HOMEWORK 3: AUGMENTED REALITY WITH PLANAR HOMOGRAPHIES

16-720A Computer Vision (Spring 2023)
https://canvas.cmu.edu/courses/32966

OUT: March 3rd, 2023
DUE: March 27th, 2023 11:59 PM
Instructor: Deva Ramanan
TAs: Kangle Deng, Vidhi Jain, Xiaofeng Guo, Chung Hee Kim, Ingrid Navarro

Instructions/Hints

- Please refer to the course logistics page for information on the **Collaboration Policy** and **Late Submission Policy**.
- **Submitting your work:** There will be two submission slots for this homework on **Gradescope**: Written and Programming.
  - **Write-up.** For written problems such as short answer, multiple choice, derivations, proofs, or plots, we will be using the written submission slot. Please use this provided template. **We don't accept handwritten submissions.** Each answer should be completed in the boxes provided below the question. You are allowed to adjust the size of these boxes, but **make sure to link your answer to each question when submitting to Gradescope**. Otherwise, your submission will not be graded.
  - **Code.** You are also required to upload your code, which you wrote to solve this homework, to the Programming submission slot. Your code may be run by TAs so please make sure it is in a workable state. The assignment must be completed using Python 3.7 or newer. We recommend setting up a conda environment, but you are free to set up your environment however you like.
  - Regrade requests can be made after the homework grades are released, however this gives the TA the opportunity to regrade your entire paper, meaning if additional mistakes are found then points will be deducted.
- **Start early!** This homework is difficult and may take a long time to complete.
- **Verify your implementation as you proceed.** If you don't verify that your implementation is correct on toy examples, you will risk having a huge mess when you put everything together.
- **Q&A.** If you have any questions or need clarifications, please post in Slack or visit the TAs during office hours. Additionally, we provide a **FAQ** (Section 7) with questions from previous semesters and some **Helpful Concepts** (Section 8). Make sure you read it prior to starting your implementations.

Overview

In this assignment, you will be implementing an AR application step by step using planar homographies. Before we step into the implementation, we will walk you through the theory of planar homographies. In the programming section, you will first learn to find point correspondences between two images and use these to estimate the homography between them. Using this homography you will then warp images and finally implement your own AR applications.

1 Preliminaries

1.1 Planar Homographies as a Warp

Q1.1 (5 points):

Q1.1

Let $\mathbf{x_1} = \begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix}^T$, $\mathbf{x_2} = \begin{bmatrix} x_2 & y_2 & 1 \end{bmatrix}^T$ be the two camera views, and $\mathbf{x}_\pi$ lying in plane π that $\mathbf{x_p} = \begin{bmatrix} x_p & y_p & 1 \end{bmatrix}^T$
We know that: $\lambda_1 x_1 = \mathbf{H_1}\ \mathbf{x_p}$ and $\lambda_2 x_2 = \mathbf{H_2}\ \mathbf{x_p}$. Represent the $\mathbf{x}_p$ by $\mathbf{H}_1$ and $\mathbf{x}_1$: $\mathbf{H_1}^{-1}\lambda_1 x_1 = \mathbf{x_p}$.

Substitue $\mathbf{x}_p$, thus we have: $\lambda_2 x_2 = \mathbf{H_2}\ \mathbf{H_1}^{-1}\lambda_1 x_1$
$x_2 \equiv \mathbf{H}_2\mathbf{H}_1^{-1}\mathbf{x}_1$

To Prove that H exist, we are now proving $H_2\mathbf{H}_1^{-1}$ exist.

Since both H_1 and H_2 has an inverse matrix H_1^{-1} and H_2^{-1}, thus $H_2\ H_1^{-1}$ also exist.

Thus we have proven $\mathbf{H} = \mathbf{H}_2\ \mathbf{H}_1^{-1}$ exist

1.2 The Direct Linear Transform

Q1.2.1 (3 points): How many degrees of freedom does $\mathbf{h}$ have?

Q1.2.1

8. Since homography $\mathbf{H}$ is defined up to a scale factor, it has 8 DOF. Thus $\mathbf{h}$ also has 8 DOF.

Q1.2.2 (2 points): How many point pairs are required to solve $\mathbf{h}$?

Q1.2.2

Since there are 8 linearly independent equations are required, at least 4 point corresponding points' pairs between two images are required.

Q1.2.3 (5 points): Derive $\mathbf{A_i}$.

Q1.2.3

$\mathrm{x}_1 \equiv \mathbf{H}\mathrm{x}_2$

$$\mathrm{x}_1 = \begin{bmatrix} \mathbf{x}_1^i \\ \mathbf{y}_1^i \\ 1 \end{bmatrix} = \lambda \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix} \begin{bmatrix} \mathbf{x}_2^i \\ \mathbf{y}_2^i \\ 1 \end{bmatrix}$$

$Then we have:$

$\mathbf{x}_1^i = \lambda(\mathrm{h}_1\ \mathrm{x}_2^i + h_2\ \mathrm{y}_2^i + h_3)$ (1)

$\mathrm{y}_1^i = \lambda(\mathrm{h}_4\ \mathrm{x}_2^i + h_5\ \mathrm{y}_2^i + h_6)$ (2)

$1 = \lambda(\mathrm{h}_7\ \mathrm{x}_2^i + h_8\ \mathrm{y}_2^i + h_9)$ (3)

From (3), we can calculate $\lambda : \lambda = 1/(\mathrm{h}_7\ \mathbf{x}_2^i + h_8\ \mathbf{y}_2^i + h_9)$. Plug this equation into equation (1) and (2) and re-arrange the equation, we can get:

