1.1 Planar Homographies as a Warp

For a planar homography to exist between two views, we either need the points in the 3D world to lie on a plane or for there to be pure rotation between the views. In this case, the points in the 3D world all do lie on a plane, so we know that a planar homography exists. The proof for this theory is shown below.

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \overset{\substack{\text{Camera intrinsics}\\ (k) \\ \uparrow}}{\begin{bmatrix} S_x & S_\theta & 0_x \\ 0 & S_y & O_y \\ 0 & 0 & 1 \end{bmatrix}} \overset{\substack{\text{camera extrinsics}\\ \text{(rotation \& translation)} \\ \uparrow}}{\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix}} \overset{\substack{\text{3D point}\\ \uparrow}}{\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}}$$

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Plug in $z = 0$

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{14} \\ m_{21} & m_{22} & m_{24} \\ m_{31} & m_{32} & m_{34} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2 camera views:

$$\begin{aligned} \lambda_1 \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} &= H_1 \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \\ \lambda_2 \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} &= H_2 \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \end{aligned} \quad \rightarrow \lambda \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = H_2 H_1^{-1} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

1.2 The Direct Linear Transform

1. 8 degrees of freedom
2. 4 matching points
3. Derivation of Ai

relating source image to destination image w / homography:

$$\begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix} \equiv \begin{bmatrix} \tilde{x}_d \\ \tilde{y}_d \\ \tilde{z}_d \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix}$$

for a given pair i of corresponding points:

$$x_d = \frac{\tilde{x}_d}{\tilde{z}_d} = \frac{h_{11}x_s + h_{12}y_s + h_{13}}{h_{31}x_s + h_{32}y_s + h_{33}}$$
$$y_d = \frac{\tilde{y}_d}{\tilde{z}_d} = \frac{h_{21}x_s + h_{22}y_s + h_{23}}{h_{31}x_s + h_{32}y_s + h_{33}}$$

where x_s, y_s, x_d, y_d w.r.t. i

rearrange terms:

$$x_0 \left(h_{31}x_s + h_{32}y_s + h_{33}\right) = h_{11}x_s + h_{12}y_s + h_{13}$$
$$y_d \left(h_{31}x_s + h_{32}y_s + h_{33}\right) = h_{21}x_s + h_{22}y_s + h_{23}$$

write as linear equation:

$$\underset{\underset{\text{Ai}}{\downarrow}}{\begin{bmatrix} x_s & y_s & 1 & 0 & 0 & 0 & -x_d x_s & -x_d y_s & -x_d \\ 0 & 0 & 0 & x_s & y_s & 1 & -y_d x_s & -y_d y_s & -y_d \end{bmatrix}} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

4. The trivial solution of Ah=0 is when h=0.
   The matrix A is not full rank. We know this because for the matrix to be full rank it would need to be a square matrix, and A is not a square matrix.
   Because the matrix is not full rank, the singular values (eigenvalues of $A^T A$) will have at least one zero.

1.4 Theory Questions

1.4.1 Homography under rotation

If two camera images are separated by pure rotation, then there will be a homography that satisfies $x_1=Hx_2$. The relationship between the two camera views is shown below, which can be used to prove that the homography exists.

Relation between 3D camera coordinates:

$$\begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = R \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \rightarrow$$

Points related by rotation matrix R (3x3)

3D → 2D:

$$\lambda \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} f_2 & 0 & 0 \\ 0 & f_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix}$$

Combine:

$$\lambda \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = K_2 R K_1^{-1} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

After combining the equations, we can see that a homography exists when there is pure rotation between camera views.

1.4.2 Understanding homographies under rotation

If a camera is rotating about its center C, then H^2 is the homography corresponding to a rotation of 2θ (if θ is the angle of rotation between the two camera orientations). This statement can be seen from getting a homography H between two camera views rotated by theta (x1 to x2), then getting another homography H for another rotation of theta (x2 to x3). Using the transitive property, we can see that the homography ends up being H^2 for a rotation of 2θ which is shown algebraically below.

$$x_1 \to x_2 :$$

$$H_{12} = K_1 R K_2^{-1}$$

$$x_2 = H_{12} x_1$$

$$x_2 \to x_3 :$$

$$H_{23} = K_2 R K_3^{-1}$$

$$x_3 = H_{23} x_2$$

$$x_3 = H_{23} \left(H_{12} x_1 \right)$$

$$x_3 = H_{23} H_{12} x_1$$

$$H_{23} = H_{12} = H \to \text{ same rotations}$$

$$x_3 = H^2 x_1$$

1.4.3 Limitations of the Planar Homography

For a planar homography to exist between two views, there either needs to be pure rotation between the views or the points in the 3D world must lie on a plane. At least one of these two conditions needs to be true for a planar homography to exist. Given two arbitrary images, at least one of the above conditions may not always hold true. In this case, a planar homography may not exist. Therefore, a planar homography alone may not be sufficient to map any arbitrary scene image to another viewpoint.

1.4.4 Behavior of lines under perspective projections

Equation of a line in 3D:

$$x = m_x z + b_x \quad (1)$$
$$y = m_y z + b_y \quad (2)$$

Assuming the center of projection is at the origin, project onto the plane $z = f$, then point (x_0, y_0, z_0) is:

$$x_i = \frac{f x_0}{z_0}$$
$$y_i = \frac{f y_0}{z_0}$$

combine equations (1) and (2):

$$\frac{x_i z_0}{f} - m_x z_0 = b_x \quad (3)$$
$$\frac{y_i z_0}{f} - m_y z_0 = b_y \quad (4)$$

Divide equation 3 by equation 4 :

$$\frac{x_i/f - m_x}{y_i/f - m_y} = \frac{b_x}{b_y}$$
$$x_i = \frac{b_x}{b_y} y_i + f\left(m_x - \frac{b_x m_y}{b_y}\right) \rightarrow x_i = m y_i + b$$

Therefore, perspective projection preserves lines. A line in 3D is projected to a line in 2D.

2.1 Feature Detection and Matching

2.1.1 FAST Detector

FAST is a corner detection method which is computationally more efficient than the Harris corner detection method. FAST is suitable for real-time video processing applications because of its high-speed performance. The FAST algorithm works by selecting a pixel p with intensity Ip in the image which is to be identified as an interest point or not. A threshold value t is also set. A circle of 16 pixels around pixel p is considered. If n contiguous pixels out of the circle of 16 pixels are brighter than Ip+t or darker than Ip-t, then pixel p is a corner. To make the algorithm fast, the pixels 1, 5, 9, and 13 in the circle of 16 are first compared to Ip. At least three of these four pixels should satisfy the threshold criteria for an interest point to exist. If three pixels do not, then that pixel p is not an interest point. If three of the four pixels do match the criteria, then all 16 pixels in the circle are checked. If n pixels then meet the threshold criteria, p is an interest point. A machine learning approach, such as decision tree, can also be applied to improve computation time. There are a few limitations to the algorithm. If n is less than 12, the algorithm does not work well because too many points are likely to be detection. Detecting multiple interest points in adjacent locations is also an issue with this algorithm, so non-maximal suppression is used.

As we learned in class, Harris corner detection works by considering a small window around each pixel in an image and shifting it to measure changes around that pixel value and determine how unique the pixel location is. Looking at these two methods, it’s clear that FAST is much more computationally efficient than Harris corner detection. For every pixel, FAST starts by only looking at 4 other pixel values nearby. However, Harris corner detection starts by considering a patch around every pixel value, shifting the patch small amounts, taking the sum of squared difference of the pixel values before and after the shift, and applying a Taylor expansion. Naturally, the FAST method will have better computational performance.

2.1.2 BRIEF Descriptor

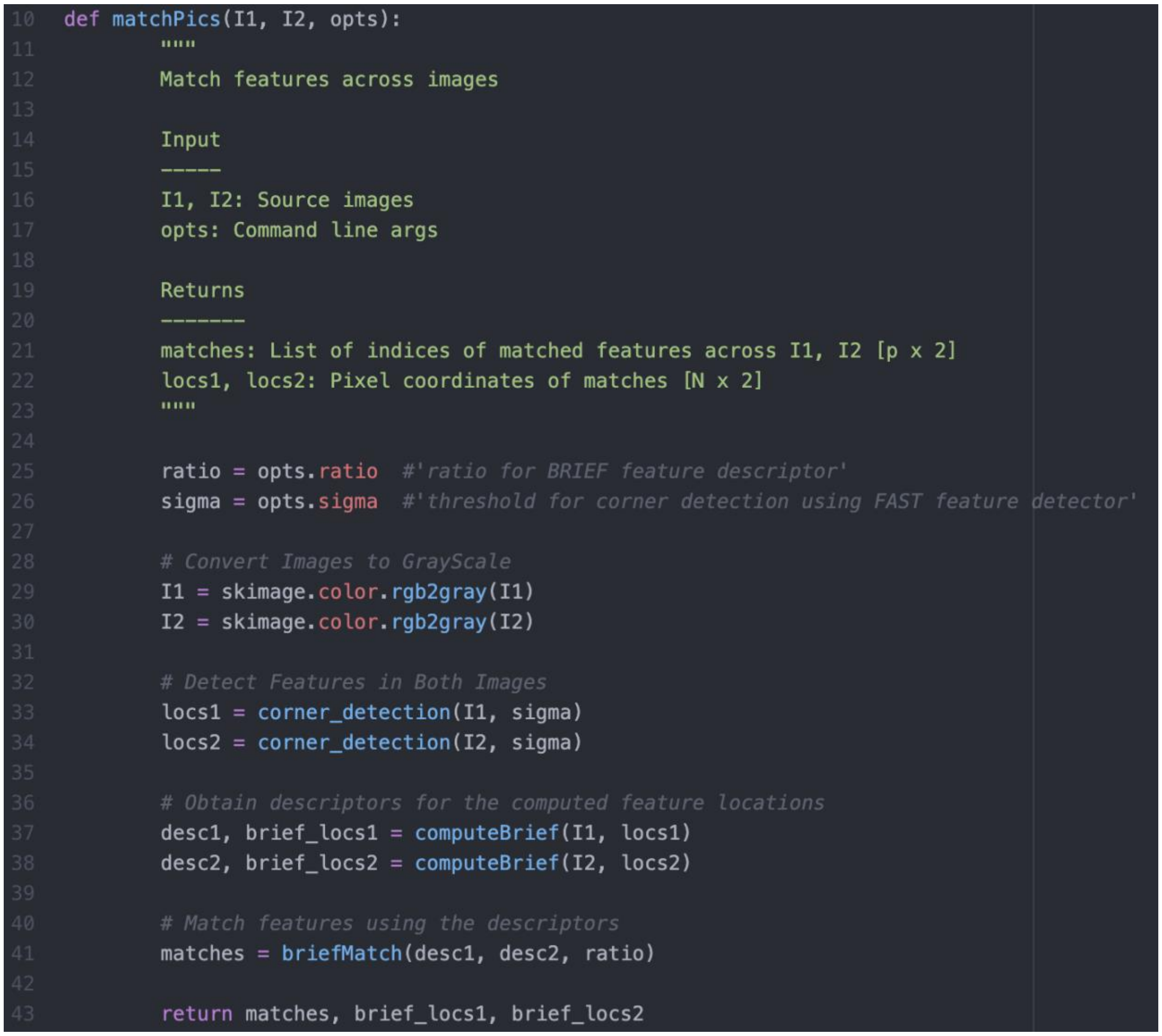
BRIEF uses binary strings as an efficient feature point descriptor. BRIEF is both fast to build and fast to match. It outperforms other descriptors we've talked about in lecture, such as SIFT, in terms of speed and recognition rate. BRIEF works by taking a patch around an interest point and first smoothing the patch because it's very noise-sensitive. After smoothing, a binary feature vector is created by comparing the pixel intensity of the interest point to the pixel intensity of other selected point pairs. These pairs are generated through certain geometric sampling approaches. 1s and 0s are saved into the binary vector through comparing the pixel intensity of the interest point with the other selected pair points.

