresponding points but the stability of matching pairs should increase. Figure 2 has only a few matching points and most of them are correct, while Figure 3 has a lot of matching points but many of them are incorrect.

Figure 4 and 5 show the ablation study of **sigma** in FAST detector. The definition of sigma is the threshold of the matching points. A pixel is considered a corner if there are over n surrounding pixels whose difference of brightness is bigger than sigma. Therefore, lower down sigma means that there will be more features in both images and hence the matching pairs will increase. Figure 4 with low sigma has more matching pairs than Figure 3 with high sigma.

Q2.1.6

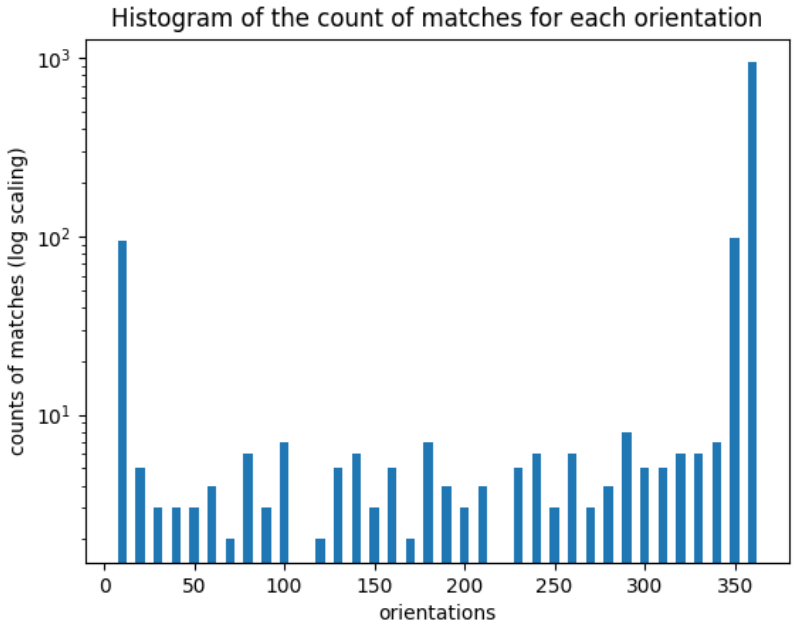

Figure 6: Histograms of the count of matching points for each orientation

angle = 10 degree, matching result = 94 (Figure 7)
angle = 100 degree, matching result = 7 (Figure 8)
angle = 200 degree, matching result = 3 (Figure 9)
angle = 300 degree, matching result = 5
angle = 350 degree, matching result = 98 (Figure 10)
angle = 360 degree, matching result = 945

The BRIEF descriptor is NOT designed to be rotation invariant. By **randomly selecting** surrounding point pairs, comparing their pixel values, and transforming a point pair into a boolean value to form a binary array, the BRIEF descriptor has high computational efficiency. However, this also makes it lose the property of rotation invariant. From figure 6 we can find that the number of matching points decreases significantly if the rotation angle is bigger than 10 degrees.