$h_7\ \mathrm{x}_1^i\ \mathrm{x}_2^i + h_8\ \mathrm{x}_1^i\ \mathrm{y}_2^i + h_9\ \mathrm{x}_1^i - h_1\ \mathrm{x}_2^i - h_2\ \mathrm{y}_2^i - h_3 = 0$

$h_7\ \mathrm{x}_2^i\ \mathrm{y}_1^i + h_8\ \mathrm{y}_1^i\ \mathrm{y}_2^i + h_9\ \mathrm{y}_1^i - h_4\ \mathrm{x}_2^i - h_5\ \mathrm{y}_2^i - h_6 = 0$

$\mathbf{A}_i\ \mathbf{h} = 0$, where $\mathbf{h} = \begin{bmatrix} \mathbf{h}_1 \\ \mathbf{h}_2 \\ \dots \\ \mathbf{h}_9 \end{bmatrix}$

Thus, $\mathbf{A}i = \begin{bmatrix} -\mathbf{x}_2^i & -\mathbf{y}_2^i & -1 & 0 & 0 & 0 & \mathbf{x}_1^i\ \mathrm{x}_2^i & \mathbf{y}_2^i\ \mathrm{x}_1^i & \mathbf{x}_1^i \\ 0 & 0 & 0 & -\mathbf{x}_2^i & -\mathbf{y}_2^i & -1 & \mathbf{x}_2^i\ \mathrm{y}_1^i & \mathbf{y}_2^i\ \mathrm{y}_1^i & \mathbf{y}_1^i \end{bmatrix}$

Q1.2.4 (5 points):

Q1.2.4

The trivial solution for **h** is 0-vector. But it is not useful for estimating **H** because it does not describe any meaningful transformation between two images.

A isn't full rank, becasue one of the rows of A can be removed without affecting the solutions. Since homography **H** is defined up to a scale factor, the size of A is 8x9 which means the number of row is less than the number of column.

Since $\mathbf{A}^T\ \mathbf{A}$ is not invertible, it has at least one zero eigenvalue, thus it will has at least one singular value.

1.3 Theory Questions

Q1.4.1 (5 points): Homography under rotation Note that $\mathbf{K_1}$ and $\mathbf{K_2}$ are the 3×3 intrinsic matrices of the two cameras and are different. $\mathbf{I}$ is 3×3 identity matrix, $\mathbf{0}$ is a 3×1 zero vector and $\mathbf{X}$ is a point in 3D space. $\mathbf{R}$ is the 3×3 rotation matrix of the camera.

Q1.4.1

$\mathbf{x_1} = \mathbf{K_1}[\mathbf{I}\ \ \mathbf{0}]\mathbf{X}$, and then we have $\mathbf{x_1} = \mathbf{K_1}\mathbf{x}'$, where $\mathbf{x}' = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$

$\mathbf{x}'$ can then be written as ${\mathbf{K_1}}^{-1}\mathrm{x}_1$

$\mathbf{x_2} = \mathbf{K_2}[\mathbf{R}\ \ \mathbf{0}]\mathbf{X}$, and then we have $\mathbf{x_2} = \mathbf{K_2}\mathbf{x}'$

Then we have $\mathbf{x_2} = \mathbf{K_2}\mathbf{R}{\mathbf{K_1}}^{-1}\mathbf{x_1}$

To prove H exist, we only needs to prove $\mathbf{K_2}\mathbf{R}{\mathbf{K_1}}^{-1}$ exists. Since ${\mathbf{K_1}}^{-1}$ has inverse, the multiplication also exists.

Thus we proved that there exists a homography $\mathbf{H}$ that satisfies $\mathbf{x_1} \equiv \mathbf{H}\mathbf{x_2}$, given two cameras separated by a pure rotation.

Q1.4.2 (5 points): Understanding homographies under rotation

Q1.4.2

Figure 1.1 shows the solution

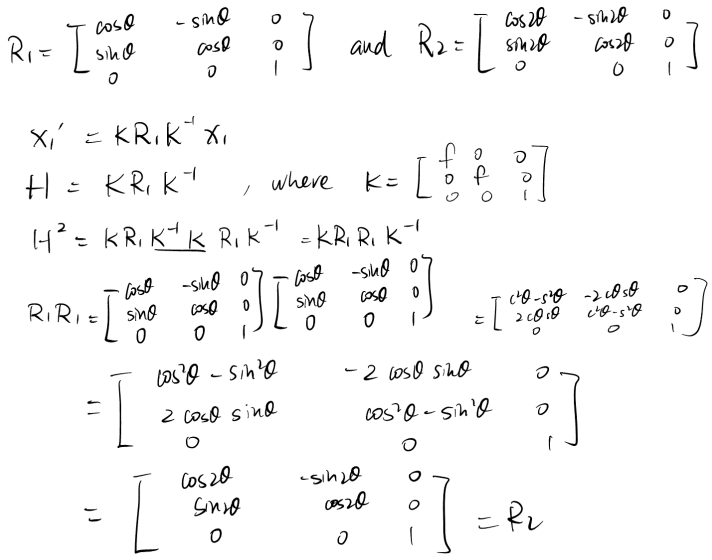


Figure 1.1: Solution for Q1.4.2.

Q1.4.3 (5 points): Limitations of the planar homography

Q1.4.3

If a scene contains objects are not coplanar, then it's not sufficient. For example it want to map image of cube from one to another viewpoint, we can only map a single plane of the cube.