BRIEF is different from SIFT or other descriptors we discussed in class because it uses binary strings as the descriptor of the point. The filter banks we saw in class can also be used as descriptors. However, they would be much more computationally intensive than using BRIEF.

2.1.3 Matching Methods

Hamming distance works by checking the number of positions at which corresponding symbols are different. This method works well for strings because we would just measure the minimum number of substitutions or errors needed to change one string into another. Because BRIEF uses binary strings as a feature point descriptor, it makes more sense to use Hamming Distance in this case as a matching metric rather than the Euclidean Distance because we are comparing binary strings. We can simply check how many elements in the string would need to be changed to determine distance.

2.1.4 Feature Matching

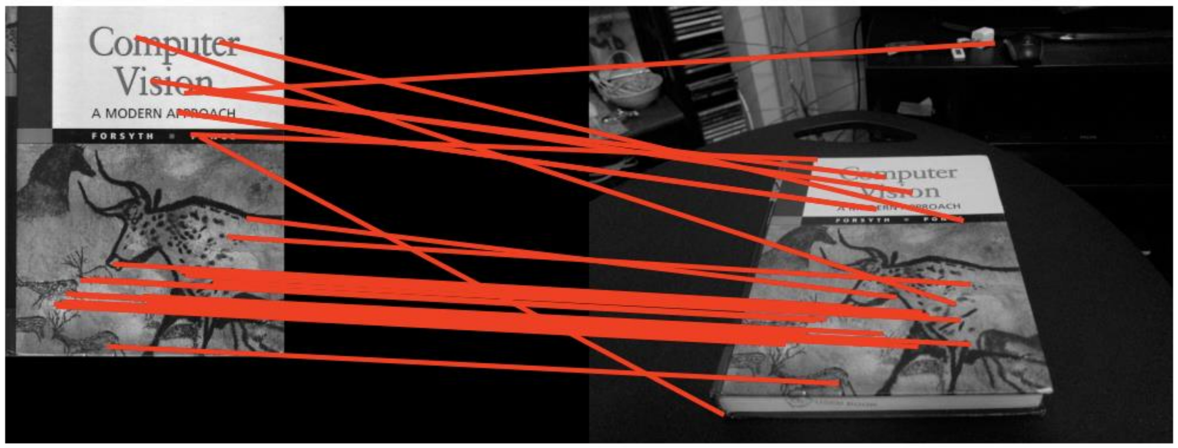


```python
def matchPics(I1, I2, opts):
        """
        Match features across images

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

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

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

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

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

        # Obtain descriptors for the computed feature locations
        desc1, brief_locs1 = computeBrief(I1, locs1)
        desc2, brief_locs2 = computeBrief(I2, locs2)

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

        return matches, brief_locs1, brief_locs2
```

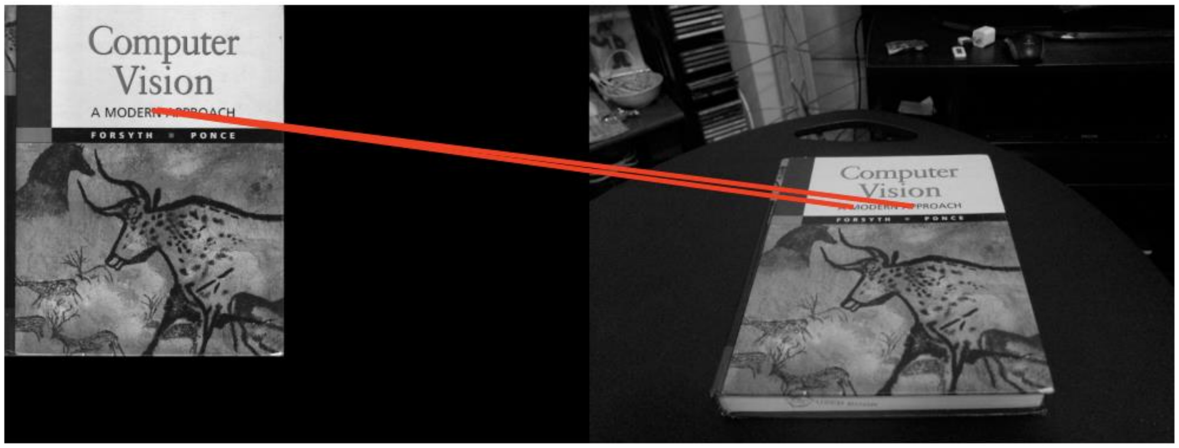
2.1.5 Feature Matching and Parameter Tuning

Below are the results of running displayMatch.py with various parameter values. From tuning the parameters, I found that increasing the ratio parameter (ratio for BRIEF feature descriptor) creates more matches since it would detect more interest points. Decreasing the ratio parameters gives us fewer matches. Increasing the sigma parameter increases the threshold for corner detection using FAST feature detector. A higher sigma value means there would need to be a larger change around each pixel value for it to be determined as an interest point. Therefore, increasing sigma would lead to fewer matches, while decreasing sigma would lead to more matches.