Figure 7: Feature Matching of cv_cover.png with rotation angle = 10 degree

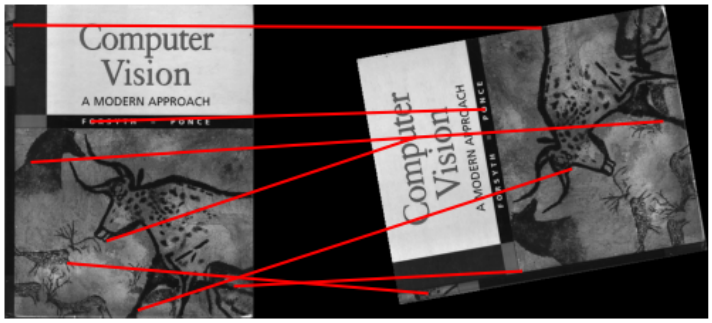

Figure 8: Feature Matching of cv_cover.png with rotation angle = 100 degree

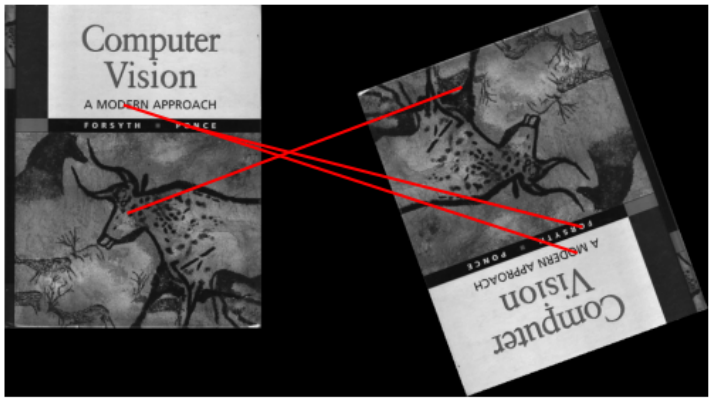

Figure 9: Feature Matching of cv_cover.png with rotation angle = 200 degree

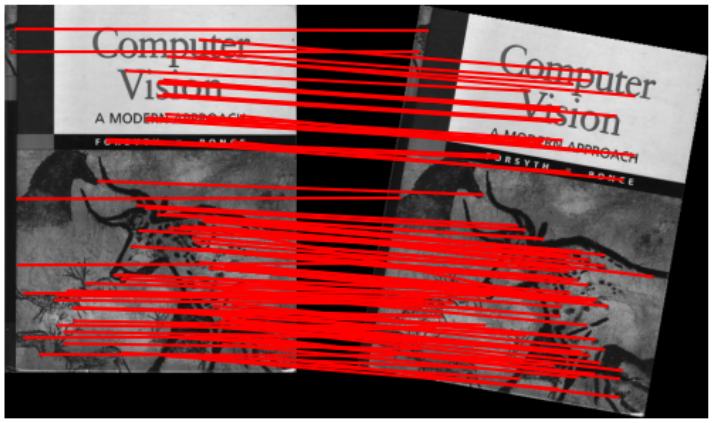

Figure 10: Feature Matching of cv_cover.png with rotation angle = 350 degree

Q2.2.4

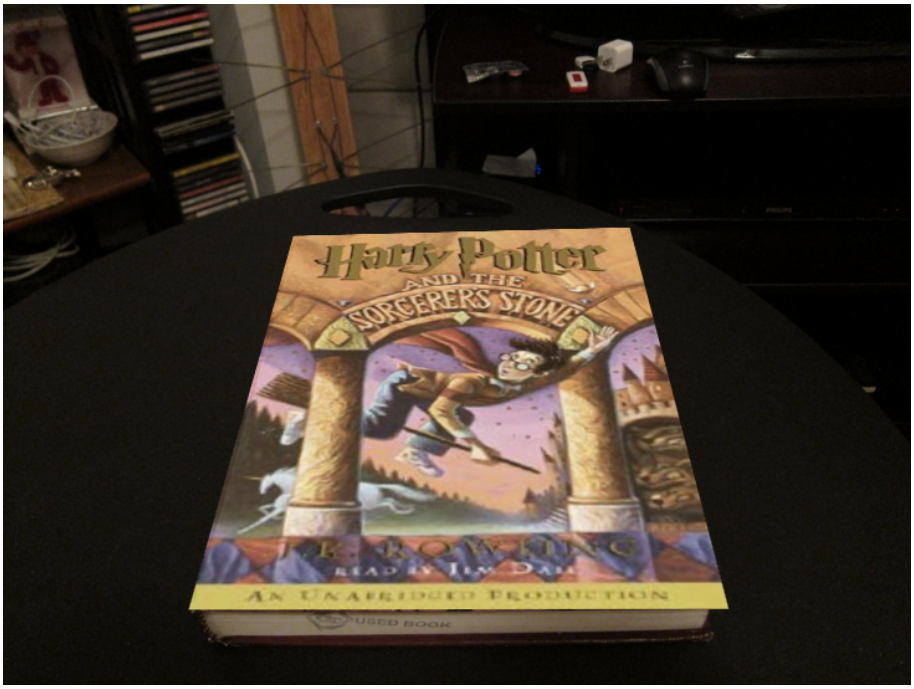

Figure 11: HarryPotterized Text book (max_iters = 500, inlier_tol = 2.0)