Q1.4.4 (5 points): Behavior of lines under perspective projections

Q1.4.4

Figure 1.2 shows the solution

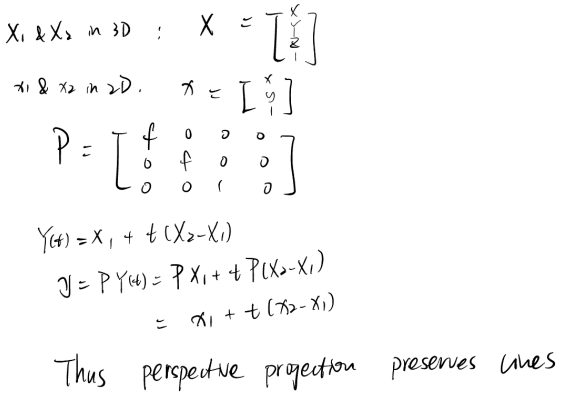


Figure 1.2: Solution for Q1.4.2.

2 Computing Planar Homographies

2.1 Feature Detection and Matching

Q2.1.1 (5 points): FAST Detector

Q2.1.1

The Harris corner detector is based on computing the eigenvalues of the local image structure matrix but the FAST detector is based on thresholding the intensity differences between pixels along concentric circles around a pixel.

In terms of computational performance, the FAST detector is generally faster than the Harris detector since the Harris detector calculates eigenvalues of local image structure which can be computationally expensive.

Q2.1.2 (5 points): BRIEF Descriptor

Q2.1.2

The BRIEF descriptor doesn't use convolutional filters to extract features from images like the filterbanks. Rather, it creates a random binary pattern for each feature point generated by comparing pixel intensities at a set of pairs of pixels around the feature point. The result of each comparison is either 1 or 0, and these values are concatenated to form a binary string that represents the descriptor.

It is possible to use a filter bank as a descriptor, but it may not be as efficient as the BRIEF descriptor.

Q2.1.3 (5 points):

Q2.1.3

Hamming distance is a measure of the difference between two binary strings of equal length. It is calculated by counting the number of positions in which the corresponding bits are different.

The Hamming distance is computationally more efficient than the Euclidean distance measure since it only requires counting the number of bit differences between two binary vectors, whereas the Euclidean distance measure requires computing the square root and sum of the squared differences between the corresponding vectors. Also since Euclidean distance computes the magnitude of the differences between two vectors, it can be affected by noise. However, the Hamming distance is more robust to noise since it only considers the number of bit differences but not the magnitude itself.

Q2.1.4 (10 points): Feature Matching

Q2.1.4 (a) result image

Figure 2.1 shows the matchi8ng result using the default parameters.

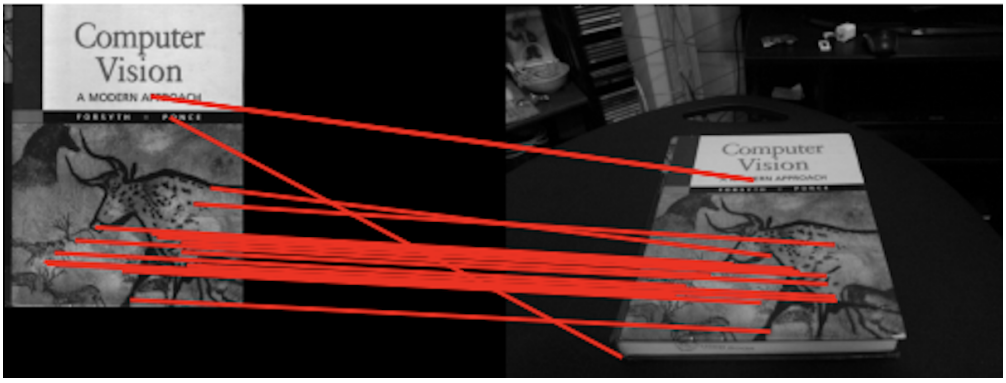


Figure 2.1: Matching result for Q2.1.4

Q2.1.4 (b) codes

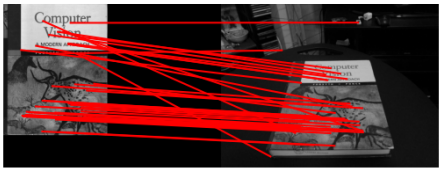
```python
import numpy as np
import cv2
import skimage.color
from helper import briefMatch
from helper import computeBrief
from helper import corner_detection

# 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
        gray_I1 = cv2.cvtColor(I1, cv2.COLOR_BGR2GRAY)
        gray_I2 = cv2.cvtColor(I2, cv2.COLOR_BGR2GRAY)

        # 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
```

Q2.1.5 (10 points):

Q2.1.5

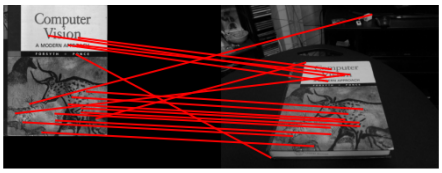
Figure 2.2 and 2.3 shows the ablation study by running with various sigma and ratio values. I first fix the sigma to be 0.15 and increase and decrease the ratio. From (a)(b), it can be seen that when I increase the ratio, the matched points number also increases obviously. This is because the ratio in the BRIEF descriptor is used to control the number of bits in the descriptor that generate binary sets that represent the feature. Increasing the ratio parameter leads to more sampling points being used, resulting in longer binary sets and thus more discriminative features.

As I fix the ratio and increase or decrease the sigma, the results are shown in (c)(d). The matched points numbers decrease as I increase the sigma. This happens because increasing the sigma in the FAST feature detector will increase the threshold for corner detection. It means that only the most prominent corners will be detected. Thus the result of increasing sigma causes a reduction in the number of matching points between two images, as there will be fewer keypoints to compare between the two images. However, the quality of the keypoints detected is better, as only the strongest corners will be considered.

