Homework 2: Augmented Reality with Planar Homographies

Srividya Gandikota
sgandiko
sgandiko@andrew.cmu.edu

1 Homographies

1.1 Prove that there exists a homography H that satisfies equation 1 given two 3×4 camera projection matrices P1 and P2 corresponding to the two cameras and a plane Π. You do not need to produce an actual algebraic expression for H. All we are asking for is a proof of the existence of H.

$$x_1 \equiv H x_2 \tag{1}$$

For this scenario, we have three specific things to consider, P_1, P_2, and Π. Let us assume a point C exists with homogeneous coordinates $[x_i, y_i, z_i, 1]$. If we consider the image projection of point C on projection matrix P_1, we see that $x_1 = P_1 C$. The same can be said about the image projection on P_2 where $x_2 = P_2 C$. The equation below shows us that H exists in the form of a factor that maps points from the system P_2 to the system defined by P_1. This also shows us that Equation 1 is accurate to a scaling factor.

$$P_1^{-1} x_1 = C = P_2^{-1} x_2 \quad \rightarrow \quad x_1 = \frac{P_1}{P_2} x_2 \tag{2}$$

$$Let\ H = \frac{P_1}{P_2} \quad \rightarrow \quad x_1 = H x_2 \tag{3}$$

1.2 Let x1 be a set of points in an image and x2 be the set of corresponding points in an image taken by another camera. Suppose there exists a homography H such that:

$$x_1^i \equiv H x_2^i$$

where $x_1^i = [x_1^i y_1^i 1]^T$ are in homogeneous coordinates, $x_1^i \in x_1$ and H is a 3×3 matrix. For each point pair, this relation can be rewritten as

$$A_i h = 0$$

where h is a column vector reshaped from H, and A_i is a matrix with elements derived from the points x_1^i and x_2^i. This can help calculate H from the given point correspondences.

1. **How many degrees of freedom does h have?**
   - There are 8 degrees of freedom in h because even though there are 9 variables present, we need to pick h_{33} to avoid any zero division, which is the scaling factor.

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

2. **How many point pairs are required to solve h?**
   - We need a total of 4 point pairs, or 8 points to solve for h.
3. **Derive A_i.**
   - Given that the equation $x_1 = Hx_2$ holds, we can do the matrix algebra below.

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

   These can then be turned into three different equations $x_1 = h_{11}x_2 + h_{12}y_2 + h_{13}$, $x_2 = h_{21}x_2 + h_{22}y_2 + h_{23}$, and $x_3 = h_{31}x_2 + h_{32}y_2 + h_{33}$. As we know that A_ih $= 0$, we can solve for A_i and get the result below given that matrix h is the same as shown.

$$\text{For } h = \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix}, \text{ we see } A_i = \begin{bmatrix} -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_1x_2 & x_1y_2 & x_1 \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & y_1x_2 & y_1y_2 & y_1 \end{bmatrix}$$

4. **When solving Ah = 0, in essence you're trying to find the h that exists in the null space of A. What that means is that there would be some non-trivial solution for h such that that product Ah turns out to be 0. What will be a trivial solution for h? Is the matrix A full rank? Why/Why not? What impact will it have on the eigen values? What impact will it have on the eigen vectors?**

- The trivial solution for h would be the case were h is a size 9 x 1 matrix of $[000...0]^T$. Matrix A has to be one of full rank because it means that there cannot be any rows of A written as a linear combination of the other. This is requirement as the columns are linearly independent. The impact of this on the eigenvalues is that if we assume that A is not full rank, then the matrix is singular and we would have at least one eigenvalue of 0. Similarly, when A is not full rank, we can also see that there will be a few columns in A that are linearly dependent. This means that we will have multiple eigenvectors that may be linearly independent, but will all link to the 0 eigenvalue. In summary, if A is not full rank, then there can be eigenvalues of 0 and there will be non-trivial solutions in null space, which are not eigenvectors.

1.3 Prove that there exists a homography H that satisfies $x_1 = Hx_2$, given two cameras separated by a pure rotation. That is, for camera 1, $x_1 = K_1[I\ 0]$ X and for camera 2, $x_2 = K_2[R\ 0]$ X. Note that K1 and K2 are the 3 $\times$ 3 intrinsic matrices of the two cameras and are different. I is 3 $\times$ 3 identity matrix, 0 is a 3 $\times$ 1 zero vector and X is a point in 3D space. R is the 3 $\times$ 3 rotation matrix of the camera.

We can rewrite the two camera equations as such: $x_1 = K_1X$ and $x_2 = K_2RX$.

To solve for H, we can substitute the two equations and end up with: $K_1X = HK_2RX$.

If you filter out X, then $H = K_1K_2^{-1}R^{-1}$ which proves that there is a homography H that does satisfies $x_1 = Hx_2$.