Q2.2.5

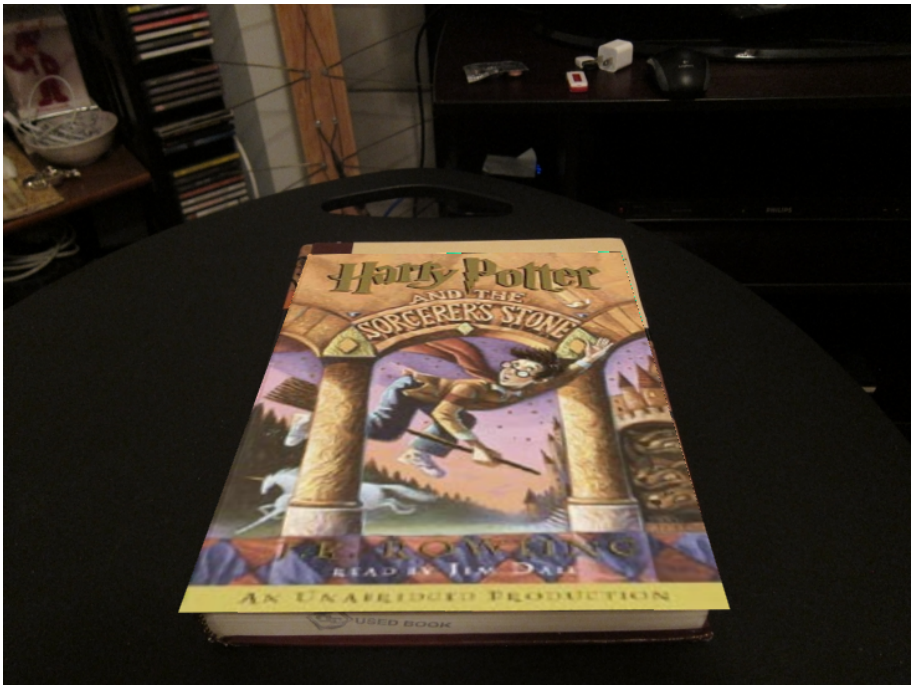

Figure 12: HarryPotterized Text book (max_iters = 100)

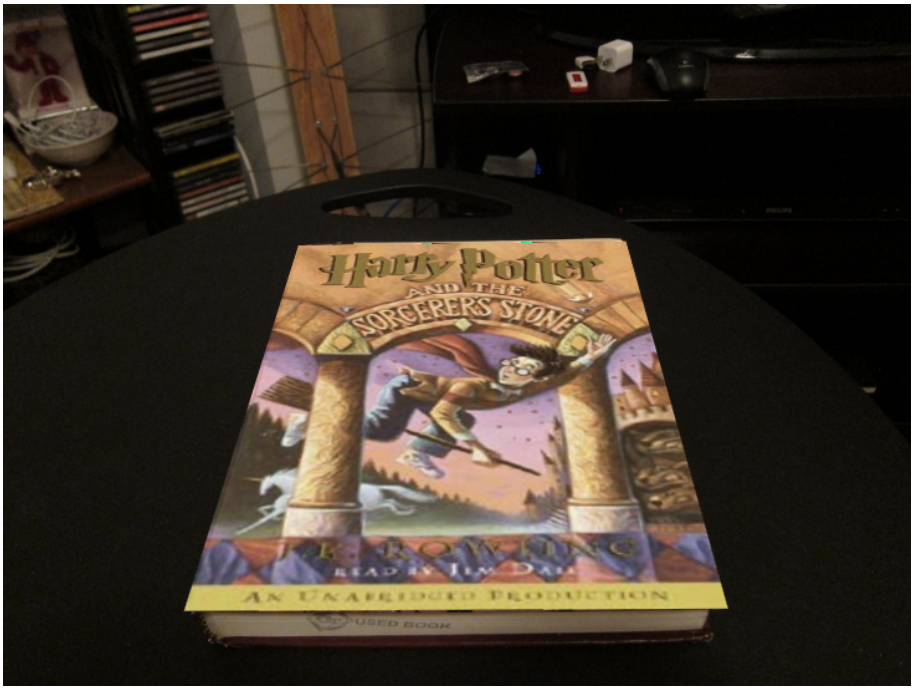

Figure 13: HarryPotterized Text book (max_iters = 1000)