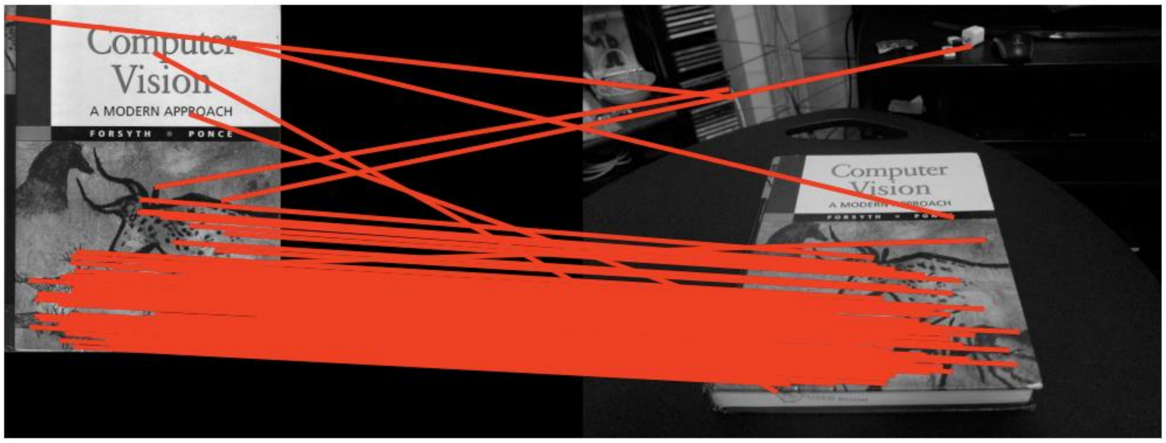
sigma = 0.15, ratio = 0.7



sigma = 0.5, ratio=0.7

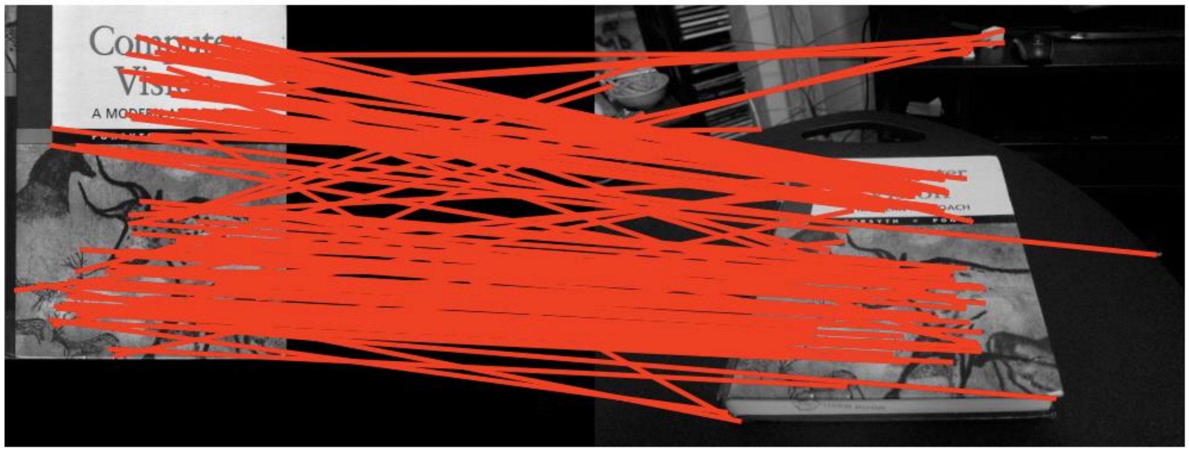


sigma = 0.05, ratio=0.7

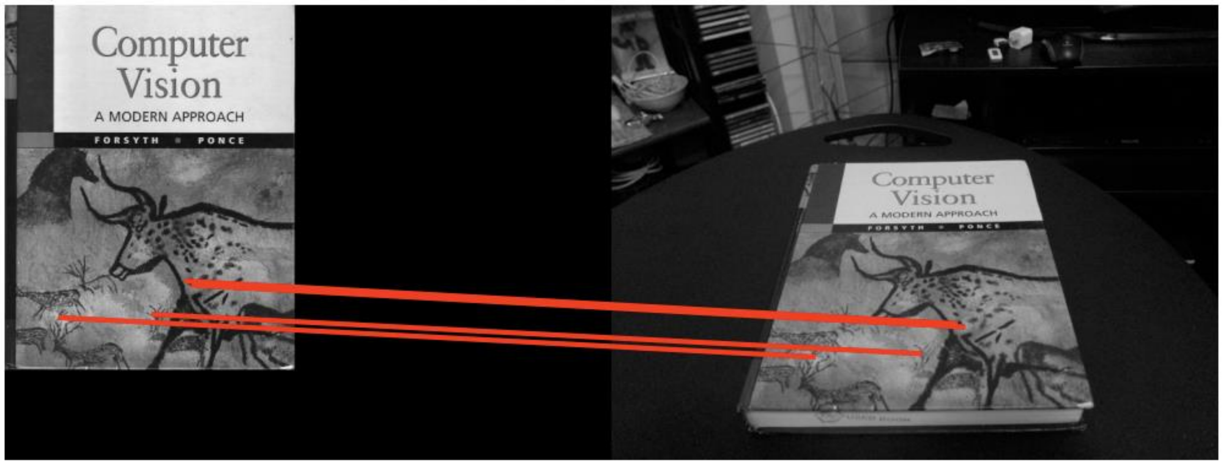


sigma = 0.15, ratio = 0.2 → ratio is too small so there are no matches

sigma=0.15, ratio = 0.9

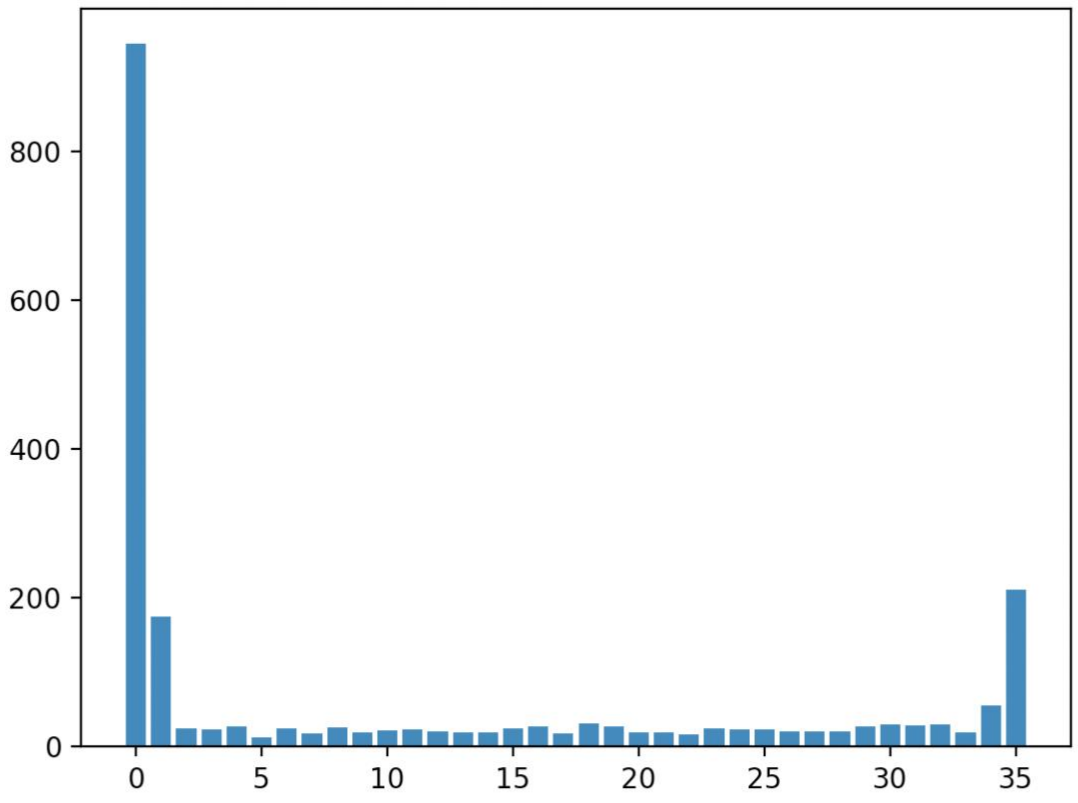


sigma = 0.15, ratio = 0.5



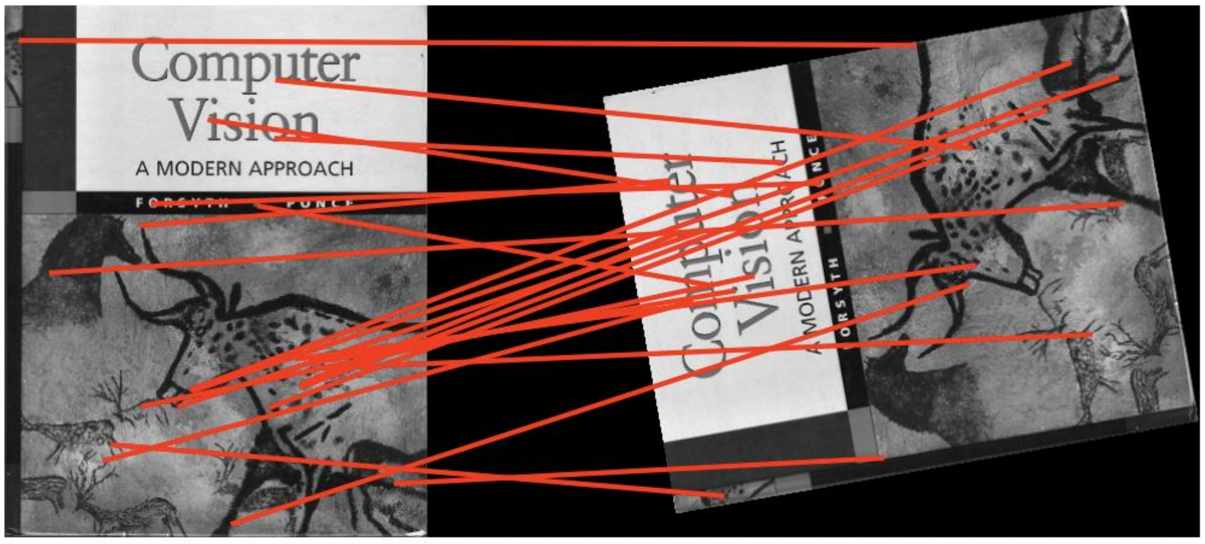
2.1.6 BRIEF and Rotations

Histogram:

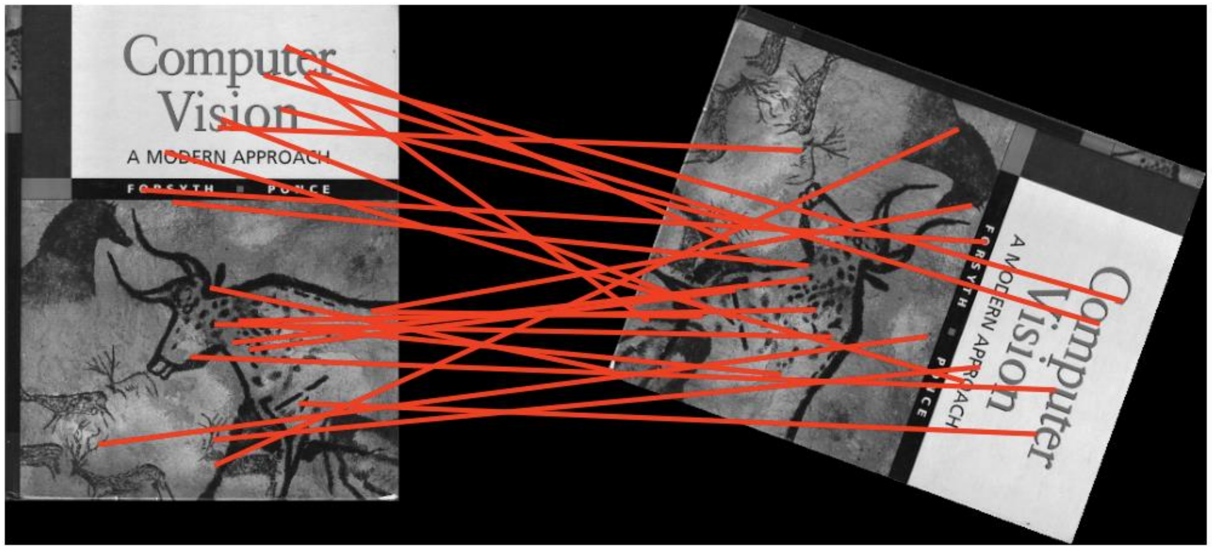


Matching Visualizations:

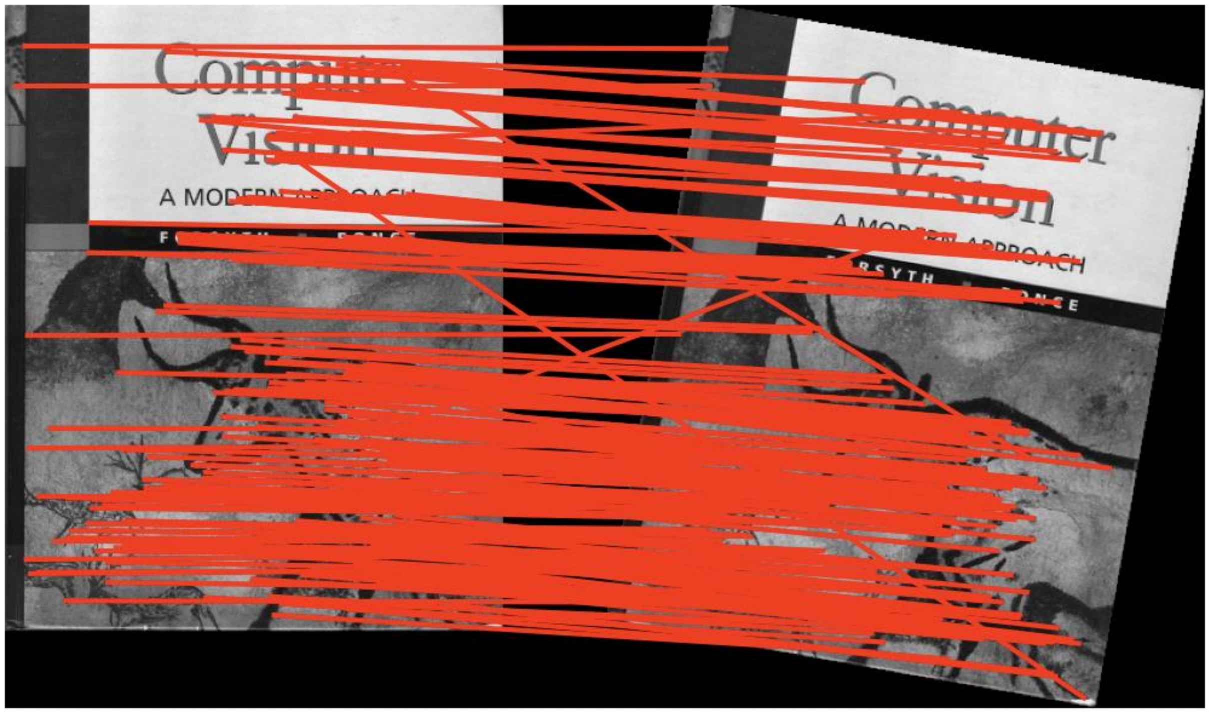
100 degree rotation



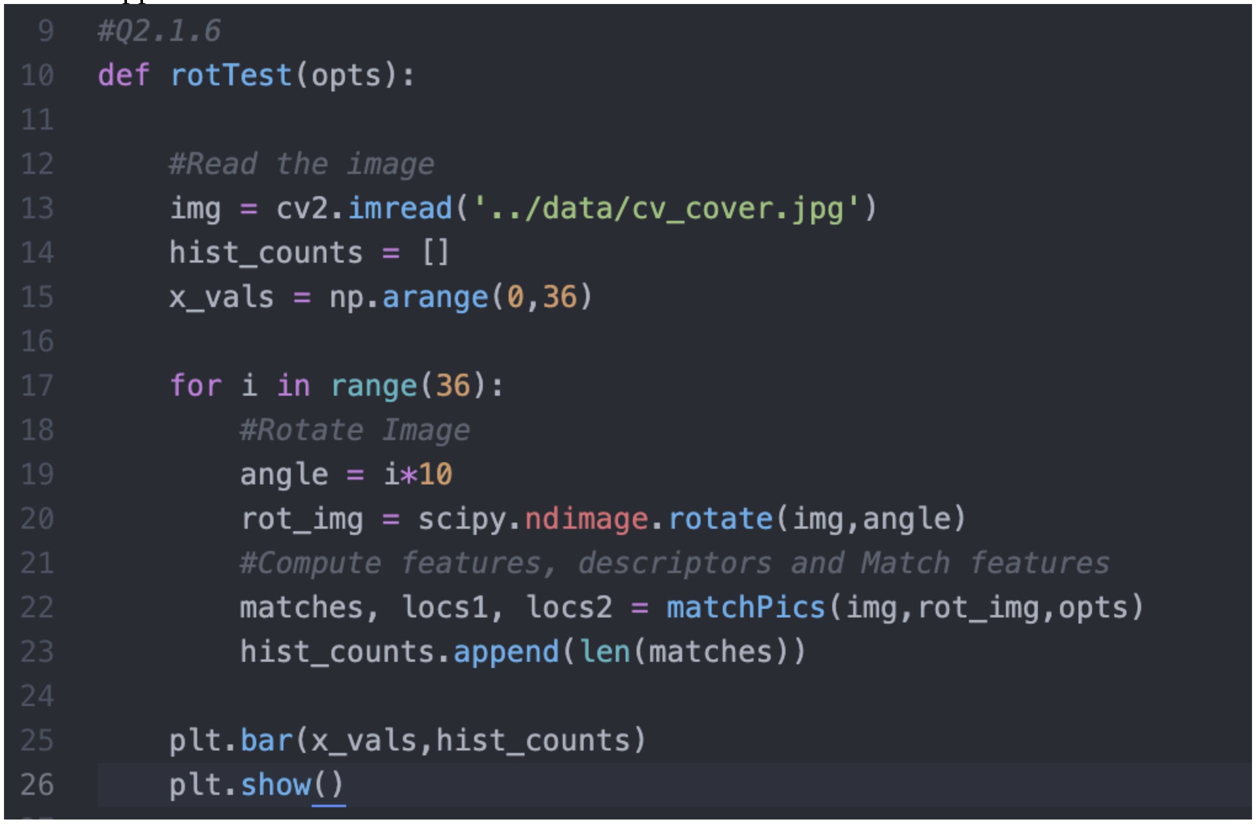
250 degree rotation



350 degree rotation



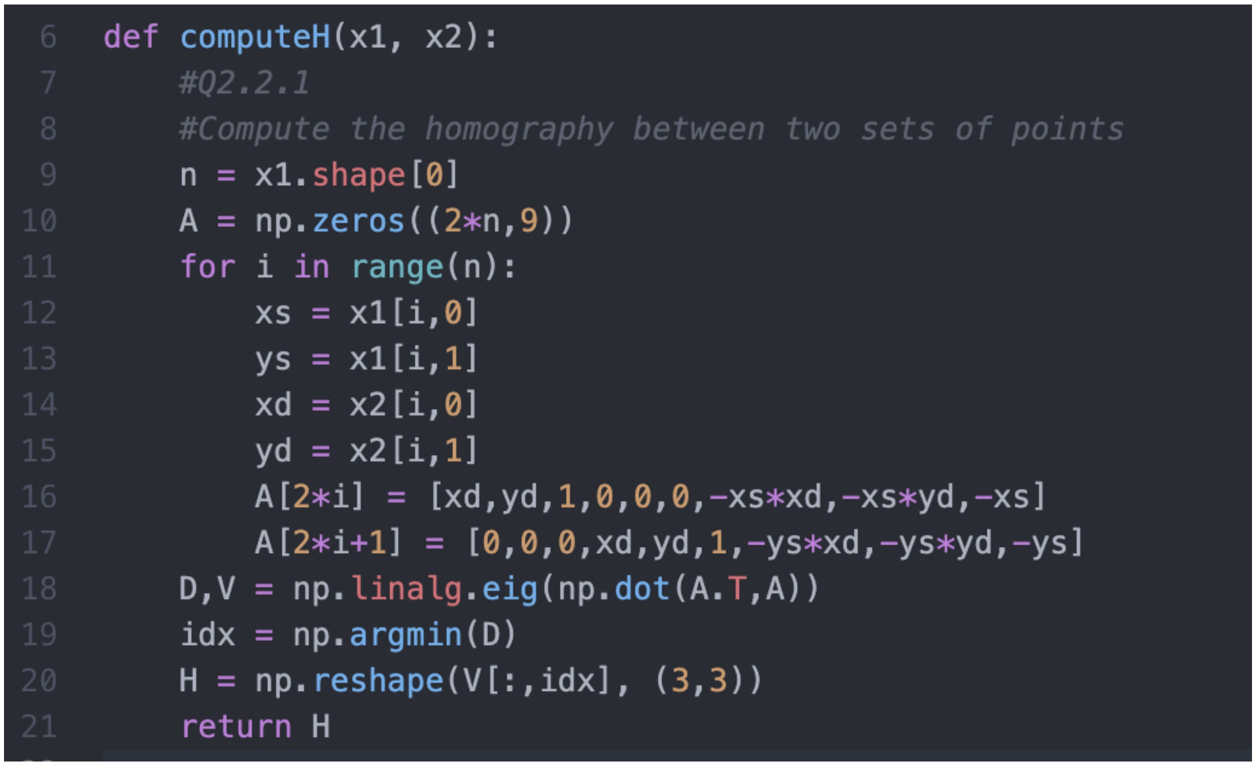
Explanation:
We can see from the histogram as well as the match visualizations of the rotated images that BRIEF is not rotation invariant. Therefore, there are many more matches when the image is rotated slightly from the original image (such as at 10 degrees or 350 degrees) than when the image is rotated to a very different orientation.

Code Snippet:


```python
#Q2.1.6
def rotTest(opts):

    #Read the image
    img = cv2.imread('../data/cv_cover.jpg')
    hist_counts = []
    x_vals = np.arange(0,36)

    for i in range(36):
        #Rotate Image
        angle = i*10
        rot_img = scipy.ndimage.rotate(img,angle)
        #Compute features, descriptors and Match features
        matches, locs1, locs2 = matchPics(img,rot_img,opts)
        hist_counts.append(len(matches))

    plt.bar(x_vals,hist_counts)
    plt.show()
```