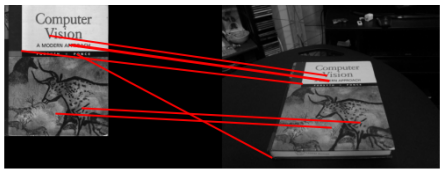


(a) sigma = 0.15, ratio = 0.75

(b) sigma = 0.15, ratio = 0.9



(c) sigma = 0.18, ratio = 0.7



(d) sigma = 0.22, ratio = 0.7

Figure 2.2: Solution for Q2.1.5

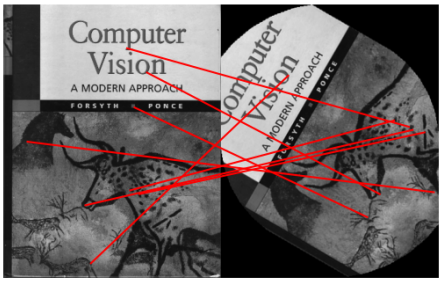
Q2.1.6 (10 points): BRIEF and Rotations

Q2.1.6 (a)

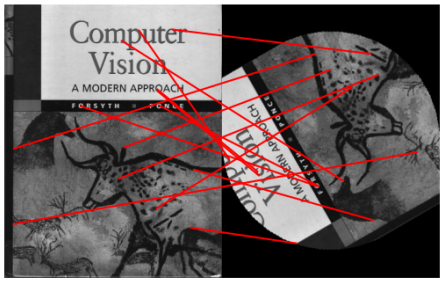
```python
import matplotlib
import numpy as np
import cv2
import scipy.ndimage
from matplotlib import pyplot as plt

from matchPics import matchPics
from opts import get_opts

# Q2.1.6
def rotTest(opts):
    # Read the image and convert to grayscale, if necessary
    img = cv2.imread('../data/cv_cover.jpg')
    # gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    hist = []
    rotate = []
    for i in range(36):

        # Rotate Image: rotate image itself in increments of 10 degrees
        if i == 0:
            rotate_img = img

        rotate_img = scipy.ndimage.rotate(rotate_img, 10, reshape =False)
        # Compute features, descriptors and Match features
        matches, locs1, locs2 = matchPics(img, rotate_img, opts)
        # Update histogram: stores a histogram of the count of matches for
    each orientation
        # print(len(matches))
        hist.append(len(matches))
        rotate.append(10*(i+1))
    # Display histogram
    plt.bar(rotate, hist, width=1.0, color='blue')
    #matplotlib.pyplot.hist(hist, bins=36)
    plt.xlabel("rotation")
    plt.ylabel("number of matches")
    plt.show()


if __name__ == "__main__":

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

Q2.1.6 (b)

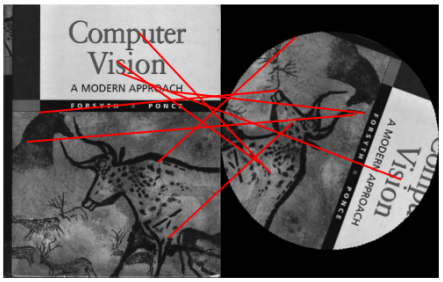
Figure 2.3 shows the matches from three different orientations and the histogram of the overall orientations from 0 to 360 degrees. From the histogram, we can see that the number of matches dropped drastically as the rotation angle increased from 0 to 10, and keeps very low until it returns to the original orientation. It works like this because BRIEF descriptors construct the descriptor by comparing random pixels' locations. Rotating an image would result in a change in the position of key points, leading to different descriptors for the same region in the image. Consequently, comparing the descriptors would involve analyzing pixel intensities at different positions.



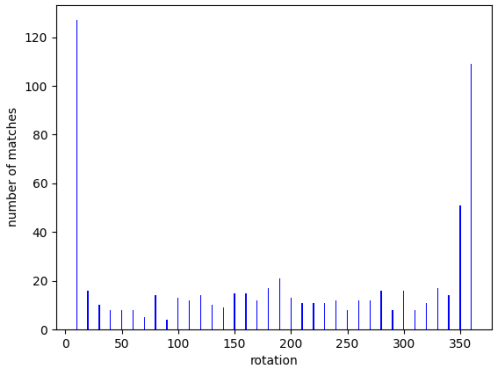
(a) orientation of 60 degrees



(b) orientation of 120 degrees



(c) orientation of 250 degrees



(d) histogram plots for all orientations

Figure 2.3: Solution for Q2.1.6

2.2 Homography Computation

Q2.2.1 (15 points): Computing the Homography

Q2.2.1

```
def computeH(x1, x2):
    # x1 and x2 are NX2 matrices containing the coordinates (x,y) of point
    pairs
    # Q2.2.1

    # Construct matrix A
    N = x1.shape[0]
    A = np.zeros((2*N, 9))
    i = 0
    n = 0
    while i < 2*N:
        x1i = x1[n][0]
        y1i = x1[n][1]
        x2i = x2[n][0]
        y2i = x2[n][1]

        A[i] = np.array([-x2i, -y2i, -1, 0, 0, 0, x2i*x1i, y2i*x1i, x1i])
        A[i+1] = np.array([0, 0, 0, -x2i, -y2i, -1, x2i*y1i, y2i*y1i, y1i])

        i += 2
        n += 1

    # Compute the homography between two sets of points
    U, Sig, Vh = np.linalg.svd(A)
    # get the index of the smallest eigenvalue
    h = Vh[-1, :]

    H2to1 = h.reshape((3,3))

    return H2to1
```

Q2.2.2 (10 points): Homography Normalization

Q2.2.2

```python
def computeH_norm(x1, x2):
    # Q2.2.2

    # Compute the centroid of the points
    # calculate the average of 4 coordinates
    x1x_avg = np.mean(x1[:, 0])
    x1y_avg = np.mean(x1[:, 1])
    x2x_avg = np.mean(x2[:, 0])
    x2y_avg = np.mean(x2[:, 1])