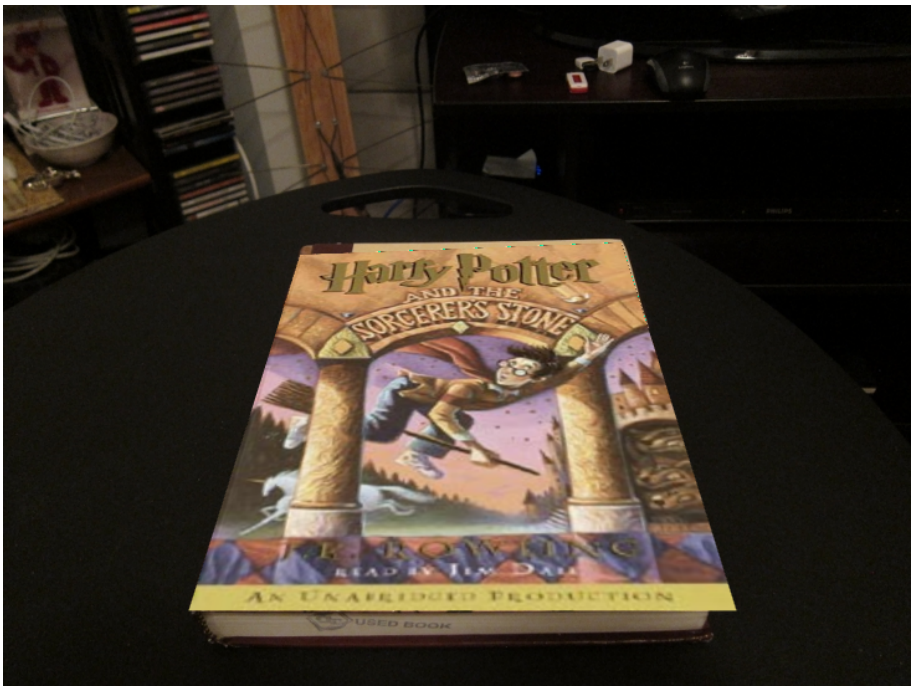

Figure 14: HarryPotterized Text book (inlier_tol = 1.0)

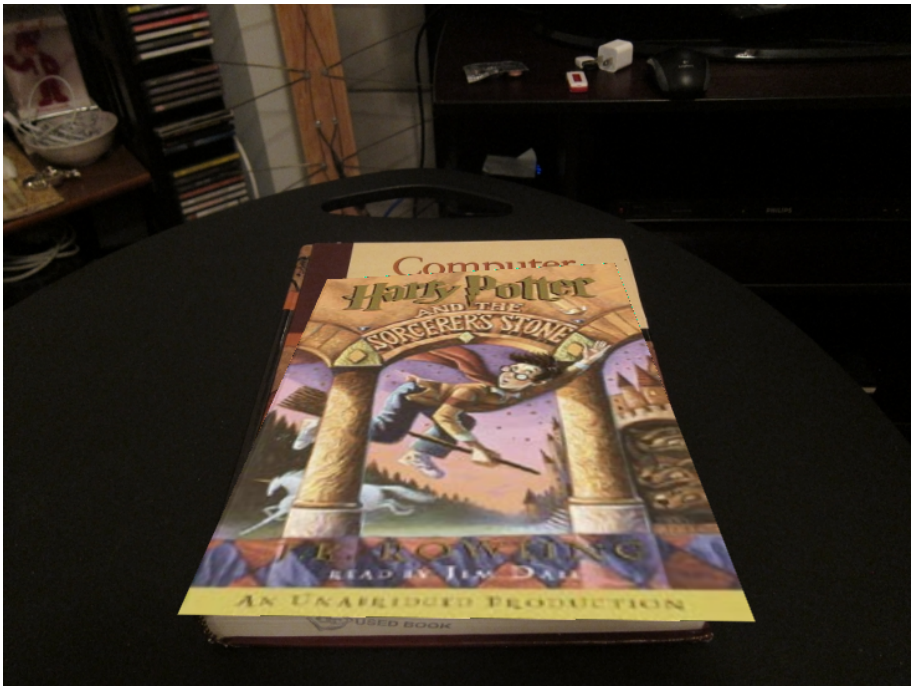

Figure 15: HarryPotterized Text book (inlier_tol = 5.0)

Figure 12 and 13 show the ablation study of **max_iters** in the RANSAC algorithm. Since RANSAC algorithm is an iterative method, it is important to make sure that the number of iterations is high enough to estimate the model parameters correctly. Figure 12 runs only 100 iterations and therefore the overlay is not as good as Figure 11, while Figure 13 runs 1000 iterations and the result is slightly better than Figure 11 (Since Figure 11 is already very good, it is not obvious.)

Figure 14 and 15 show the ablation study of **inlier_tol** in the RANSAC algorithm. The tolerance of the error affects the number of the inliers. Either too small or too large will affect the accuracy of the estimated output of RANSAC algorithm. Figure 14 and Figure 15 demonstrate that the default inlier_tol = 2.0 is already a good tolerance. Smaller tolerance (inlier_tol = 1.0) makes the result a little bit worse and larger tolerance (inlier_tol = 5.0) makes the performance of the overlay really bad.

Question 3

Q3.1

Figure 16: Screenshot of ar.avi (Overlay is near the center)

Figure 17: Screenshot of ar.avi (Overlay is near the left)

Figure 18: Screenshot of ar.avi (Overlay is near the right)