1.4 Suppose that a camera is rotating about its center C, keeping the intrinsic parameters K constant. Let H be the homography that maps the view from one camera orientation to the view at a second orientation. Let θ be the angle of rotation between the two. Show that H^2 is the homography corresponding to a rotation of 2θ. Please limit your answer within a couple of lines. A lengthy proof indicates that you're doing something too complicated (or wrong).

We know that rotation matrix R corresponds to the rotation of angle θ for homography H. Given that K is constant, we can see that H^2 will correspond to R^2 similarly. Lets use the equations $H = K[R|0]K^{-1}$ and R' = R*R. The homography for this rotation can be seen below to correspond to 2θ.

$$\begin{aligned} H' &= K[R' \,|\, 0]K^{-1} \\ &= K[R \cdot R \,|\, 0]K^{-1} \\ &= K[R \,|\, 0][R \,|\, 0]K^{-1} \\ &= K[R \,|\, 0]K^{-1}K[R \,|\, 0]K^{-1} \\ &= HK[R \,|\, 0]K^{-1} \\ &= HH \\ &= H^2 \end{aligned}$$

1.5 Why is the planar homography not completely sufficient to map any arbitrary scene image to another viewpoint? State your answer concisely in one or two sentences.

Planar homography is not completely sufficient because it assumes that a surface is flat meaning it's incapable for situations with heavy depth variations. Because of this assumption, it can't handle perspective changes and makes it harder to map arbitrary scene images to different viewpoints.

1.6 We stated in class that perspective projection preserves lines (a line in 3D is projected to a line in 2D). Verify algebraically that this is the case, i.e., verify that the projection P in x = PX preserves lines.

To demonstrate perspective projection, let's use the perspective matrix P, which transforms a homogeneous coordinate $\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$ in world to a homogeneous coordinate $\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$ in image. The perspective matrix P with f as focal length can be expressed as:

$$P = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Now, consider a point $\mathbf{X}$ in 3D homogeneous coordinates, the perspective transformation $\mathbf{PX}$ can be calculated as:

$$\mathbf{PX} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} fX \\ fY \\ Z \end{bmatrix}$$

After the perspective projection, the 3D point $\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$ is transformed into $\begin{bmatrix} fX \\ fY \\ Z \end{bmatrix}$ in 3D homogeneous coordinates. Now, to obtain the non-homogeneous coordinates, we need to divide by the third coordinate Z:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \frac{1}{Z} \begin{bmatrix} fX \\ fY \\ Z \end{bmatrix} = \begin{bmatrix} \frac{fX}{Z} \\ \frac{fY}{Z} \\ 1 \end{bmatrix}$$

This shows us that that perspective projection preserves lines from 3D to 2D.

2 Computing Planar Homographies

2.1 Feature Detection and Matching

2.1.1 How is the FAST detector different from the Harris corner detector that you've seen in the lectures? Can you comment on its computational performance compared to the Harris corner detector?

The FAST detector is different than the Harris corner detector as it works by choosing a pixel instead of a moving window to compare 16 surrounding pixels. The pixel gets identified as a corner if it is a darker or lighter pixel than its surroundings by using a threshold. Because of these differences, the FAST detector has a low computation cost but it might be more likely to also falsely identify corners in noisy images. The Harris corner is more computationally expensive but it makes up for it with its accuracy and robustness to noise.

2.1.2 How is the BRIEF descriptor different from the filterbanks you've seen in the lectures? Could you use any one of those filter banks as a descriptor?

The BRIEF descriptor is a very fast method that gets an area around a point and uses a gaussian filter on that. From there, it then picks random N points pairs to match and get a descriptor. On the other hand, the filter banks we learned in class take much more time to find the same results. Theoretically, you would be able to modify some of the filter banks, like a Gabor filter, to work as a descriptor but you need to manipulate the results much further for that.

2.1.3 The BRIEF descriptor belongs to a category called binary descriptors. In such descriptors the image region corresponding to the detected feature point is represented as a binary string of 1s and 0s. A commonly used metric used for such descriptors is called the Hamming distance. Please search online to learn about Hamming distance and Nearest Neighbor, and describe how they can be used to match interest points with BRIEF descriptors. What benefits does the Hamming distance distance have over a more conventional Euclidean distance measure in our setting?

The Hamming distance is a measurement that finds the difference between two binary codes. In the case of nearest neighbors, given two images, it picks some points from the first image and places it into a set. Then it uses the ground truth to find the points for the second set. Once the descriptors are found, it goes through each point in the first set and uses the nearest neighbors algorithm to identify a match. In terms of benefits that the Hamming distance has over Euclidean distances, it is more easier for the CPU to calculate because all it does it apply an XOR for comparison.