    # Shift the origin of the points to the centroid
    s1 = np.zeros((x1.shape[0]))
    s2 = np.zeros((x2.shape[0]))
    for i in range(x1.shape[0]):
        s1[i] = np.sqrt((x1[i, 0] - x1x_avg)**2 + (x1[i, 1] - x1y_avg)**2)
        s2[i] = np.sqrt((x2[i, 0] - x2x_avg) ** 2 + (x2[i, 1] - x2y_avg) ** 2)

    # Normalize the points so that the largest distance from the origin is
    equal to sqrt(2)
    scale1 = np.sqrt(2) / np.max(s1)
    scale2 = np.sqrt(2) / np.max(s2)

    # Similarity transform 1
    T1 = np.array([[scale1, 0, -scale1 * x1x_avg], [0, scale1, -scale1 *
    x1y_avg], [0, 0, 1]])
    x1_homo = np.hstack((x1, np.ones((x1.shape[0], 1))))
    x1_homo = T1 @ x1_homo.T
    x1_norm = x1_homo.T[:, 0:2]

    # Similarity transform 2
    T2 = np.array([[scale2, 0, -scale2 * x2x_avg], [0, scale2, -scale2 *
    x2y_avg], [0, 0, 1]])
    x2_homo = np.hstack((x2, np.ones((x2.shape[0], 1))))
    x2_homo = T2 @ x2_homo.T
    x2_norm = x2_homo.T[:, 0:2]

    # Compute homography
    H_norm = computeH(x1_norm, x2_norm)

    # Denormalization
    H2to1 = np.linalg.inv(T1) @ H_norm @ T2

    return H2to1
```

Q2.2.3 (25 points): Implement RANSAC

Q2.2.3

```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

    # locs1, locs2 are Nx2 matrices containing the matched points
    num_pts = locs1.shape[0] #2

    # Run RANSAC for max_iters iteration
    # bestH2to1 is homography H with most inliers found for during RANSAC
    bestH2to1 = None

    best_num_inliers = 0

    # inliers is a vector of length N with a 1 at those matches, 0 elsewhere
    inliers = np.zeros(num_pts)

    x1_homo = np.hstack((locs1, np.ones((locs1.shape[0], 1))))
    x2_homo = np.hstack((locs2, np.ones((locs2.shape[0], 1))))

    for i in range(max_iters):
        # choose 4 random points
        indices = np.random.choice(num_pts, 4, replace=False)
        x1_rand = locs1[indices]
        x2_rand = locs2[indices]
        # print("x1: ", x1_rand)
        # print("x2: ", x2_rand)

        # compute homography H using these points
        H2to1 = computeH_norm(x1_rand, x2_rand)

        # Apply H to all points in locs2
        proj_locs2 = H2to1 @ x2_homo.T
        proj_locs2 = proj_locs2 / proj_locs2[2, :]

        # compute distances between projected points and locs1
        distances = np.sqrt(np.sum((x1_homo.T - proj_locs2) ** 2, axis=0))

        # Count inliers
        num_inliers = np.sum(distances < inlier_tol)

        if num_inliers > best_num_inliers:
            bestH2to1 = H2to1
            best_num_inliers = num_inliers
            inliers = distances < inlier_tol

    return bestH2to1, inliers
```

Q2.2.4 (10 points): Automated Homography Estimation and Warping

Q2.2.4

```python
def warpImage(opts):
    # Read 3 images in:
    img1 = cv2.imread('../data/cv_cover.jpg')
    img2 = cv2.imread('../data/cv_desk.png')
    img3 = cv2.imread('../data/hp_cover.jpg')

    # compute H using cv_cover and cv_desk
    matches, locs1, locs2 = matchPics(img1, img2, opts)
    # the coordinate of locs returned from matchPics is (y,x)
    pair1 = locs1[matches[:, 0]]
    pair2 = locs2[matches[:, 1]]

    bestH2to1, inliers = computeH_ransac(pair1, pair2, opts)

    # Use computed H to warp hp_cover to cv_desk
    # h, w = img2.shape[:2]
    # hp_cover_warped = cv2.warpPerspective(img3, bestH2to1, (w, h))
    # cv2.imshow('Warped HP cover', hp_cover_warped)
    # cv2.waitKey(0)

    hp_resize = cv2.resize(img3, (img1.shape[1], img1.shape[0]))
    composite_img = compositeH(bestH2to1, hp_resize, img2)

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

Q2.2.4

```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.

    # Create mask of same size as template
    mask = np.ones(template.shape)

    # Warp mask by appropriate homography
    H2to1 = np.linalg.inv(H2to1)
    warp_mask = cv2.warpPerspective(cv2.transpose(mask), H2to1, (img.shape[0],
     img.shape[1]))
    warp_mask = cv2.transpose(warp_mask)

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

    # Use mask to combine the warped template and the image
    index = np.nonzero(warp_mask)
    img[index] = warp_template[index]

    # make the color back to rgb
    composite_img = cv2.cvtColor(img.astype('uint8'), cv2.COLOR_BGR2RGB)

    return composite_img
```

Figure 2.4 shows the result image.

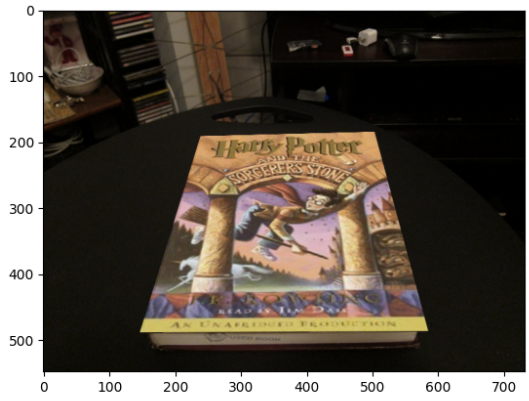


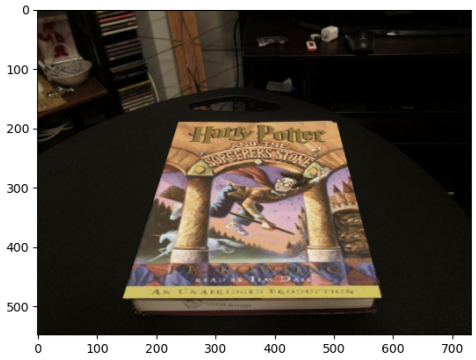
Figure 2.4: Composite image with desk images for Q2.2.4

Q2.2.5 (10 points): RANSAC Parameter Tuning

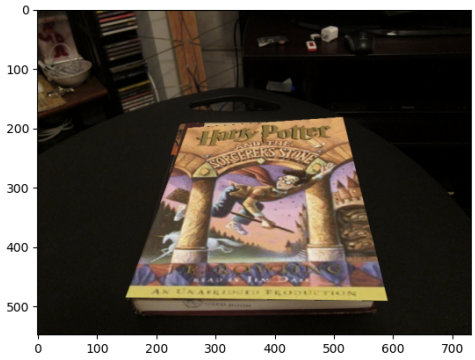
Q2.2.5

Figure 2.5 shows the ablation study by running with various max iters and inlier tol values. (a)(b) shows the result of increasing inlier tol with fixed max iters. Increasing the inlier tolerance in RANSAC will make the system consider more points as inliers which can lead to a more robust estimations. But increasing the tolerance too much will lead to more false positives in the final set of inliers which will cause the deform of the warp image as shown in (b).

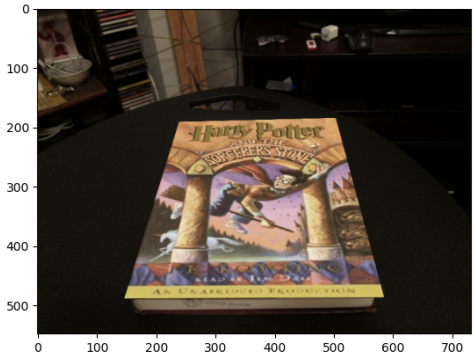
(c)(d) shows the result of increasing the max iters parameter with fixed inlier tolerance in RANSAC. The result difference is not so obvious with the iteration number increase. But logically increasing max iterations will lead the system to perform more iterations, which can lead to a better result.



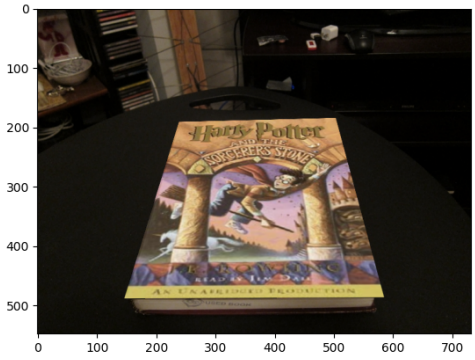
(a) max iters = 500, inlier tol = 5.0



(b) max iters = 500, inlier tol = 15.0



(c) max iters = 5000, inlier tol = 2.0



(d) max iters = 10000, inlier tol = 2.0

Figure 2.5: Solution for Q2.2.5

3 Creating your Augmented Reality application

3.1 Incorporating video (20 points)

Q3.1

```python
import numpy as np
import cv2

#Import necessary functions
from loadVid import loadVid
from helper import loadVid
from matplotlib import pyplot as plt
from opts import get_opts
from matchPics import matchPics
from planarH import computeH_ransac, compositeH

# Write script for Q3.1
def mymatchImg(opts, coverImg, book_frame, ar_frame):

    # matched the points between the cover of book image and the book image
    that is in video
    matches, locs1, locs2 = matchPics(coverImg, book_frame, opts)

    # the coordinate of locs returned from matchPics is (y,x)
    pair1 = locs1[matches[:, 0]]
    pair2 = locs2[matches[:, 1]]

    # compute homography using lImg and rImg
    bestH2to1, inliers = computeH_ransac(pair1, pair2, opts)

    # remove the top and bottom black space
    ar_frame = ar_frame[44:315, :, :]

    # crop the width of ar_frame
    croped_width = int(coverImg.shape[1] / coverImg.shape[0] * ar_frame.shape
    [0])
    disregard_width = int((ar_frame.shape[1] - croped_width)/2)
    ar_frame = ar_frame[:, 0+disregard_width:disregard_width+croped_width, :]

    # resize the ar_frame to be the same as cover
    ar_resize = cv2.resize(ar_frame, (coverImg.shape[1], coverImg.shape[0]))

    composite_img = compositeH(bestH2to1, ar_resize, book_frame)

    # plt.imshow(composite_img)
    # plt.show()

    return composite_img
```

Q3.1 Continued

```python
def ar(opts):
    ar_vid = loadVid('../data/ar_source.mov')
    book_vid = loadVid('../data/book.mov')
    coverImg = cv2.imread('../data/cv_cover.jpg')

    f_book, h_book, w_book, _ = book_vid.shape
    f_ar, _, _, _ = ar_vid.shape

    fps = 30
    fourcc = cv2.VideoWriter_fourcc(*'XVID')
    out = cv2.VideoWriter('../result/ar.avi', fourcc, fps, (w_book, h_book))

    for i in range(f_ar):
        if i <= 27 or i >= 157:
            print(i)
            b_frame = book_vid[i]
            ar_frame = ar_vid[i]
            composite_img = mymatchImg(opts, coverImg, b_frame, ar_frame)
            out.write(composite_img)

    cv2.destroyAllWindows()
    out.release()


if __name__ == "__main__":

    opts = get_opts()
    ar(opts)
```

The full ar.avi can be accessed through the link: https://drive.google.com/file/d/1ErW4OT35nn4O3zA5HX6UlnIhd7eeDeSX/view?usp=share_link

Figure 3.1: Ovelay at left (top). Overlay at center (middle). Overlay at right (bottom).