Extra Credit

Q4.1x

(a) left

(b) right

Figure 19: Original Image

Figure 20: Panorama

Code Appendix

main.py

```python
import cv2
from helper import plotMatches
from opts import get_opts

from ar_helper import matchPics, briefRotTest, composeWarpedImg
from planarH import computeH, computeH_norm, computeH_ransac


def main():
    opts = get_opts()

    cv_cover = cv2.imread('../data/cv_cover.jpg')
    cv_desk = cv2.imread('../data/cv_desk.png')
    hp_cover = cv2.imread('../data/hp_cover.jpg')

    # Q2.1.4
    matches, locs1, locs2 = matchPics(cv_cover, cv_desk, opts)
    # Display matched features
    plotMatches(cv_cover, cv_desk, matches, locs1, locs2)

    # Q2.1.6
    # briefRotTest(cv_cover, opts)

    # Check Q2.2.1, Q2.2.2, and Q2.2.3
    # Extract the matching points
    # locs1 = locs1[matches[:, 0]]
    # locs2 = locs2[matches[:, 1]]
    # Flip x and y coordinates
    # locs1[:, [0, 1]] = locs1[:, [1, 0]]
    # locs2[:, [0, 1]] = locs2[:, [1, 0]]

    # Q2.2.1
    # H2to1 = computeH(locs1, locs2)
    # print(f"H2to1 = {H2to1}")
    # Q2.2.2
    # H2to1 = computeH_norm(locs1, locs2)
    # print(f"H2to1 = {H2to1}")
    # Q2.2.3
    # best_H2to1, inliers = computeH_ransac(locs1, locs2, opts)
    # print(f"best_H2to1 = {best_H2to1}")
    # print(f"inliers = {inliers}")

    # Q2.2.4
    # composite_img = composeWarpedImg(cv_cover, cv_desk, hp_cover, opts)
```

```
45      # cv2.imwrite('../result/hp_desk.png', composite_img)
46
47
48  if __name__ == '__main__':
49      main()
```

ar.py

```python
# Import necessary functions
import numpy as np
import cv2
import skimage
import skimage.io
from loadVid import loadVid
import imageio
from opts import get_opts
from ar_helper import composeWarpedImg

opts = get_opts()


def main():
    # Load in necessary data
    src_frames = loadVid('../data/ar_source.mov')
    book_frames = loadVid('../data/book.mov')
    cv_cover = skimage.io.imread('../data/cv_cover.jpg')

    # Cut the zero padded region and convert to RGB
    src_frames = src_frames[:, 48:-48, :, ::-1]
    book_frames = book_frames[:, :, :, ::-1]
    size = len(src_frames)

    # Initialize composite frames
    composite_frames = []

    # Warp each frame
    for index in range(len(book_frames)):
        # Pad at the end of src_frames with the beginning of src_frames until
        # number of srcframes equals number of book_frames
        if index >= len(src_frames):
            append_frame = src_frames[index % size].reshape(1,
                src_frames.shape[1], src_frames.shape[2], src_frames.shape[3])
            src_frames = np.append(src_frames, append_frame, axis = 0)

        # Crop src_frames so that it has the same aspect ratio as the cv_cover
        src_frame = cv2.resize(src_frames[index], dsize = (cv_cover.shape[1],
                                                           cv_cover.shape[0]))

        # Change the ratio at 435th frame to make sure that there are enough
        # matching points
        if index == 435:
            opts.ratio = 0.8
        else:
            opts.ratio = 0.7
```

```
47
48          # Use ar_helper.composeWarpedImg to place each src_frame correctly over
49          # each book_frame, using cv_cover as a reference for warping
50          composite_frame = composeWarpedImg(cv_cover, book_frames[index],
51                                                          src_frame, opts)
52          # Check composite frame
53          # cv2.imshow('composite_frame', composite_frame)
54          # cv2.waitKey(1)
55
56          # Update composite frames
57          composite_frames.append(composite_frame)
58          print(f"Index {index} done.")
59
60      # save composite frames into a video, where composite_frames is a list of
61      # image arrays of shape (h, w, 3) obtained above
62      imageio.mimwrite('../result/ar.avi', composite_frames, fps=30)
63
64
65  if __name__ == '__main__':
66      main()
```

ar_helper.py

```python
from helper import briefMatch, computeBrief, detectCorners, convert2Gray,
                                                            plotMatches
from planarH import computeH_ransac

import numpy as np
import cv2
import matplotlib.pyplot as plt
import scipy


def matchPics(img1, img2, opts):
    # img1, img2 : Images to match
    # opts: input opts
    ratio = opts.ratio  # 'ratio for BRIEF feature descriptor'
    sigma = opts.sigma  # 'threshold for corner detection using FAST feature
                                                            # detector'

    # Convert Images to GrayScale
    img1 = convert2Gray(img1)
    img2 = convert2Gray(img2)

    # Detect Features in Both Images
    locs1 = detectCorners(img1, sigma)
    locs2 = detectCorners(img2, sigma)

    # Obtain descriptors for the computed feature locations
    desc1, locs1 = computeBrief(img1, locs1)
    desc2, locs2 = computeBrief(img2, locs2)

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

    return matches, locs1, locs2


# Q2.1.5
def briefRotTest(img, opts):
    # in increments of 10 degrees from 0 to 360
    match_per_angle = []
    for angle in range(10, 361, 10):
        # Rotate Image
        rotate_img = scipy.ndimage.rotate(img, angle)

        # Compute features, descriptors and Match features
        matches, locs1, locs2 = matchPics(img, rotate_img, opts)