2.1.4 Please implement a function in matchPics.py

The implementation of this function can be seen below in Figure 1. Along with that, when tested with a sigma value of 0.13 and a ratio of 0.7, the results are pretty well defined as seen in Figure 2.

```python
def matchPics(I1, I2, opts):
    #I1, I2 : 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
    #https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html
    I1 = cv2.cvtColor(I1, cv2.COLOR_BGR2GRAY)
    I2 = cv2.cvtColor(I2, cv2.COLOR_BGR2GRAY)

    #Detect Features in Both Images
    features_1 = corner_detection(I1, opts.sigma)
    features_2 = corner_detection(I2, opts.sigma)

    #Obtain descriptors for the computed feature locations
    descrip_1, locs1 = computeBrief(I1, features_1)
    descrip_2, locs2 = computeBrief(I2, features_2)

    #Match features using the descriptors
    matches = briefMatch(descrip_1, descrip_2, opts.ratio)

    return matches, locs1, locs2
```

Figure 1: matchPics Function Implementation

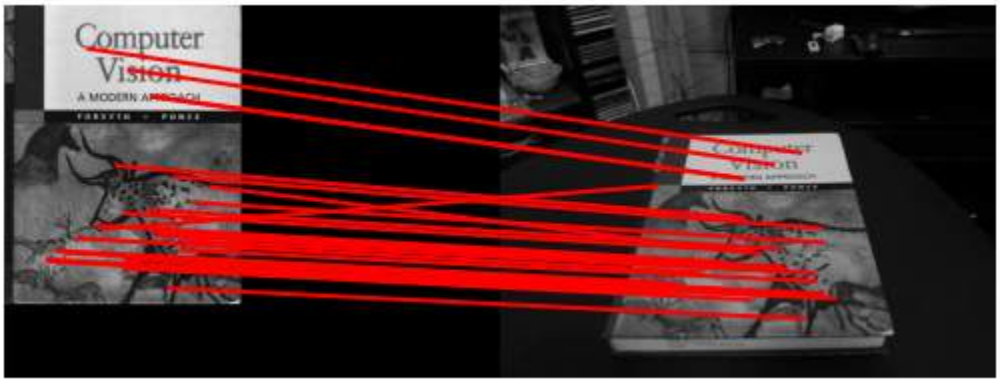


Figure 2: Results of Feature Matching

2.1.5 Conduct a small ablation study by running q2_1_4.py with various sigma and ratio values. Include the figures displaying the matched features with various parameters in your writeup, and explain the effect of these two parameters respectively.

The results of the study can be found in Figure 3 and 4 below. Based on the images, we can see that we can get more feature matches by lowering the value of sigma. This is because sigma is the corner detection threshold and we get more detected corners for a lower sigma, giving us more points for the descriptor. If we increase the ratio, we see that there are many more matched points because of there being more points for the descriptor.

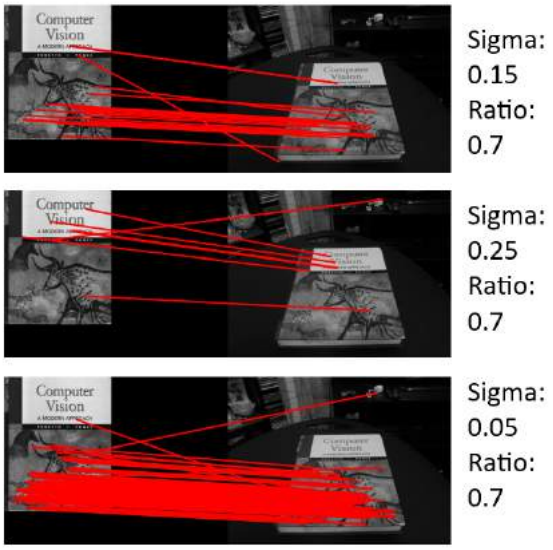


Figure 3: Results of Sigma Changes on Feature Matching

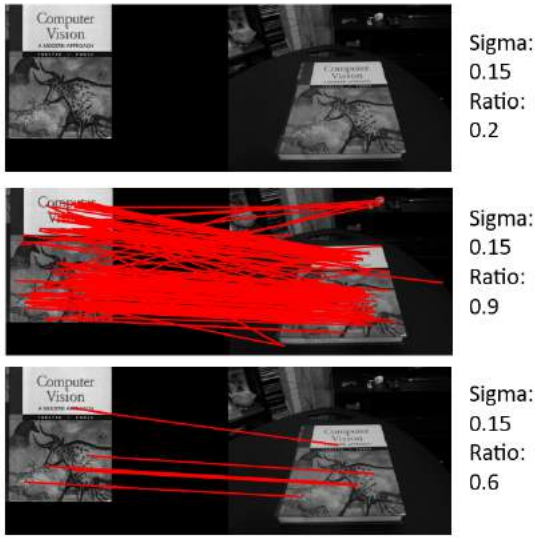


Figure 4: Results of Ratio Changes on Feature Matching

2.1.6 Visualize the histogram and the feature matching result at three different orientations and include them in your write-up. Explain why you think the BRIEF descriptor behaves this way.