3.2 Make Your AR Real Time (Extra Credit - 15 points)

Q3.2

4 Create a Simple Panorama (10 points)

Q4

```python
import numpy as np
import cv2

# Import necessary functions
from matplotlib import pyplot as plt
from opts import get_opts
from matchPics import matchPics
from planarH import computeH_ransac, compositeH


# Q4
def panaromaImage(opts):
    lImg = cv2.imread('../data/left.jpg')
    rImg = cv2.imread('../data/right.jpg')
    # matched the points
    lh, lw,_ = lImg.shape
    rh, rw,_ = rImg.shape

    pana_resize = cv2.copyMakeBorder(rImg, 0, abs(rh-lh),
                                     int(max(lImg.shape[1], rImg.shape[1]) *
    1.1) - rw,
                                     0, cv2.BORDER_CONSTANT)
    matches, locs1, locs2 = matchPics(lImg, pana_resize, opts)
    # the coordinate of locs returned from matchPics is (y,x)
    pair1 = locs1[matches[:, 0]]
    pair2 = locs2[matches[:, 1]]

    # compute homography using lImg and rImg
    bestH2to1, inliers = computeH_ransac(pair1, pair2, opts)

    # Copy the left image onto the panorama image
    pImg = compositeH(bestH2to1, lImg, pana_resize)

    plt.imshow(pImg)
    plt.show()


if __name__ == "__main__":

    opts = get_opts()
    panaromaImage(opts)
```

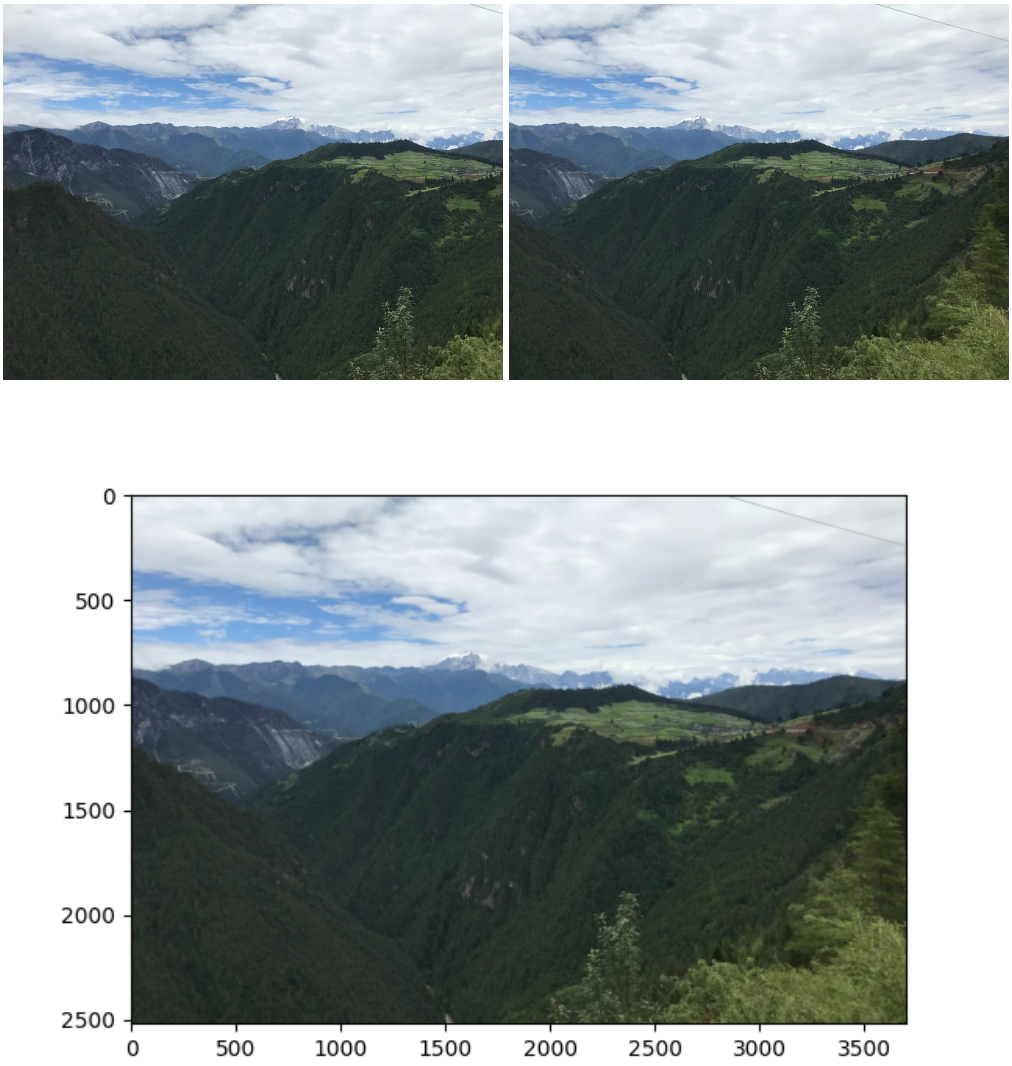


Figure 4.1: My left image (top left). My right image (top right). Panorama (bottom).

5 HW3 Distribution Checklist

After unpacking `hw3.zip`, you should have a folder `hw3` containing one folder for the data (`data`), one for your code (`code`) and one for extra credit questions (`ec`). In the `code` folder, where you will primarily work, you will find:

- `ar.py`: script for Section 3
- `briefRotTest.py`: script to test BRIEF with rotations
- `displayMatch.py`: script to display matched features
- `HarryPoterize.py`: script to automate homography and warp between source and target images
- `helper.py`: some helper functions
- `matchPics.py`: script to match features between pair of images
- `opts.py`: some command line arguments
- `planarH.py`: script to estimate planar homography
- `panaroma.py`: script you need to accomplish for Section 4

6 HW3 submission checklist

Submit your write-up and code to Gradescope.

- **Writeup.** The write-up should be a pdf file named <**AndrewId**>**_hw3.pdf**.
- **Code.** The code should be submitted as a zip file named <**AndrewId**>**.zip**. By extracting the zip file, it should have the following files in the structure defined below. (Note: Neglecting to follow the submission structure will incur a huge score penalty!)

  When you submit, remove the folder `data/` if applicable, as well as any large temporary files that we did not ask you to create.

  - <**andrew_id**>**/** # A directory inside .zip file
    - `code/`
      - <!– all of your .py files >
    - `ec/`
      - <!– all of your .py files >
    - `<andrew_id>_hw3.pdf` make sure you upload this pdf file to Gradescope. Please assign the locations of answers to each question on Gradescope.

7 FAQs

Credits: Cherie Ho

1. In `matchPics`, `locs` stands for pixel locations. `locs1` and `locs2` can have different sizes, since matches gives the mapping between the two for corresponding matches. We use `skimage.feature.match` descriptors (API) to calculate the correspondences.
2. Normalized homography - The function `computeH` norm should return the homography `H` between unnormalized coordinates and not the normalized homography `H` norm. As mentioned in the writeup, you can use the following steps as a reference:

$$\mathbf{H}_{\text{norm}} = \texttt{computeH}(\texttt{x1\_normalized}, \texttt{x2\_normalized})$$
$$\mathbf{H} = \mathbf{T}_1^{-1} @ \mathbf{H}_{\text{norm}} @ \mathbf{T}_2$$

3. The `locs` produced by matchPics are in the form of [`row`, `col`], which is (y,x) in coordinates. Therefore, you should first swap the columns returned by matchPics and then feed into Homography estimation.
4. Note that the third output `np.linalg.svd` is `vh` when computing homographies.
5. When debugging homographies, it is helpful to visualize the matches, and checking homographies with the same image. If there is not enough matches, try tuning the parameters.
6. If your images look blue-ish, your red and blue channels may be flipped. This is common when using cv2.imread. You can flip the last channel like so: hp_cover = hp_cover[:, :, [2, 1, 0]] or using another library (skimage.io.imread).
7. For Extra credit Q3.2, we'd like for you to speed up AR so that the processing is happening in real-time. This means we want each "for loop" you run with the ar.py to run in less than 1/30 seconds. You should not need to use multiprocessing. Take a look at your Q3.1 timings. Which step/steps are taking the most time? Can you replace these with faster functions? You are allowed to use functions from other libraries.
8. A common bug is reversing the direction of homography. Make sure it's what you expect!

8 Helpful Concepts

Credits: Jack Good

- **Projection vs. Homography:** A projection, usually written as $\mathbf{P}_{3\times4}$, maps homogeneous 3D world coordinates to the view of a camera in homogeneous 2D coordinates. A planar homography, usually written as $\mathbf{H}_{3\times3}$, under certain assumptions, maps the view of one camera in 2D homogeneous coordinates to the view of another camera in 2D homogeneous coordinates.
- **Deriving homography from projections:** When deriving a homography from projections given assumptions about the world points or the camera orientations, make sure to include the intrinsic matrices of the two cameras, $\mathbf{K}_1$ and $\mathbf{K}_2$, which are 3×3 and invertible, but generally cannot be assumed to be identity or diagonal. Note the rule for inverting matrix products: $(AB)^{-1} = B^{-1}A^{-1}$. When this rule is applied, even when both views are the same camera and $\mathbf{K} = \mathbf{K}_1 = \mathbf{K}_2$, $\mathbf{K}$ is still part of H and does not cancel out.
- **Conditions for planar homography to exist:** For a planar homography to exist between two views, we need either the points in the 3D world lie on a plane (as shown in 1.1 and applied in the Harry Potterize task), or there is pure rotation between the two views (as shown in 1.3 and applied in the Panorama task). We do not require both conditions to hold - only one or the other.

- **Definition of a line in 3D:** While we can define a line in 2D as the points (x, y) satisfying $ax + by + c = 0$, or equivalently in homogeneous coordinates, this does not generalize to 3D. More specifically, (x, y, z) such that $ax + by + cz + d = 0$ defines a plane in 3D. Moreover, while a line is uniquely identified by two or more collinear points, that does not define the line in its entirety. The simplest way to do so is to specify a line as all points $\mathbf{x} \in \mathbb{R}^3$ such that $\mathbf{x} = \mathbf{x}_1 + \lambda(\mathbf{x}_2 - \mathbf{x}_1)$, where $\mathbf{x}_1$ and $\mathbf{x}_2$ are two different points lying on the line, and $\lambda \in \mathbb{R}$. Several equivalent forms exist, and these definitions can be extended to homogeneous coordinates by appending a 1 value to each point.