```

```
        # Display four orientations (10, 100, 200, and 350 degrees)
        if (angle == 10 or angle == 100 or angle == 200 or angle == 350):
            plotMatches(img, rotate_img, matches, locs1, locs2)

        # Update histogram
        match_per_angle.append(len(matches))
        print(f"angle = {angle} degree, matching result = {len(matches)}")

    # Display histogram
    plt.bar([*range(10, 361, 10)], match_per_angle, width = 5, log = True)
    plt.title('Histogram of the count of matches for each orientation')
    plt.xlabel('orientations')
    plt.ylabel('counts of matches (log scaling)')
    plt.show()


def composeWarpedImg(img_source, img_target, img_replacement, opts):
    # Obtain the homography that warps img_source to img_target, then use it to
    # overlay img_replacement over img_target
    # Obtain features from both source and target images
    matches, locs1, locs2 = matchPics(img_source, img_target, opts)

    # Extract the matching points
    locs1 = locs1[matches[:, 0]]
    locs2 = locs2[matches[:, 1]]
    # Flip x and y coordinates
    locs1[:, [0, 1]] = locs1[:, [1, 0]]
    locs2[:, [0, 1]] = locs2[:, [1, 0]]

    # Get homography by RANSAC
    H2to1, inliers = computeH_ransac(locs1, locs2, opts)
    H1to2 = np.linalg.inv(H2to1)

    # Create a composite image after warping the replacement image on top of the
    # target image using the homography
    # Scale the replacement image properly
    img_replacement = cv2.resize(img_replacement, dsize = (img_source.shape[1],
                                                           img_source.shape[0]))

    # Warp the replacement image
    hp_warp = cv2.warpPerspective(img_replacement, H1to2, dsize =
                                      (img_target.shape[1], img_target.shape[0]))
    # Check the replacement warpping
    # cv2.imshow('hp_warp', hp_warp)
    # cv2.waitKey(0)

    # Delete the target areas for composition
    delete_warp = cv2.warpPerspective(np.ones_like(img_replacement), H1to2,
```

```
95                              dsize = (img_target.shape[1], img_target.shape[0]))
96      img_target_area_deleted = img_target - img_target * delete_warp
97      # Check the image with target area deleted
98      # cv2.imshow('img_target_area_deleted', img_target_area_deleted)
99      # cv2.waitKey(0)
100
101     # Composite the image
102     composite_img = img_target_area_deleted + hp_warp
103
104     return composite_img
```

planarH.py

```python
import numpy as np

def computeH(locs1, locs2):
    # Q2.2.1
    # Compute the homography between two sets of points
    # Form Matrix A to solve for h
    # Initialize the size of A
    num_match = locs1.shape[0]
    A = np.zeros((2*num_match, 9))

    # A corresponding pair provides two constraints
    for index in range(num_match):
        # Extract corresponding points (x1, y1) and (x2, y2)
        x1, y1 = locs1[index, :]
        x2, y2 = locs2[index, :]
        # Form A based on Q1.2(3)
        A[2*index, :] = [-x2, -y2, -1, 0, 0, 0, x1*x2, x1*y2, x1]
        A[2*index+1, :] = [0, 0, 0, -x2, -y2, -1, y1*x2, y1*y2, y1]

    # Apply SVD to solve for h
    u, s, vh = np.linalg.svd(A)
    h = vh.T[:, -1]

    # Change h into H
    H2to1 = h.reshape(3,3)

    return H2to1


def computeH_norm(locs1, locs2):
    # Q2.2.2
    # Compute the centroid of the points
    locs1_mean = np.mean(locs1, axis = 0)
    locs2_mean = np.mean(locs2, axis = 0)

    # Shift the origin of the points to the centroid
    locs1 = locs1 - locs1_mean
    locs2 = locs2 - locs2_mean

    # Normalize the points so that the largest distance from the origin is equal
    # to sqrt(2)
    max_dis1 = np.max(np.sqrt(np.sum(locs1*locs1, axis = 1)))
    locs1 = (np.sqrt(2) / max_dis1) * locs1
    max_dis2 = np.max(np.sqrt(np.sum(locs2*locs2, axis = 1)))
    locs2 = (np.sqrt(2) / max_dis2) * locs2