The visualized histogram can be seen in Figure 5 below. We observe that the number of feature matches drops immediately as the angle rotation increases. This may be due to the BRIEF descriptors analyzing random pixels and making the bitstring descriptor. This means that every time we rotate the image, we will get a different descriptor for the specified patch and means that out comparisons will be wrong.

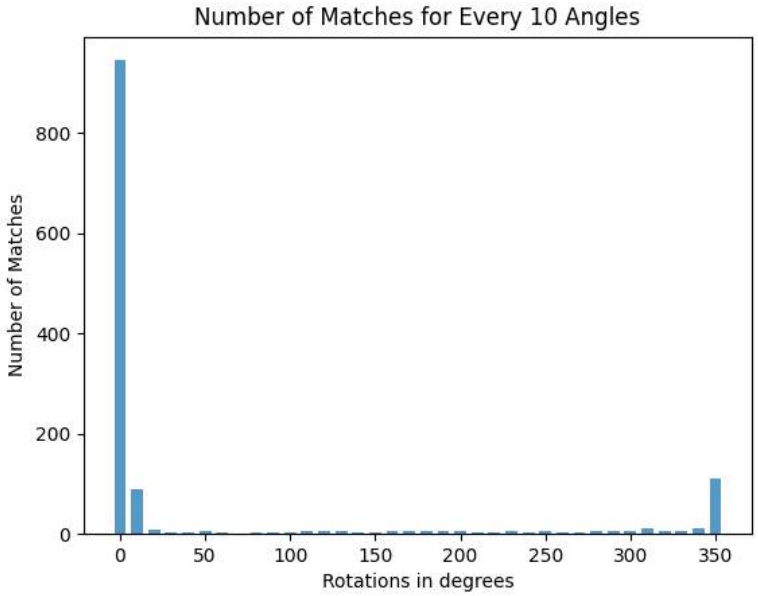


Figure 5: Histogram of Matches Per Angle Rotation

2.2 Homography Computation

2.2.1 Write a function computeH in planarH.py that estimates the planar homography from a set of matched point pairs.

The implementation of this function can be found in Figure 6 below.

```python
def computeH(x1, x2):
    #Q2.2.1
    #Compute the homography between two sets of points
    N = x1.shape[0]
    A = np.zeros((2 * N, 9))

    for i in range(N):
        A[2 * i, :] = [-x1[i, 0], -x1[i, 1], -1, 0, 0, 0, x1[i, 0] * x2[i, 0], x2[i, 0] * x1[i, 1], x2[i, 0]]
        A[2 * (i+1), :] = [0, 0, 0, -x1[i, 0], -x1[i, 1], -1, x2[i, 1] * x1[i, 0], x2[i, 1] * x1[i, 1], x2[i, 0]]

    # Solve for h (homography matrix in flattened form)
    _, _, V = np.linalg.svd(A)
    h = V[-1, :] / V[-1, -1]

    # Reshape h to get the homography matrix
    H2to1 = h.reshape((3, 3))
    return H2to1
```

Figure 6: computeH Function Implementation

2.2.2 Implement the function computeH_norm:

My implementation for the function compute_norm can be found below in Figure 7.

```python
def computeH_norm(x1, x2):
    #Q2.2.2
    #Compute the centroid of the points
    mean_x1 = np.mean(x1[:, 0])
    mean_y1 = np.mean(x1[:, 1])

    mean_x2 = np.mean(x2[:, 0])
    mean_y2 = np.mean(x2[:, 1])

    x1_centroid = [mean_x1, mean_y1]
    x2_centroid = [mean_x2, mean_y2]

    #Shift the origin of the points to the centroid
    x1_shifted = x1 - x1_centroid
    x2_shifted = x2 - x2_centroid

    #Normalize the points so that the largest distance from the origin is equal to sqrt(2)
    max_distance_x1 = np.max(np.sqrt((x1_shifted[:, 0] ** 2) + (x1_shifted[:, 1] ** 2)))
    max_distance_x2 = np.max(np.sqrt((x2_shifted[:, 0] ** 2) + (x2_shifted[:, 1] ** 2)))
    scale_x1 = np.sqrt(2) / max_distance_x1
    scale_x2 = np.sqrt(2) / max_distance_x2

    #Similarity transform 1
    scale_mat_x1 = np.eye(3) * scale_x1
    translation_mat_x1 = np.array([[1, 0, -mean_x1], [0, 1, -mean_y1], [0, 0, 1]])
    T1 = np.dot(scale_mat_x1, translation_mat_x1)

    # Similarity transform 2
    scale_mat_x2 = np.eye(3) * scale_x2
    translation_mat_x2 = np.array([[1, 0, -mean_x2], [0, 1, -mean_y2], [0, 0, 1]])
    T2 = np.dot(scale_mat_x2, translation_mat_x2)

    #Compute homography
    H_norm = computeH(x1_shifted, x2_shifted)

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

    return H2to1
```

Figure 7: compute_norm Function Implementation

2.2.3 Write a function: bestH2to1, inliers = computeH_ransac(locs1, locs2, opts) where locs1 and locs2 are N × 2 matrices containing the matched points.