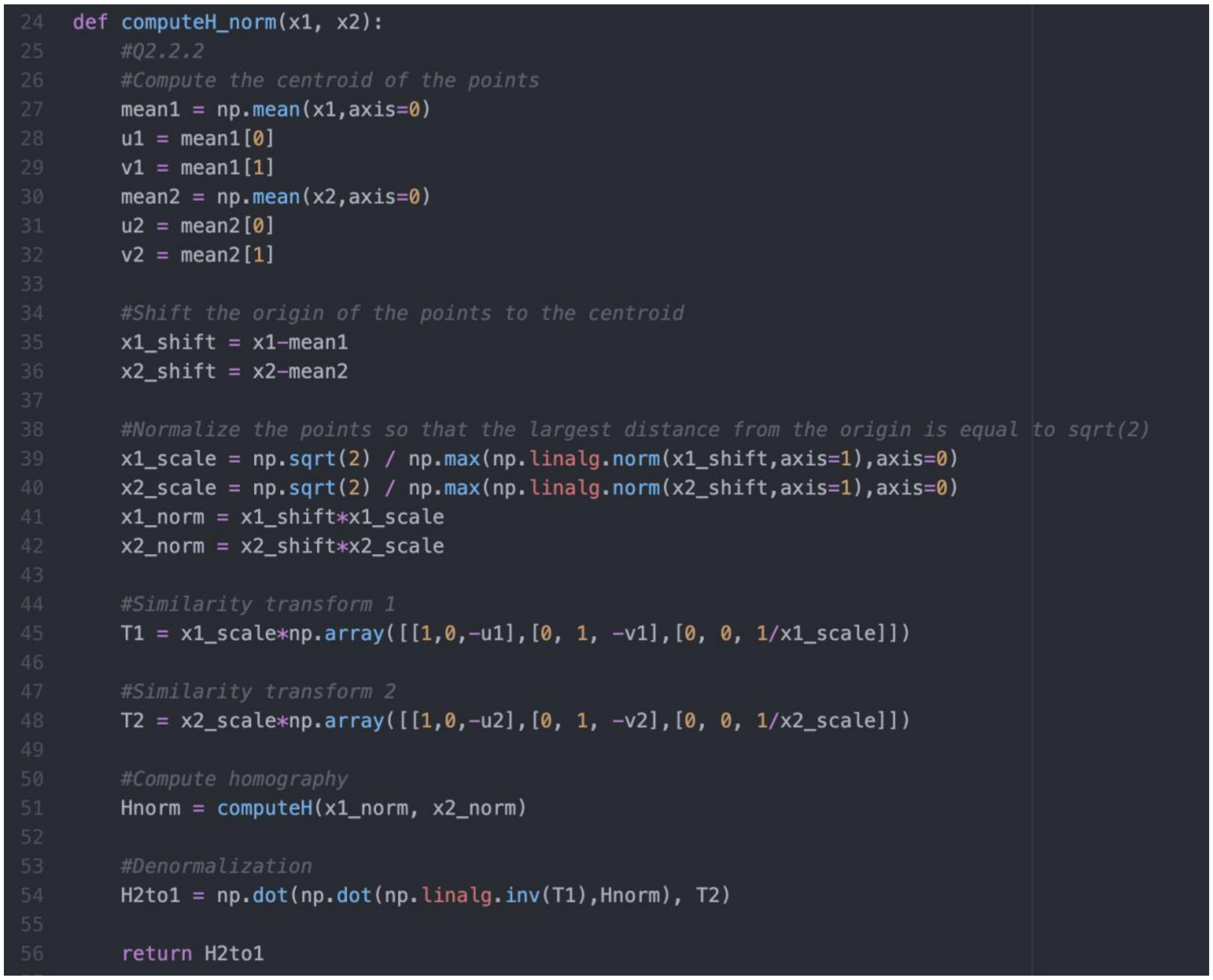
2.2 Homography Computation

2.2.1 Computing the Homography



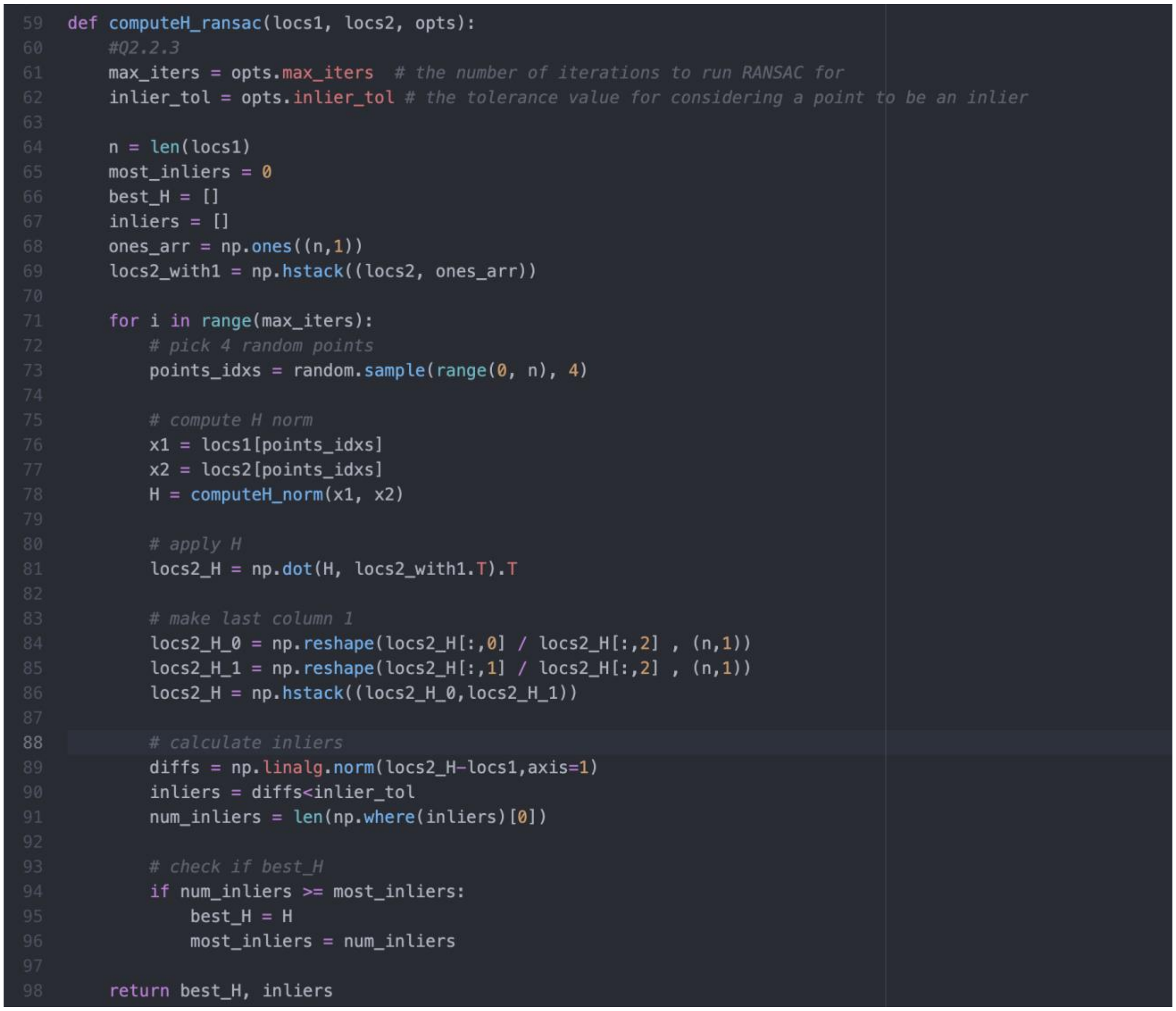
```
def computeH(x1, x2):
    #Q2.2.1
    #Compute the homography between two sets of points
    n = x1.shape[0]
    A = np.zeros((2*n,9))
    for i in range(n):
        xs = x1[i,0]
        ys = x1[i,1]
        xd = x2[i,0]
        yd = x2[i,1]
        A[2*i] = [xd,yd,1,0,0,0,-xs*xd,-xs*yd,-xs]
        A[2*i+1] = [0,0,0,xd,yd,1,-ys*xd,-ys*yd,-ys]
    D,V = np.linalg.eig(np.dot(A.T,A))
    idx = np.argmin(D)
    H = np.reshape(V[:,idx], (3,3))
    return H
```

2.2.2 Homography Normalization



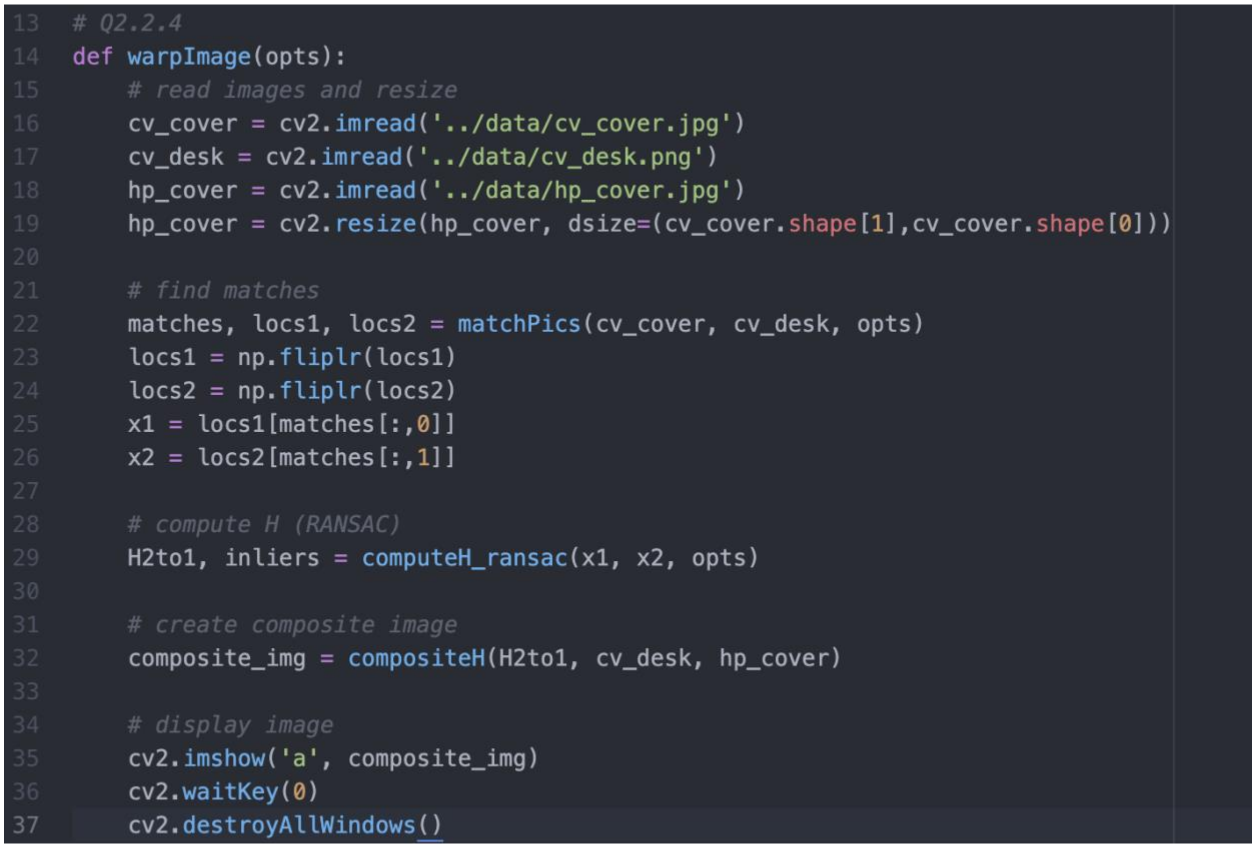
```python
def computeH_norm(x1, x2):
    #Q2.2.2
    #Compute the centroid of the points
    mean1 = np.mean(x1,axis=0)
    u1 = mean1[0]
    v1 = mean1[1]
    mean2 = np.mean(x2,axis=0)
    u2 = mean2[0]
    v2 = mean2[1]

    #Shift the origin of the points to the centroid
    x1_shift = x1-mean1
    x2_shift = x2-mean2

    #Normalize the points so that the largest distance from the origin is equal to sqrt(2)
    x1_scale = np.sqrt(2) / np.max(np.linalg.norm(x1_shift,axis=1),axis=0)
    x2_scale = np.sqrt(2) / np.max(np.linalg.norm(x2_shift,axis=1),axis=0)
    x1_norm = x1_shift*x1_scale
    x2_norm = x2_shift*x2_scale

    #Similarity transform 1
    T1 = x1_scale*np.array([[1,0,-u1],[0, 1, -v1],[0, 0, 1/x1_scale]])

    #Similarity transform 2
    T2 = x2_scale*np.array([[1,0,-u2],[0, 1, -v2],[0, 0, 1/x2_scale]])

    #Compute homography
    Hnorm = computeH(x1_norm, x2_norm)

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

    return H2to1
```

2.2.3 Implement RANSAC



```python
def computeH_ransac(locs1, locs2, opts):
    #Q2.2.3
    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

    n = len(locs1)
    most_inliers = 0
    best_H = []
    inliers = []
    ones_arr = np.ones((n,1))
    locs2_with1 = np.hstack((locs2, ones_arr))

    for i in range(max_iters):
        # pick 4 random points
        points_idxs = random.sample(range(0, n), 4)

        # compute H norm
        x1 = locs1[points_idxs]
        x2 = locs2[points_idxs]
        H = computeH_norm(x1, x2)

        # apply H
        locs2_H = np.dot(H, locs2_with1.T).T

        # make last column 1
        locs2_H_0 = np.reshape(locs2_H[:,0] / locs2_H[:,2] , (n,1))
        locs2_H_1 = np.reshape(locs2_H[:,1] / locs2_H[:,2] , (n,1))
        locs2_H = np.hstack((locs2_H_0,locs2_H_1))

        # calculate inliers
        diffs = np.linalg.norm(locs2_H-locs1,axis=1)
        inliers = diffs<inlier_tol
        num_inliers = len(np.where(inliers)[0])

        # check if best_H
        if num_inliers >= most_inliers:
            best_H = H
            most_inliers = num_inliers

    return best_H, inliers
```