```

```
    # Similarity transform 1
    T1 = np.array([[(np.sqrt(2) / max_dis1), 0, -(np.sqrt(2) / max_dis1) *
                                                    locs1_mean[0]],
                   [0, (np.sqrt(2) / max_dis1), -(np.sqrt(2) / max_dis1) *
                                                    locs1_mean[1]],
                   [0, 0, 1]])

    # Similarity transform 2
    T2 = np.array([[(np.sqrt(2) / max_dis2), 0, -(np.sqrt(2) / max_dis2) *
                                                    locs2_mean[0]],
                   [0, (np.sqrt(2) / max_dis2), -(np.sqrt(2) / max_dis2) *
                                                    locs2_mean[1]],
                   [0, 0, 1]])

    # Compute homography using normalized locs
    H2to1_norm = computeH(locs1, locs2)

    # Denormalization
    H2to1 = np.dot(np.dot(np.linalg.inv(T1), H2to1_norm), T2)

    return H2to1


def computeH_ransac(locs1, locs2, opts):
    # Q2.2.3
    # Compute the best fitting homography given a list of matching points
    # the number of iterations to run RANSAC for
    max_iters = opts.max_iters
    # the tolerance value for considering a point to be an inlier
    inlier_tol = opts.inlier_tol

    # Get the number of matching points
    num_match = locs1.shape[0]
    print(f"num of matches = {num_match}")

    # Change locs1 and ocs2 into Homogeneous Coordinates : for error checking
    locs1_homo = np.hstack((locs1, np.ones((num_match, 1))))
    locs2_homo = np.hstack((locs2, np.ones((num_match, 1))))

    # Initialize max inliers
    max_inliers = 0

    # RANSAC Algorithm
    for iter in range(max_iters):
        # Randomly select 4 point pairs to compute H
        np.random.seed()
        sample = np.random.choice(num_match, size = 4, replace = False)
        locs1_sample = locs1[sample]
```

```
        locs2_sample = locs2[sample]
        H = computeH_norm(locs1_sample, locs2_sample)

        # Compute the error of predicted locs1
        # Compute predicted locs1 (Dimension: 3*N)
        locs1_pred = np.dot(H, locs2_homo.T)
        # Normalize predicted locs1 (Dimension: N*3)
        locs1_pred = locs1_pred.T / locs1_pred.T[:, -1].reshape(num_match, 1)
        # Compute the error and the euclidean distance
        error = locs1_homo - locs1_pred
        dis = np.sqrt(np.sum(error*error, axis = 1))

        # Initialize and Update current inliers
        current_inliers = np.zeros((num_match, 1))
        current_inliers[sample] = 1
        current_inliers[dis < inlier_tol] = 1
        # Compute the number of inliers
        num_inliers = np.sum(current_inliers)

        # Update best H and inliers if needed
        if (num_inliers > max_inliers):
            max_inliers = num_inliers
            best_H2to1 = H
            inliers = current_inliers

    return best_H2to1, inliers
```

helper.py

```python
import numpy as np
import cv2
from matplotlib import pyplot as plt
import skimage.feature

PATCHWIDTH = 9


def convert2Gray(img):
    # Convert image to grayscale
    if len(img.shape) == 3:
        img_gray = skimage.color.rgb2gray(img)
    else:
        img_gray = img.astype(np.float64) / 255.0

    return img_gray


def briefMatch(desc1, desc2, ratio):

    matches = skimage.feature.match_descriptors(
        desc1, desc2, 'hamming', cross_check=True, max_ratio=ratio)
    return matches


def plotMatches(img1, img2, matches, locs1, locs2):
    fig, ax = plt.subplots(nrows=1, ncols=1)
    img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
    plt.axis('off')
    skimage.feature.plot_matches(
        ax, img1, img2, locs1, locs2, matches, matches_color='r',
                                                        only_matches=True)
    plt.show()
    return


def makeTestPattern(patchWidth, nbits):
    np.random.seed(0)
    compareX = patchWidth*patchWidth * np.random.random((nbits, 1))
    compareX = np.floor(compareX).astype(int)
    np.random.seed(1)
    compareY = patchWidth*patchWidth * np.random.random((nbits, 1))
    compareY = np.floor(compareY).astype(int)

    return (compareX, compareY)
```

```
47
48
49  def computePixel(img, idx1, idx2, width, center):
50      halfWidth = width // 2
51      col1 = idx1 % width - halfWidth
52      row1 = idx1 // width - halfWidth
53      col2 = idx2 % width - halfWidth
54      row2 = idx2 // width - halfWidth
55      return 1 if img[int(center[0]+row1)][int(center[1]+col1)] <
56                              img[int(center[0]+row2)][int(center[1]+col2)] else 0
57
58
59  def computeBrief(img, locs):
60      patchWidth = 9
61      nbits = 256
62      compareX, compareY = makeTestPattern(patchWidth, nbits)
63      m, n = img.shape
64
65      halfWidth = patchWidth//2
66
67      locs = np.array(list(filter(lambda x: halfWidth <=
68                  x[0] < m-halfWidth and halfWidth <= x[1] < n-halfWidth, locs)))
69      desc = np.array([list(map(lambda x: computePixel(img, x[0], x[1],
70                          patchWidth, c), zip(compareX, compareY))) for c in locs])
71
72      return desc, locs
73
74
75  def detectCorners(img, sigma):
76      # fast method
77      result_img = skimage.feature.corner_fast(
78          img, n=PATCHWIDTH, threshold=sigma)
79      locs = skimage.feature.corner_peaks(result_img, min_distance=1)
80      return locs
```