My implementation for the function computeH_ransac can be found in Figure 8 below.

```
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_homogeneous = np.hstack((locs1[:, [1,0]], np.ones((locs1[:, [1,0]].shape[0], 1))))
    locs2_homogeneous = np.hstack((locs2[:, [1,0]], np.ones((locs2[:, [1,0]].shape[0], 1))))

    inlier_count = 0
    for i in range(opts.max_iters):
        #Get Random 4 Point Pairs
        np.random.seed(seed = None)
        random_points = np.random.choice(locs1[:, [1,0]].shape[0], 4, False)
        locs1_sample = locs1[:, [1,0]][random_points, :]
        locs2_sample = locs2[:, [1,0]] [random_points, :]

        #Compute Homography
        H = computeH_norm(locs1_sample, locs2_sample)

        #Transform with Homography
        locs2_trans = H @ locs2_homogeneous.T
        locs2_trans /= locs2_trans[2, :]
        locs2_trans = locs2_trans.T

        #Inliers Calculation
        error_calc = np.linalg.norm(locs1_homogeneous - locs2_trans, axis=1)
        inliers = np.sum(error_calc < opts.inlier_tol)

        #Select best fits
        if inlier_count <= inliers:
            inlier_count = inliers
            bestH2to1 = H

    return bestH2to1, inliers
```

Figure 8: computeH_ransac Function Implementation

2.2.4 Write a script HarryPotterize.py, the function compositeH and include results in writeup.

My implementation for the function compositeH can be found in Figure 9 below. Along with that, my implementation for the script HarryPotterize.py can be found in Figure 10 below with the results being saved into results by default. The results of the AR placement of Harry Potter cover can be seen in Figure 11 below.

```
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

    #Create Image Masks
    img_mask = cv2.warpPerspective(np.ones((template.shape)), np.linalg.inv(H2to1), (img.shape[1], img.shape[0]))

    #Warp Template
    warp_homography = cv2.warpPerspective(template, np.linalg.inv(H2to1), (img.shape[1], img.shape[0]))
    inverted_mask = (img_mask == 0).astype(int)

    #Apply Mask and Form Composite
    composite_img = inverted_mask * img + warp_homography
    print(composite_img.dtype)
    #Convert Color Scheme
    #Acomposite_img = composite_img.astype(np.float32)
    #composite_img = cv2.cvtColor(composite_img, cv2.COLOR_BGR2RGB)
    return composite_img
```

Figure 9: compositeH Function Implementation

```python
import numpy as np
import cv2
import skimage.io
import skimage.color
from opts import get_opts

#Import necessary functions
import matplotlib.pyplot as plt
from matchPics import matchPics
from planarH import computeH_ransac
from planarH import compositeH

#Write script for Q2.2.4
opts = get_opts()

#Reads Cover, Desk, and Harry Potter Cover
cv_desk = cv2.imread('../data/cv_desk.png')
cv_cover = cv2.imread('../data/cv_cover.jpg')
hp_cover = cv2.imread('../data/hp_cover.jpg')

#Compute Homography
matches, locs1, locs2 = matchPics(cv_desk, cv_cover, opts)
best_H2to1, inliers = computeH_ransac(locs2[matches[:,1]], locs1[matches[:,0]], opts)
hp_resize = cv2.resize(hp_cover,(cv_cover.shape[1],cv_cover.shape[0]))

composite_img = compositeH(best_H2to1, hp_resize, cv_desk)
cv2.imwrite('../results/Figure_10.jpg', composite_img)
```