2.2.4 Automated Homography Estimation and Warping

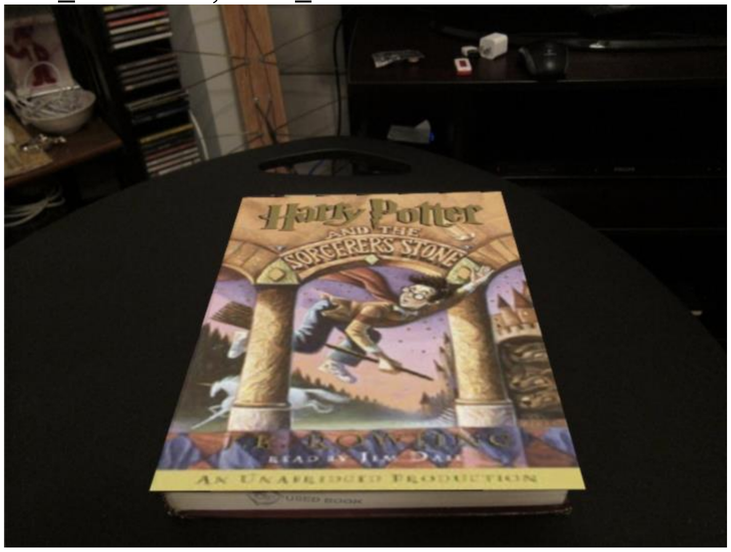


```python
# Q2.2.4
def warpImage(opts):
    # read images and resize
    cv_cover = cv2.imread('../data/cv_cover.jpg')
    cv_desk = cv2.imread('../data/cv_desk.png')
    hp_cover = cv2.imread('../data/hp_cover.jpg')
    hp_cover = cv2.resize(hp_cover, dsize=(cv_cover.shape[1],cv_cover.shape[0]))

    # find matches
    matches, locs1, locs2 = matchPics(cv_cover, cv_desk, opts)
    locs1 = np.fliplr(locs1)
    locs2 = np.fliplr(locs2)
    x1 = locs1[matches[:,0]]
    x2 = locs2[matches[:,1]]

    # compute H (RANSAC)
    H2to1, inliers = computeH_ransac(x1, x2, opts)

    # create composite image
    composite_img = compositeH(H2to1, cv_desk, hp_cover)

    # display image
    cv2.imshow('a', composite_img)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

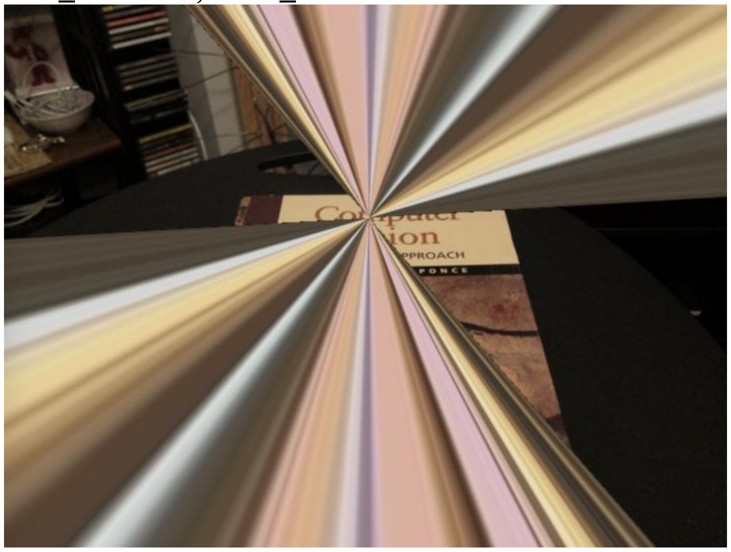
2.2.5 RANSAC Parameter Tuning

The inlier tolerance (inlier_tol) represents the tolerance value for considering a point to be an inlier. Increasing this tolerance would lead to more inliers that may not be as accurate of a match for the homography under consideration. Decreasing the tolerance would reduce the number of inliers but increase the accuracy in determining how correctly that point corresponds to the homography. The number of iterations (max_iters) represents how many times to run RANSAC for. Increasing max_iters increases the likelihood of finding an accurate homography because there are more random samples of points to use and more potential homographies to check. Decreasing max_iters reduces the likelihood of finding an accurate homography because there are fewer potential homographies to pick from, so the results may not work out. Below are result images of changing these two parameters.

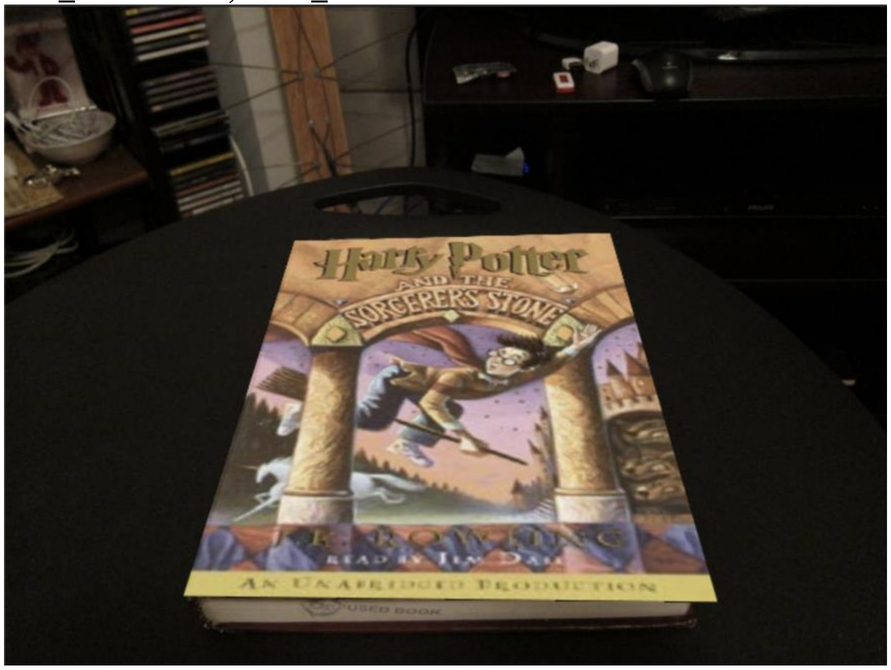
max_iters = 500, inlier_tol=2



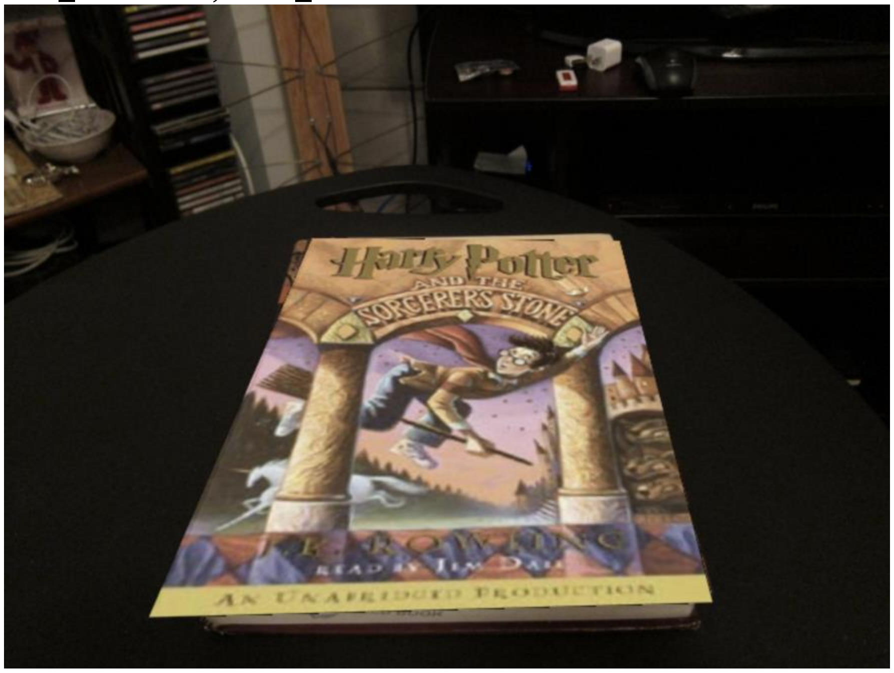
max_iters = 50, inlier_tol=2 → failed because max iters was too small



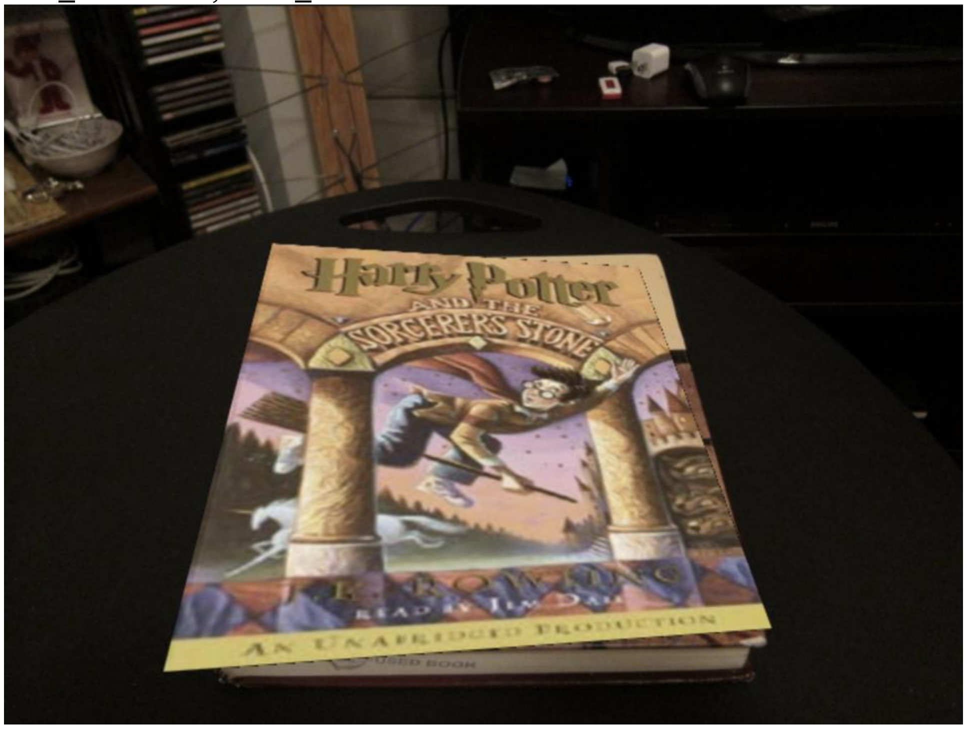
max_iters = 2000, inlier_tol=2



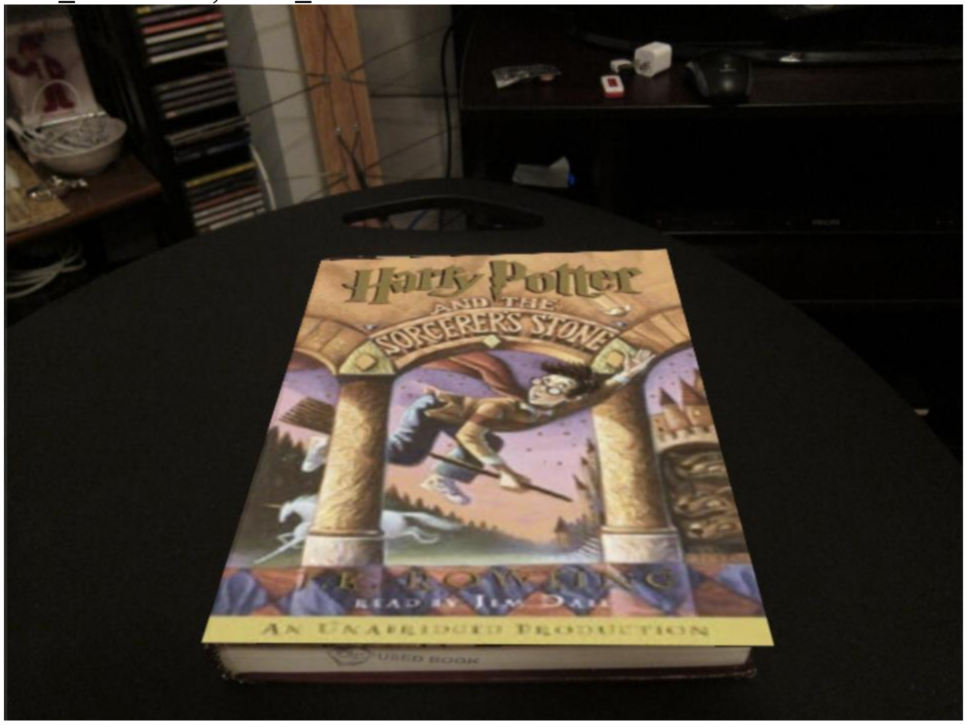
max_iters = 500, inlier_tol = 20



max_iters = 500, inlier_tol = 50

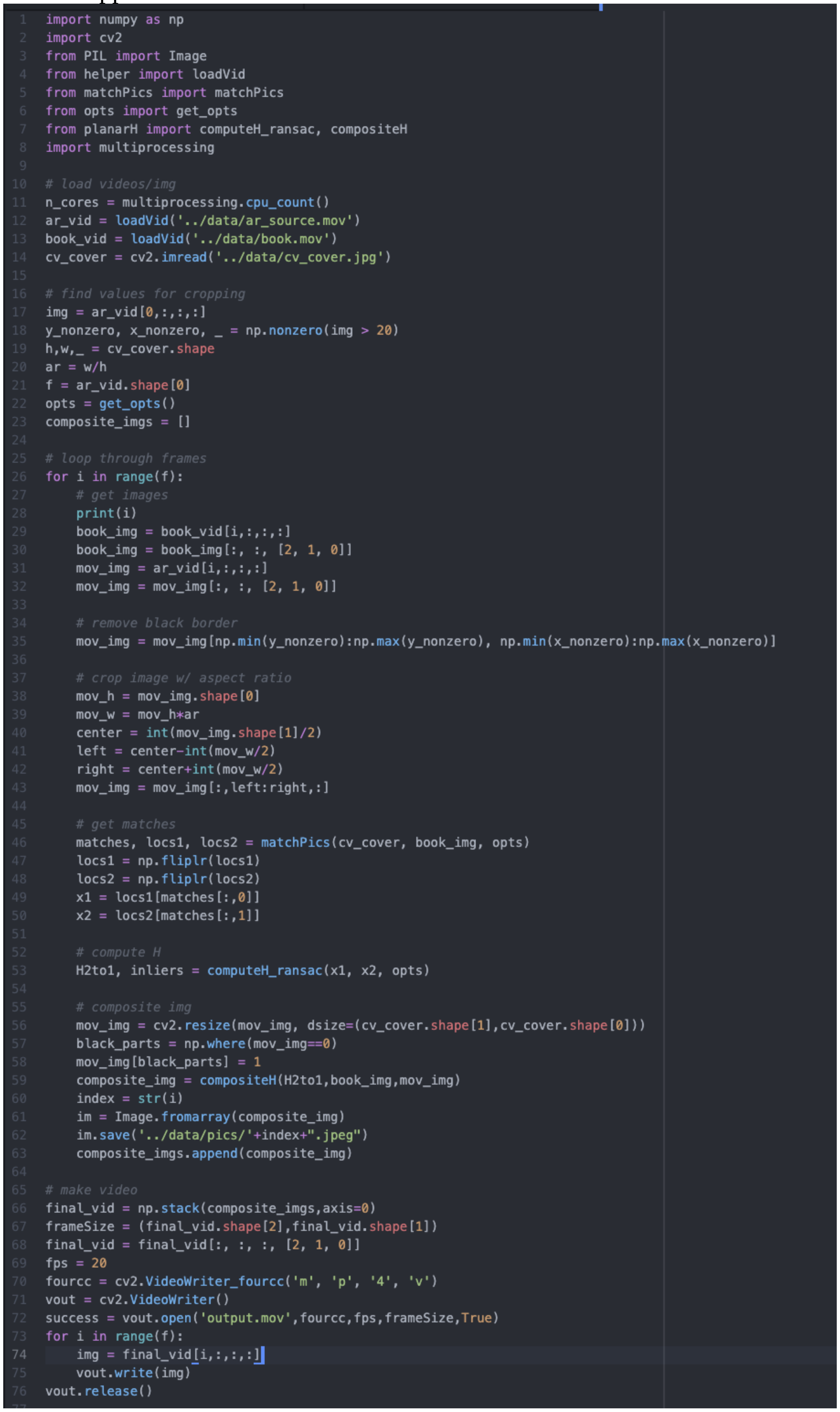


max_iters = 500, inlier_tol = 0.5



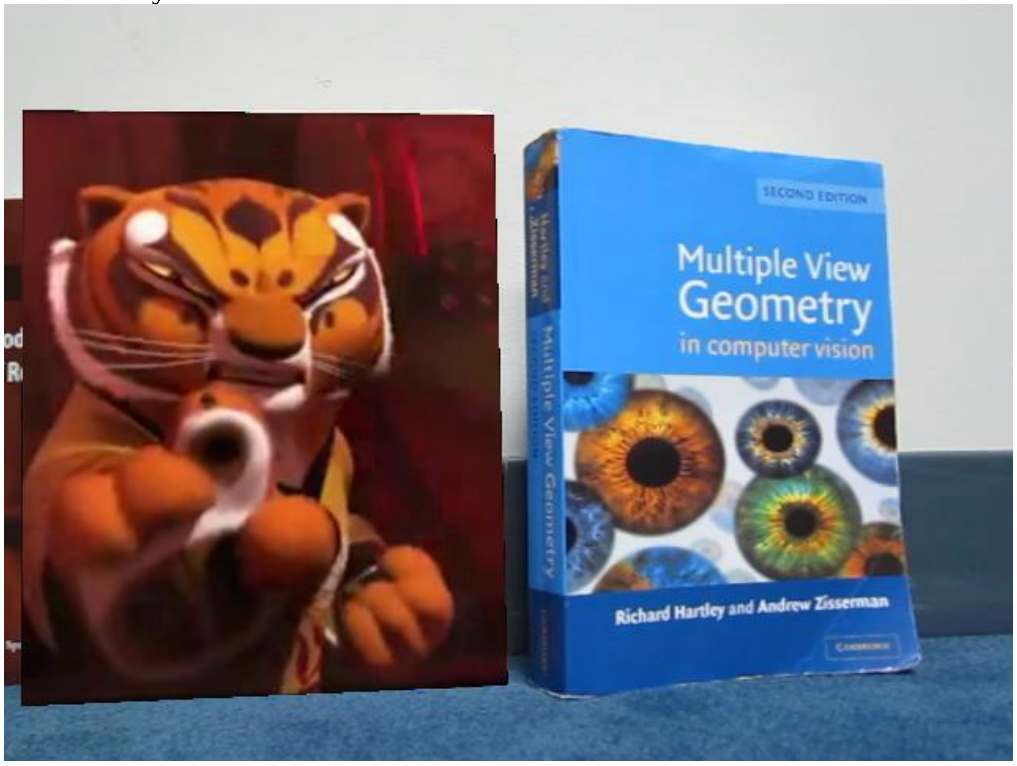
3.1 Incorporating Video

Code Snippet:

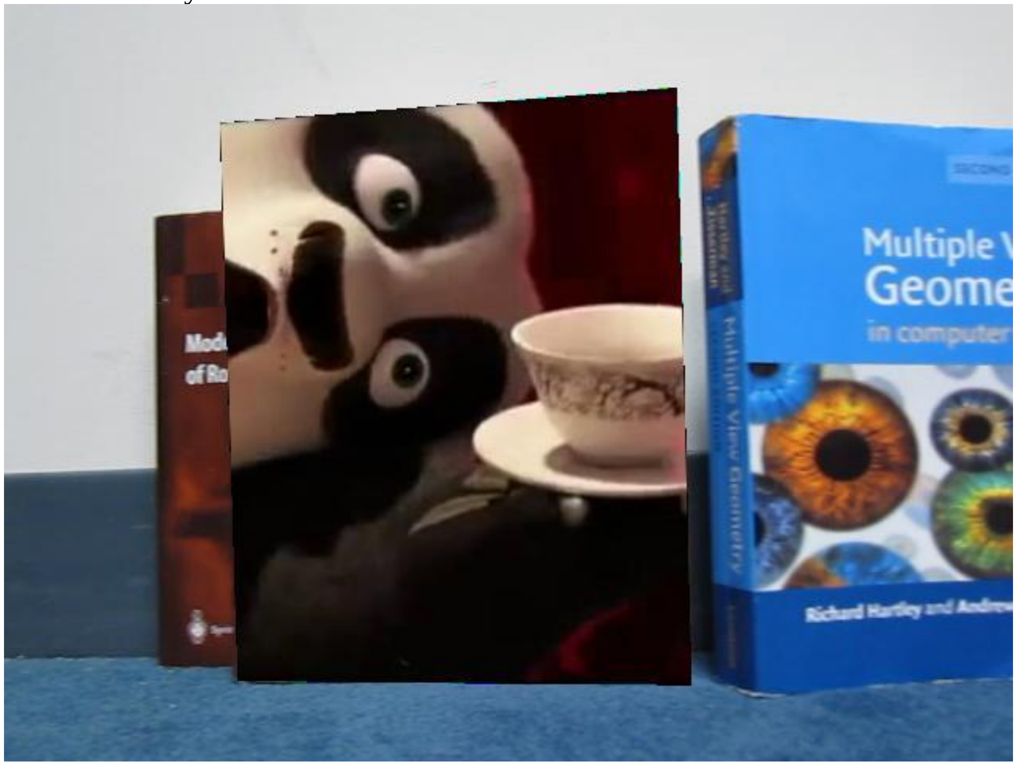