loadVideo.py

```python
import numpy as np
import cv2


def loadVid(path):
    # Create a VideoCapture object and read from input file
    # If the input is the camera, pass 0 instead of the video file name
    cap = cv2.VideoCapture(path)

    # Check if camera opened successfully
    if (cap.isOpened() == False):
        print("Error opening video stream or file")

    i = 0
    # Read until video is completed
    frames = []
    while (cap.isOpened()):
        # Capture frame-by-frame
        i += 1
        ret, frame = cap.read()
        if ret == True:
            # Store the resulting frame
            frames.append(frame[np.newaxis, ...])
        else:
            break

    # When everything done, release the video capture object
    cap.release()

    frames = np.concatenate(frames, axis=0)
    frames = np.squeeze(frames)

    return frames
```

opts.py

```python
'''
Hyperparameters wrapped in argparse
This file contains most of tuanable parameters for this homework


You can change the values by changing their default fields or by command-line
arguments. For example, "python main.py --sigma 0.15 --ratio 0.7"
'''

import argparse


def get_opts():
    parser = argparse.ArgumentParser(description='16-720 HW2: Homography')

    # Feature detection (requires tuning)
    parser.add_argument('--sigma', type=float, default=0.15,
              help='threshold for corner detection using FAST feature detector')
    parser.add_argument('--ratio', type=float, default=0.7,
                                       help='ratio for BRIEF feature descriptor')

    # Ransac (requires tuning)
    parser.add_argument('--max_iters', type=int, default=500,
                                help='the number of iterations to run RANSAC for')
    parser.add_argument('--inlier_tol', type=float, default=2.0,
             help='the tolerance value for considering a point to be an inlier')

    # Additional options (add your own hyperparameters here)

    opts = parser.parse_args()

    return opts
```

panorama.py

```python
# Import necessary packages and functions
import sys
helper_function_path = '../python'
sys.path.insert(1, helper_function_path)

import numpy as np
import cv2

from ar_helper import matchPics
from planarH import computeH_ransac

from opts import get_opts
opts = get_opts()

# Write script for Q4.1x
def panorama(img1, img2, opts):
    # Match the left and right images
    matches, locs1, locs2 = matchPics(img1, img2, opts)

    # Get the coordinates of the matching pairs
    # Extract the matching points
    locs1 = locs1[matches[:, 0]]
    locs2 = locs2[matches[:, 1]]
    # Flip x and y coordinates
    locs1[:, [0, 1]] = locs1[:, [1, 0]]
    locs2[:, [0, 1]] = locs2[:, [1, 0]]

    # Compute Homography Matrix using RANSAC
    H2to1, inliers = computeH_ransac(locs1, locs2, opts)

    # Warp image2 to image1
    # Know the size of the input images
    h1, w1 = img1.shape[:2]
    h2, w2 = img2.shape[:2]
    # Compute the size of image2 after transfromation
    size_img2 = np.array([[0, 0], [w2, 0], [w2, h2], [0, h2]],
                                            dtype = np.float32).reshape(-1, 1, 2)
    trans_size_img2 = cv2.perspectiveTransform(size_img2, H2to1)
    # Set up the warping size (compute the boundary using trans_size_img2)
    warp_w = int(min(trans_size_img2[1][0][0], trans_size_img2[2][0][0]))
    warp_h = int(min(trans_size_img2[2][0][1], trans_size_img2[3][0][1]))
                  - int(max(trans_size_img2[0][0][1], trans_size_img2[1][0][1]))
    # Warp image2
    warp_img2 = cv2.warpPerspective(img2, H2to1, dsize = (warp_w, warp_h))
    # Check warp image2
    # cv2.imshow('warp_img2', warp_img2)
```

```
47      # cv2.waitKey(0)
48
49      # Composite image1 with warp image2
50      composite = warp_img2
51      composite[:warp_h, :w1, :] = img1[:warp_h, :, :]
52
53      return composite
54
55
56  # Main function
57  left = cv2.imread('left.jpg')
58  right = cv2.imread('right.jpg')
59  composite = panorama(left, right, opts)
60  cv2.imwrite('panorama.png', composite)
```