Figure 10: HarryPotterize Script Implementation

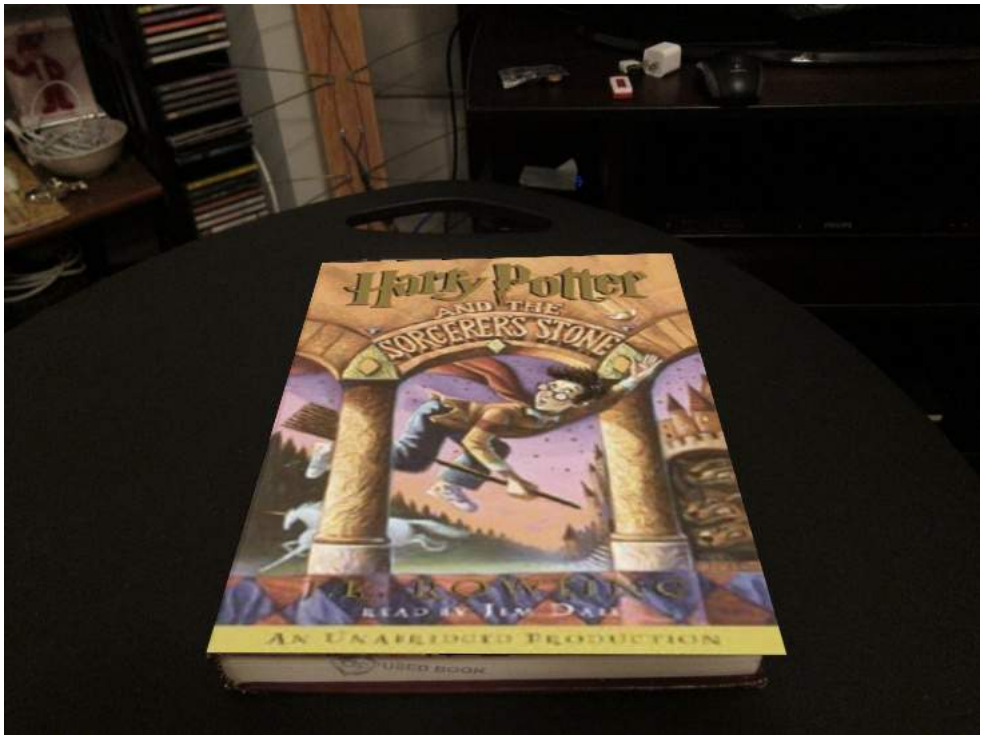


Figure 11: Results of CompositeImage

2.2.5 Conduct a small ablation study by running HarryPotterize.py with various max_iters and inlier_tol values. Include the result images in your writeup, and explain the effect of these two parameters respectively.

We can see in Figure 11 that if we increase the number of maximum iterations, we notice that the image gets matched much better to the cover. This may be due to the fact that by increasing the number of iterations, the number of inlier points that get matched also increase, leading to a much better matched image.

Figure 12: Results of study on Max_iters

We can see in Figure 12 that if we increase the inlier tolerances, we notice that the image gets matched much worse to the cover. This is because of the fact that by increasing the number of iterations, the inliers that can be matched gets decreased, leading to a much worse image fit.

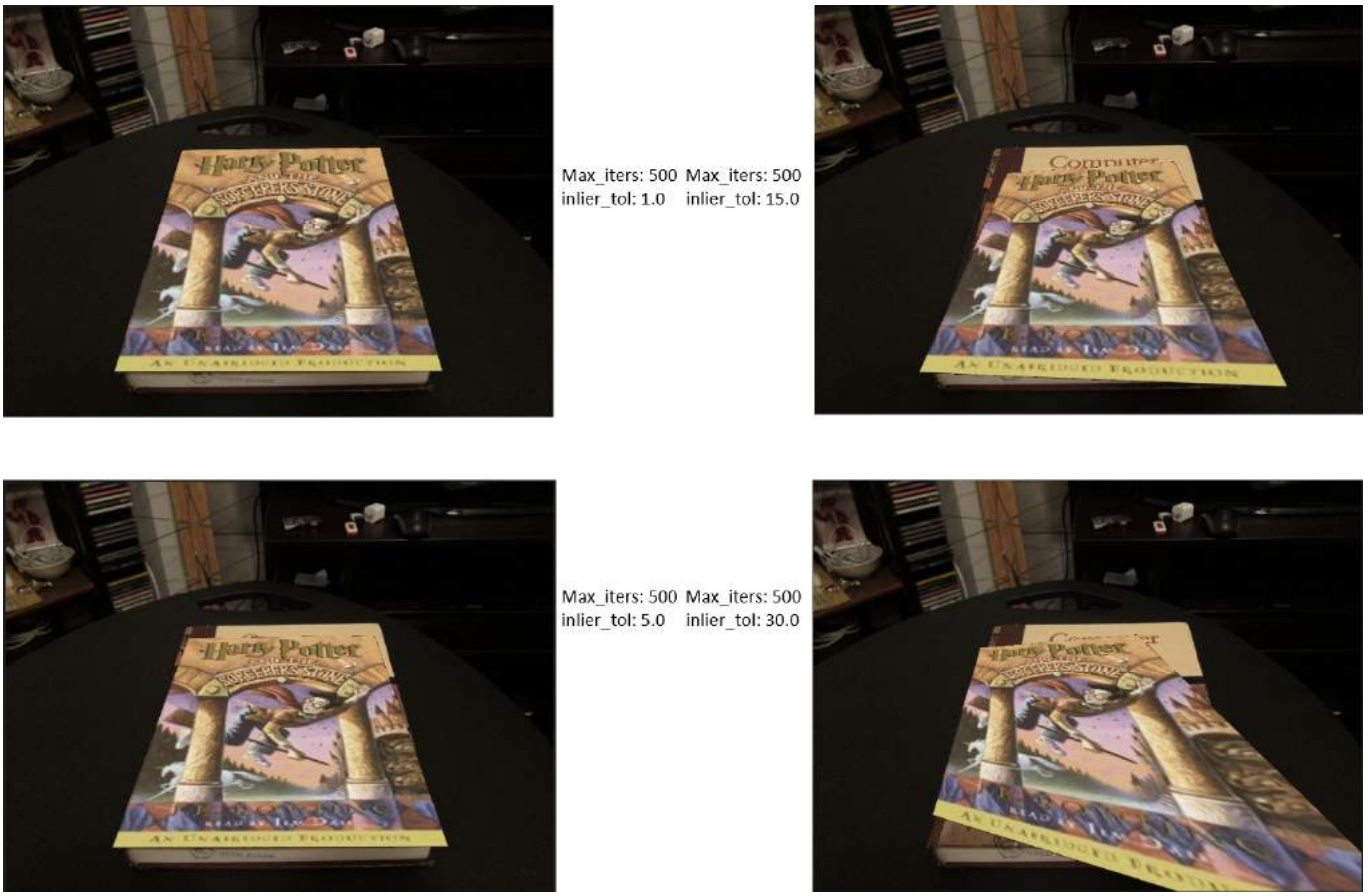


Figure 13: Results of study on inlier_tol

3 Creating Your Augmented Reality Application

3.1 Please write a script ar.py to implement this AR application and save your result video as ar.avi in the result/ directory. You may use the function loadVid() that we provide to load the videos. Your result should be similar to the LifePrint project. n your writeup, include three screenshots of your ar.avi at three distinct timestamps (e.g. when the overlay is near the center, left, and right of the video frame) in your write-up. See Figure 5 as an example of where the overlay is in the center of the video frame.

My implementation for the script ar.py can be found in Figure 14 below. Along with that, Figures 15, 16, and 17 show how well the overlays matched up at three distinct timesteps of the video, 5 seconds, 7 seconds, and 13 seconds.

```python
import numpy as np
import cv2
#Import necessary functions
import skimage.io
import skimage.color
from opts import get_opts
from matchPics import matchPics
from planarH import computeH_ransac
from planarH import compositeH
from loadVid import loadVid
from PIL import Image
#Write script for Q3.1
opts = get_opts()
#Load in video and image data
ar_vid = loadVid('../data/ar_source.mov')
cv_cover = cv2.imread('../data/cv_cover.jpg')
book_vid = loadVid('../data/book.mov')

#Set up feature matching
locs1_arr=[]
locs2_arr=[]
matches_arr=[]
bestH2to1_arr=[]
composite_list=[]

#Create video padding
frame_difference = book_vid.shape[0] - ar_vid.shape[0]
frames_to_pad = []
for i in range(frame_difference):
    frames_to_pad.append(ar_vid[i, :, :, :])
video_pad = np.stack(frames_to_pad, axis=0)
ar_vid = np.concatenate((ar_vid, video_pad), axis=0)

#Start feature mapping and homography
for i in range(book_vid.shape[0]):
    matches, locs1, locs2 = matchPics(book_vid[i,:,:,:], cv_cover, opts)
    locs1[:, [0,1]] = locs1[:, [1,0]]
    locs2[:, [0,1]] = locs2[:, [1,0]]
    bestH2to1, inliers = computeH_ransac(locs1[matches[:,0]], locs2[matches[:,1]], opts)
    dimension =(cv_cover.shape[1], cv_cover.shape[0])
    aspect_ratio = cv_cover.shape[1] / cv_cover.shape[0]
    #Create video cover
    new_cover = ar_vid[i,:,:,:]
    #Remove black bars
    new_cover = new_cover[44:-44,:]
    height, weight, channels = new_cover.shape
    width_ar = height * cv_cover.shape[1] / cv_cover.shape[0]
    new_cover = new_cover[:, int((weight/2) - width_ar/2):int((weight/2) + width_ar/2)]
    new_cover = cv2.resize(new_cover, dimension)

    # Combine augment into frame
    composite_img = compositeH(bestH2to1, new_cover, book_vid[i,:,:,:])
    composite_list.append(composite_img)
    print(str(i) + " out of " + str(book_vid.shape[0]))

#Generate video
#https://www.geeksforgeeks.org/saving-a-video-using-opencv/#
fps = 25
output_dir = '../results/ar.avi'
fourcc = cv2.VideoWriter_fourcc(*"MJPG")
video = None
size = composite_list[0].shape[1], composite_list[0].shape[0]

for img in composite_list:
    if video is None:
        video = cv2.VideoWriter(output_dir, fourcc, float(fps), size, True)
    video.write(np.uint8(img))

video.release()
```

Figure 14: Implementation of ar.py

I didn't implement multiprocessing for this section because I was having difficulties so this meant that the runtime for this was almost 2-3 hours. However, even though it took such a long time, we can see that the results below show us that the tracking was accurate and able to follow across all of the timesteps.

Figure 15: Results of AR at 5 seconds (Left)

Figure 16: Results of AR at 7 seconds (Center)

Figure 17: Results of AR at 13 seconds (Right)

4 Extra Credit

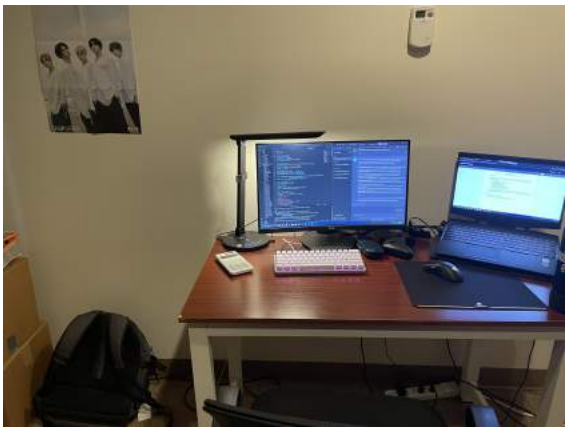
4.1 Take two pictures with your own camera, separated by a pure rotation as best as possible, and then construct a panorama with panorama.py. Be sure that objects in the images are far enough away so that there are no parallax effects. You can use python module cpselect to select matching points on each image or some automatic method. Submit the original images, the final panorama, and the script panorama.py that loads the images and assembles a panorama. We have provided two images for you to get started (data/pano left.jpg and data/pano right.jpg). Please use your own images when submitting this project. In your submission, include your original images and the panorama result image in your write-up. See Figure 7-9 below for example.

My implementation for the panorama.py script can be found in Figure 18 below. One key thing to note for this function is I used a reference on how to go about stitching the images together. The code itself was written by me but I used some suggestions on some approaches to make sure the images match over each other. (https://kushalvyas.github.io/stitching.html)

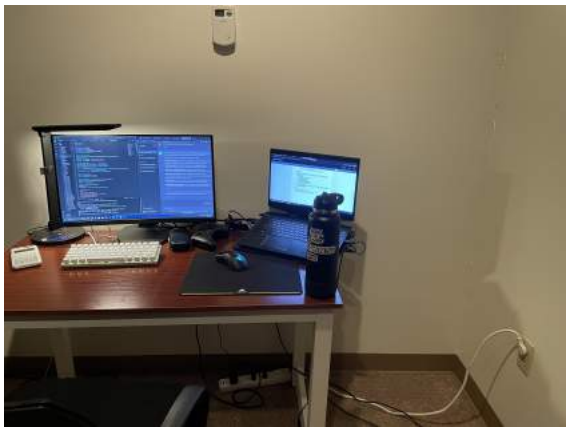
```python
import numpy as np
import cv2

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

# Write script for Q4.2x
#https://kushalvyas.github.io/stitching.html
#Get Images to Pano
img_left = cv2.imread('../data/online_left.jpg')
img_right = cv2.imread('../data/online_right.jpg')

#Get Image Dimensions
img_left_height, img_left_weight, _ = img_left.shape
img_right_height, img_right_weight, _ = img_right.shape

# Calculate the width for image adjustment
max_weight = max(img_left_weight, img_right_weight)
adjusted_width = round(max_weight * 1.2)

# Create a padded version of img_right to match img_left dimensions
padded_img_right = cv2.copyMakeBorder(
    img_right,
    top=0,
    bottom=img_right_height - img_left_height,
    left=adjusted_width - img_right_weight,
    right=0,
    borderType=cv2.BORDER_CONSTANT,
    value=0
)

# Match features between img_left and padded_img_right
matches, locs1, locs2 = matchPics(img_left, padded_img_right, get_opts())

# Display the matched features
plotMatches(img_left, padded_img_right, matches, locs1, locs2)


#Find the Matched Points
locs1 = locs1[matches[:, 0], 0:2]
locs2 = locs2[matches[:, 1], 0:2]

#Compute Homography
bestH2to1, inliers = computeH_ransac(locs1, locs2, get_opts())
img_pano = compositeH(bestH2to1, img_left, padded_img_right)

#Max and Generate Panoramic
pano_im = np.maximum(padded_img_right, img_pano)
print('Done')
cv2.imwrite('../results/online_pano.png', img_pano)
```

Figure 18: Implementation of panorama.py

I ran a couple of tests after writing the function to test the image stitching properties. My first set of images included photos I took of my table. We can easily see that inf Figure 21, the panorama did try to form but there is some parallax effects visible because it wasn't taken far away from each other.



(a) Left Image of Table



(b) Right Image of Table

Figure 20: Implementation of panorama.py on My Images

Since I did not have any photos from my phone that were from a far distances, I took some from an online source to try to see if the panoramic disfiguration came from the parallax effect or if the script was incorrect. Figure 22 looks like an almost perfect panoramic so we can see here that the script works for photos with distinct features and taken at a much further range.

(a) Left Image of Mountain

(b) Right Image of Mountain

Figure 22: Implementation of panorama.py on Online Images