```python
import numpy as np
import cv2
from PIL import Image
from helper import loadVid
from matchPics import matchPics
from opts import get_opts
from planarH import computeH_ransac, compositeH
import multiprocessing

# load videos/img
n_cores = multiprocessing.cpu_count()
ar_vid = loadVid('../data/ar_source.mov')
book_vid = loadVid('../data/book.mov')
cv_cover = cv2.imread('../data/cv_cover.jpg')

# find values for cropping
img = ar_vid[0,:,:,:]
y_nonzero, x_nonzero, _ = np.nonzero(img > 20)
h,w,_ = cv_cover.shape
ar = w/h
f = ar_vid.shape[0]
opts = get_opts()
composite_imgs = []

# loop through frames
for i in range(f):
    # get images
    print(i)
    book_img = book_vid[i,:,:,:]
    book_img = book_img[:, :, [2, 1, 0]]
    mov_img = ar_vid[i,:,:,:]
    mov_img = mov_img[:, :, [2, 1, 0]]

    # remove black border
    mov_img = mov_img[np.min(y_nonzero):np.max(y_nonzero), np.min(x_nonzero):np.max(x_nonzero)]

    # crop image w/ aspect ratio
    mov_h = mov_img.shape[0]
    mov_w = mov_h*ar
    center = int(mov_img.shape[1]/2)
    left = center-int(mov_w/2)
    right = center+int(mov_w/2)
    mov_img = mov_img[:,left:right,:]

    # get matches
    matches, locs1, locs2 = matchPics(cv_cover, book_img, opts)
    locs1 = np.fliplr(locs1)
    locs2 = np.fliplr(locs2)
    x1 = locs1[matches[:,0]]
    x2 = locs2[matches[:,1]]

    # compute H
    H2to1, inliers = computeH_ransac(x1, x2, opts)

    # composite img
    mov_img = cv2.resize(mov_img, dsize=(cv_cover.shape[1],cv_cover.shape[0]))
    black_parts = np.where(mov_img==0)
    mov_img[black_parts] = 1
    composite_img = compositeH(H2to1,book_img,mov_img)
    index = str(i)
    im = Image.fromarray(composite_img)
    im.save('../data/pics/'+index+".jpeg")
    composite_imgs.append(composite_img)

# make video
final_vid = np.stack(composite_imgs,axis=0)
frameSize = (final_vid.shape[2],final_vid.shape[1])
final_vid = final_vid[:, :, :, [2, 1, 0]]
fps = 20
fourcc = cv2.VideoWriter_fourcc('m', 'p', '4', 'v')
vout = cv2.VideoWriter()
success = vout.open('output.mov',fourcc,fps,frameSize,True)
for i in range(f):
    img = final_vid[i,:,:,:]
    vout.write(img)
vout.release()
```

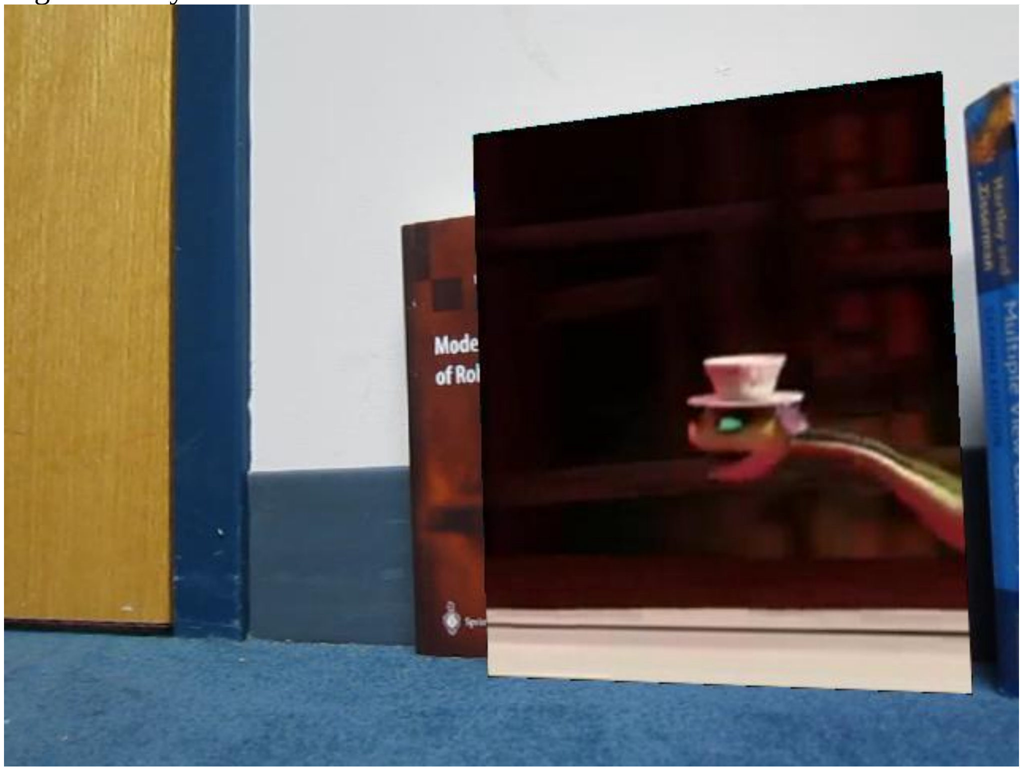
Left Overlay Video Frame:


SECOND EDITION
Multiple View Geometry in computer vision
Richard Hartley and Andrew Zisserman
Multiple View Geometry
Hartley and Zisserman

Center Overlay Video Frame:



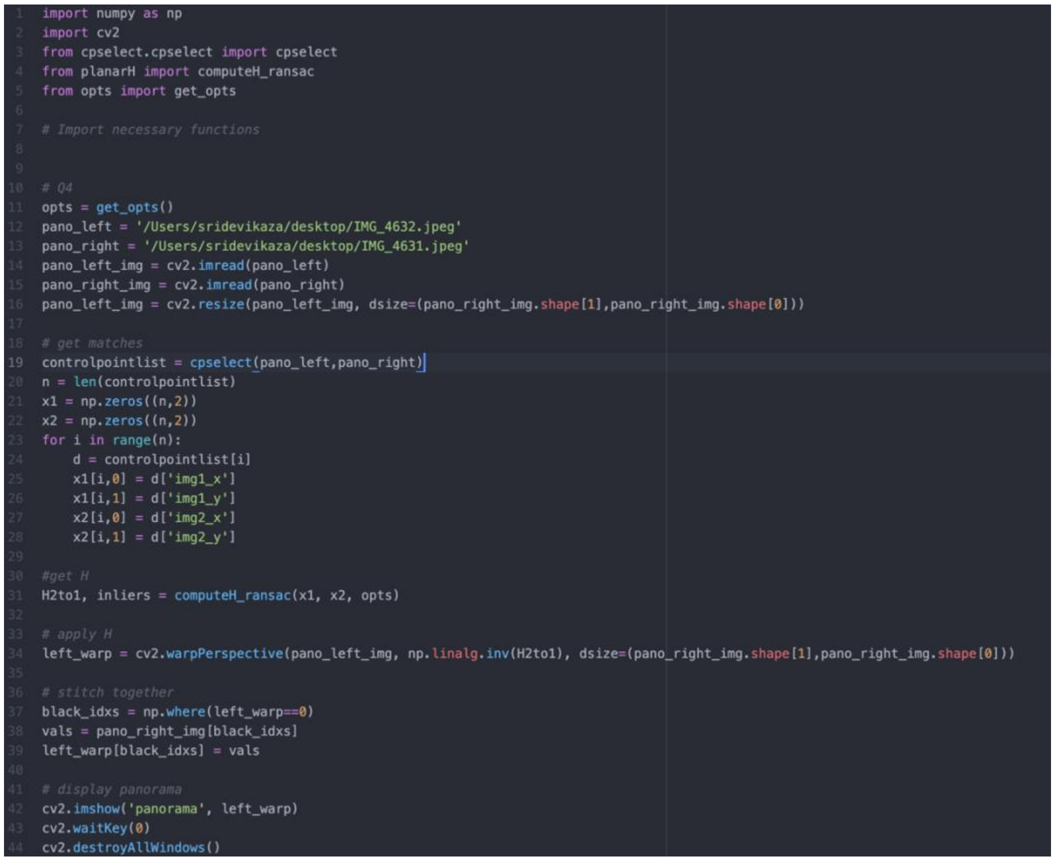
Right Overlay Video Frame:



Drive link to video:
https://drive.google.com/file/d/1GovA7YG4seAmOvVqTBZM6kREBkXbUuM2/view?usp=sharing

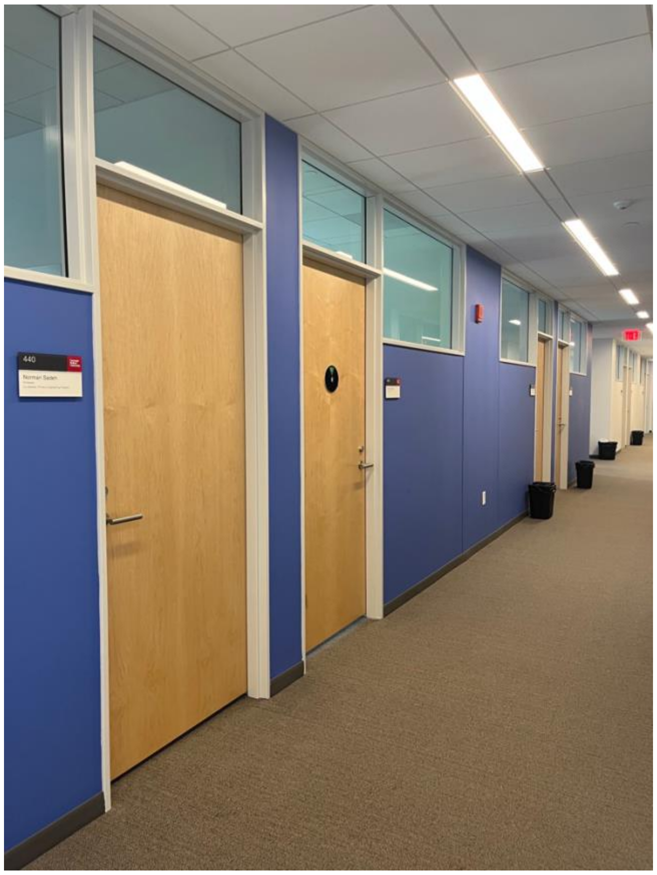
4 Create a Simple Panorama

Code Snippet:

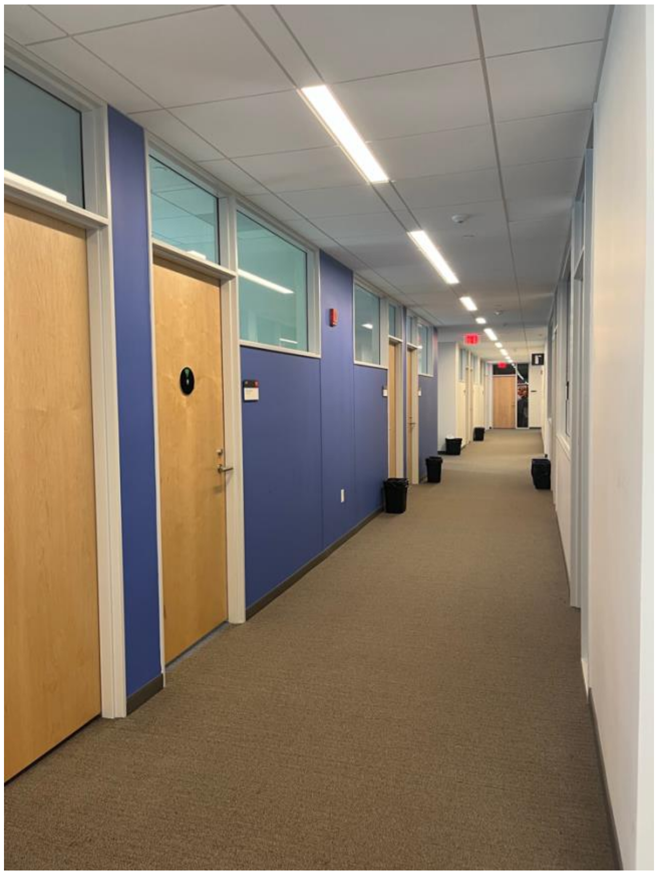


```python
import numpy as np
import cv2
from cpselect.cpselect import cpselect
from planarH import computeH_ransac
from opts import get_opts

# Import necessary functions



# Q4
opts = get_opts()
pano_left = '/Users/sridevikaza/desktop/IMG_4632.jpeg'
pano_right = '/Users/sridevikaza/desktop/IMG_4631.jpeg'
pano_left_img = cv2.imread(pano_left)
pano_right_img = cv2.imread(pano_right)
pano_left_img = cv2.resize(pano_left_img, dsize=(pano_right_img.shape[1],pano_right_img.shape[0]))

# get matches
controlpointlist = cpselect(pano_left,pano_right)
n = len(controlpointlist)
x1 = np.zeros((n,2))
x2 = np.zeros((n,2))
for i in range(n):
    d = controlpointlist[i]
    x1[i,0] = d['img1_x']
    x1[i,1] = d['img1_y']
    x2[i,0] = d['img2_x']
    x2[i,1] = d['img2_y']

#get H
H2to1, inliers = computeH_ransac(x1, x2, opts)

# apply H
left_warp = cv2.warpPerspective(pano_left_img, np.linalg.inv(H2to1), dsize=(pano_right_img.shape[1],pano_right_img.shape[0]))

# stitch together
black_idxs = np.where(left_warp==0)
vals = pano_right_img[black_idxs]
left_warp[black_idxs] = vals

# display panorama
cv2.imshow('panorama', left_warp)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Original Left Image:



Original Right Image:



Panorama: