Computer Vision (16-720 B): Homework 2

Name - Saharsh Agarwal
AndrewID - saharsh2

Question 1

Question 1.1: Planar Homographies as a Warp

Answer 1.1

$$x = P\hat{X} \tag{1}$$

where x is image point (homogeneous 3-vector), P is the camera projection matrix (3x4 Dimension) and $\hat{X}$ is world point (homogeneous 4-vector). Since P helps project points from 3D world to the plane of an image captured by camera, the inverse is assumed possible here. Thus, P^{-1} exists and hence, the existence of H is proved below.

$$\hat{X} = X_\pi, \qquad x_1 = P_1 X_\pi, \qquad x_2 = P_2 X_\pi, \tag{2}$$

X_π lies on the plane (given in question and P_1 and P_2 correspond to Camera C and C' respectively. Using equations 2 -

$$X_\pi = P_1^{-1} x_1 = P_2^{-1} x_2 \qquad => \qquad x_2 = P_2 P_1^{-1} x_1, \qquad x_1 = P_1 P_2^{-1} x_2 \tag{3}$$

Since x_1 and x_2 are homogeneous coordinates and can be scaled to arbitrary value by scalar λ -

$$x_1 = \lambda P_1 P_2^{-1} x_2, \qquad x_1 \propto P_1 P_2^{-1} x_2, \qquad x_1 \equiv H x_2 \tag{4}$$

Hence, proved that there exists an H (=$\lambda P_1 P_2^{-1}$) such that $x_1 \equiv$ Hx_2.

Question 1.2: Direct Linear Transform

1. How many degrees of freedom does "h"have ?

Ans - h has 8 degrees of freedom. (Shown below)

H is a 3x3 dimension matrix. h is flattened column matrix of H, thus, of dimension 9x1.

Since x_1^i and x_2^i are homogeneous form of the 2D coordinates and we can remove the arbitrary scaling factor from h matrix, we have one less parameter to deal with. Hence, h is normalised and has only 8 degrees of freedom (proved later).

2. How many point pairs are required to solve "h"?

Ans - 4 Pairs required. (Shown below)

There are 8 parameters in the normalised **h**. Thus, it requires 8 equations to be fully determined. Each pair of points contribute to 2 equations and thus, 4 pairs of points are required to solve for h.

3. Derive A_i.

Proof for Q1.2 - 1 and 2 also in this writeup. **Ans** -

$$H = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}, \qquad x_1^i \equiv \lambda H x_2^i \qquad h = \begin{bmatrix} a & b & c & d & e & f & g & h & i \end{bmatrix}^T \tag{5}$$

x_1^i and x_2^i are matching pairs and for convenience shown as (p,q) and (u,v) coordinates. Using Direct Linear Transformation:

$$p = \lambda(au+bv+c), \qquad q = \lambda(du+ev+f), \qquad 1 = \lambda(gu+hv+i) \tag{6}$$

Dividing p and q with 1, to remove the scaling factor λ-

$$p(gu+hv+i) = (au+bv+c), \qquad q(gu+hv+i) = (du+ev+f), \tag{7}$$

Rearranging and simplifying -

$$g(pu)+h(pv)+i(p)-a(u)-b(v)-c=0, \qquad g(qu)+h(qv)+i(q)-d(u)-e(v)-f=0 \tag{8}$$

For the given h, $A^i h = 0$:

$$A^i = \begin{bmatrix} -u & -v & -1 & 0 & 0 & 0 & pu & pv & p \\ 0 & 0 & 0 & -u & -v & -1 & qu & qv & q \end{bmatrix} \tag{9}$$

Since each point correspondence provides 2 equations, 4 such pairs will be required minimum to determine h completely.

4. Solving for Ah = 0

When solving for Ah = 0, trivial solution of **h** suggests that for the given points the Homography is not possible. Example - In real we have more than 4 point pairs but all have noises. Thus noisy coordinates in the over-determined system may produce a **no Homography possible** result - thereby leaving h=0 as the trivial solution. Hence, we also relax the objective from Ah=0 -> to minimising ||Ah||.

(a)When we have noisy data for greater than or equal to 4 matching points (no colinearity), the rank is still full (overdetermined for N>4). However, h might have trivial solution.

(b) If we have 3 out of the 4 points colinear in both the images, then the rank of A will be less than 8, i.e., not a full rank.

(c)If matching points greater than or equal to 4 and exact points given (no noise and no colinearity) - A has full rank.

Eigenvalues represent the reprojection error. **Smallest eigenvalues of $A^T A$ will not be 0 in the case of trivial solution.** This is because points from 1 image do not perfectly match to another image using any possible homography. This indicates that h selected for the lowest eigenvalue will not be able to make Ah = 0. So **all eigenvalues will be greater than 0** and **no eigenvector can form the basis of the null space of A** (except for a h = trivial solution). The eigenvector corresponding to the smallest eigenvalue is the best homography we can achieve, not perfect.

Question 1.4: Theory Questions

Q 1.4.1

K_i, constant for a camera C_i, is dimension 3x3. X in the homogeneous representation of the 3D point is 4 valued vector. Let x by the mapping of X on C_1 and x' be the mapping on C_2. Since the 2 cameras are separated only by Rotation -

$$x = K_1 \begin{bmatrix} I & 0 \end{bmatrix} X, \qquad x' = K_2 \begin{bmatrix} R & 0 \end{bmatrix} X \tag{10}$$

Using the equations above -

$$X = (K_2 \begin{bmatrix} R & 0 \end{bmatrix})^{-1} x', \qquad => \qquad x = K_1 \begin{bmatrix} I & 0 \end{bmatrix} \begin{bmatrix} R & 0 \end{bmatrix}^{-1} K_2^{-1} x' \tag{11}$$

Since IA = A and assuming scaling factor as λ -

$$x_1 \equiv H x_2 \qquad where \quad H = \lambda K_1 R^{-1} K_2^{-1} \tag{12}$$

Hence, proved te existence of homography H.

Q 1.4.2

Camera C is on plane x-y and rotation θ in the z-axis.

Homography H that maps point (x_1) from 1 position rotated by angle a to another position(x_2) is -

$$H(a) = \begin{bmatrix} cos(a) & -sin(a) & 0 \\ sin(a) & cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{13}$$

for $x_1 \equiv H x_2$. Thus H^2 becomes -

$$=> \begin{bmatrix} cos(a) & -sin(a) & 0 \\ sin(a) & cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix} X \begin{bmatrix} cos(a) & -sin(a) & 0 \\ sin(a) & cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad => \begin{bmatrix} cos^2(a) - sin^2(a) & -2sin(a)cos(a) & 0 \\ 2cos(a)sin(a) & cos^2(a) - sin^2(a) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{14}$$

Using trignometric identities -

$$H^2(a) = \begin{bmatrix} cos(2a) & -sin(2a) & 0 \\ sin(2a) & cos(2a) & 0 \\ 0 & 0 & 1 \end{bmatrix} = H(2a) \tag{15}$$

Hence, proved above - if H is pure rotation then H^2 is the same rotation but with twice the angle

Q 1.4.3

Limitations of Planar Homography -

Planar Homography is completely suficient for the points that are on a world plane (being projected on a 2D plane of canvas). It is also sufficient for all points if the camera motion between the two images is a pure rotation about its center. (If the points captured are very very far from the camera, minor translatory perturbations wouldn't hurt calculations.)

Thus, anything outside the above domain - limits the use of planar Homography.

Q 1.4.4

Without loss of generality, centre of projection can be assumed to be Origin. Assuming z_0 =1 will also not harm. x' and y' coordinates of points on line after being projected onto a plane z=z_0.

Any line in 3D space is -

$$\frac{x-a}{p} = \frac{y-b}{q} = \frac{z-c}{r} \qquad OR \quad x = m_x z + b_x, \quad y = m_y z + b_y \tag{16}$$

Projecting any point A(x,y,z) on the line above onto a plane z=z_0 (viewed from Camera C) -> using similarity of triangles, the point A' on the plane -

$$A'(x', y', z') = (\frac{xz_0}{z}, \frac{yz_0}{z}, z_0) \tag{17}$$

Thus, substituting (x,y,z) from the line equation in terms of x',y',z' - we get,

$$\frac{x'z}{z_0} = m_x z + b_x, \quad \frac{y'z}{z_0} = m_y z + b_y \tag{18}$$

Dividing the 2 euations above, to remove z term:

$$\frac{x' - m_x z_0}{y' - m_y z_0} = \frac{b_x}{b_y} \qquad => \qquad y' = (\frac{b_y}{b_x})x' + z_0(m_y - \frac{b_y}{b_x}m_x) \tag{19}$$

Since b_x, b_y, Z_0, m_x and m_y are constants, x' and y' can be ralated as $y' = mx' + c'$. A(x,y,z) can be any points in the 3D space and on the line shown at the starting of the question. It's projection on the camera canvas (x',y',z') also satisfies the equation of the line, thereby, proving that a line in 3D is projected to a line in 2D.

Question 2 - Computing Planar Homographies

Question 2.1: Feature Detection and Matching

Q 2.1.1 : FAST Detector

FAST feature detector finds features by comparing pixel values with its 16 neighbours . P is a corner if n (=12 originally) out of the 16 are all brighter than $I_p + t$ or are all darker than $I_p - t$ values. (I_p is the pixel intensity and threshold **t** is parameter we can provide). Harris Corner Detector is a slow but good method. Image patch intensity is compared to the Image patch intensities in the neighbouring regions. The minimum for the Sum of Squared Errors is seen. If this minimum is also greater than a threshold, we detect it as a corner. This is intuitive as a corner displays change of intensity in multiple directions, unlike edge or flat region.

FAST Detector, as the name suggests, is indeed computationally more efficient than Harris Corner Detector. It is faster than many other well-known feature extraction methods.

Q 2.1.2 : BRIEF Descriptor - binary robust independent elementary features

BRIEF locally describes the region around the point of interest in a binary format based on pixel intensity comparisons. However, the filter banks in HW1 record how a pixel responds to different filters at different scales along all its channels.

We can use of those filter banks as a descriptor. In fact, Haar Wavelets uses responses of bank of filters as a descriptor.

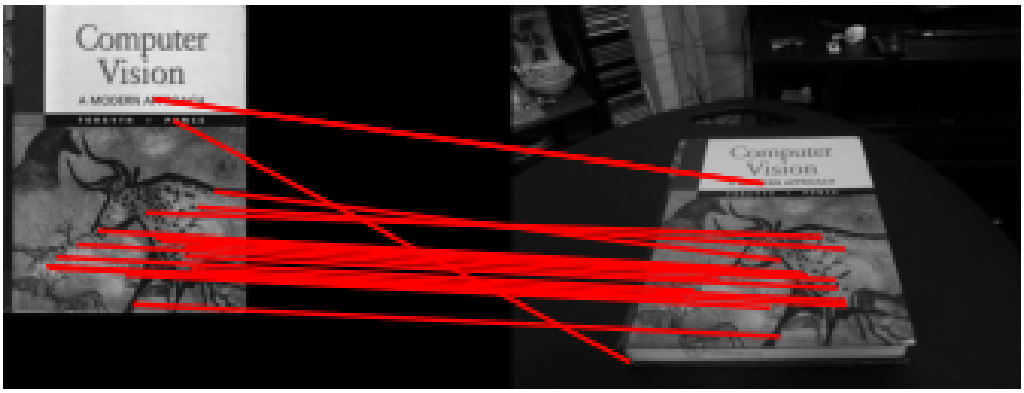
Question 2.1.3 : Matching Methods

Hamming distance between 2 BRIEF descriptors of the same length - is the number of bit positions for which the bits are different (sum of XOR). Lesser Hamming distance suggests better match. So, we find points of interest in 2 images to be matched and defined descriptors for all those positions individually using BRIEF. For each point in the first image, the nearest neighbor in the second image is found using least Hamming Distance. This nearest neighbour is called the match for this point in image 1.

Hamming distance calculation is much faster than calculating Euclidean distance.

Question 2.1.4: Feature Matching

Figur 1: Visualizing Matched points. [Sigma = 0.15, Ratio = 0.7 (Default)]



The code snippet for **matchPics.py** after implementation - (Used **displayMatch.py** to call the **matchPics.py** and plot the image above)

Figur 2: Code Snippet **matchPics.py**

```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'

        # TODO: Convert Images to GrayScale
        real_image1 = I1 # just to keep a copy
        real_image2 = I2
        if (len(I1.shape)>=3):
            I1 = cv2.cvtColor(I1, cv2.COLOR_BGR2GRAY)
        if (len(I2.shape)>=3):
            I2 = cv2.cvtColor(I2, cv2.COLOR_BGR2GRAY)

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

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

        # TODO: Match features using the descriptors
        matches = briefMatch(desc1,desc2,ratio)
        #helper.plotMatches(I1,I2,matches,locs1,locs2)

        return matches, locs1, locs2
```

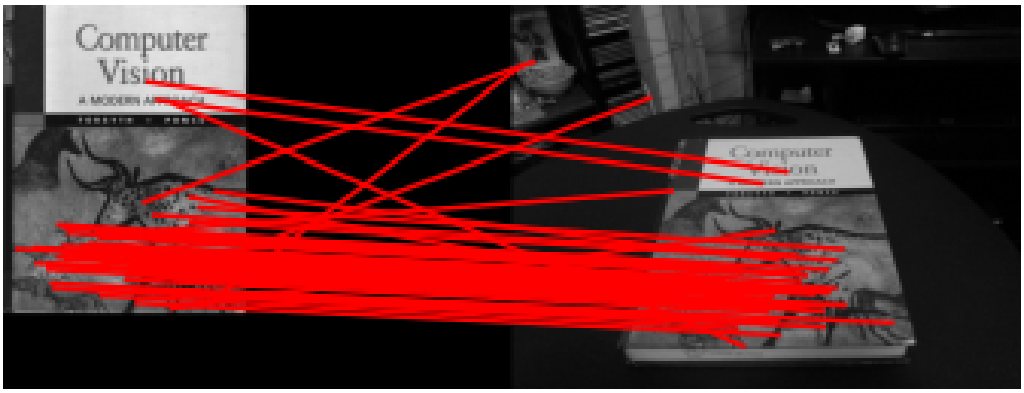
Figur 3: Code Snippet **displayPics.py**

```python
import numpy as np
import cv2
from matchPics import matchPics
from helper import plotMatches
from opts import get_opts


def displayMatched(opts, image1, image2):
    """
    Displays matches between two images

    Input
    -----
    opts: Command line args
    image1, image2: Source images
    """

    matches, locs1, locs2 = matchPics(image1, image2, opts)

    #display matched features
    plotMatches(image1, image2, matches, locs1, locs2)

if __name__ == "__main__":

    opts = get_opts()
    image1 = cv2.imread('../data/cv_cover.jpg')
    image2 = cv2.imread('../data/cv_desk.png')

    displayMatched(opts, image1, image2)
```

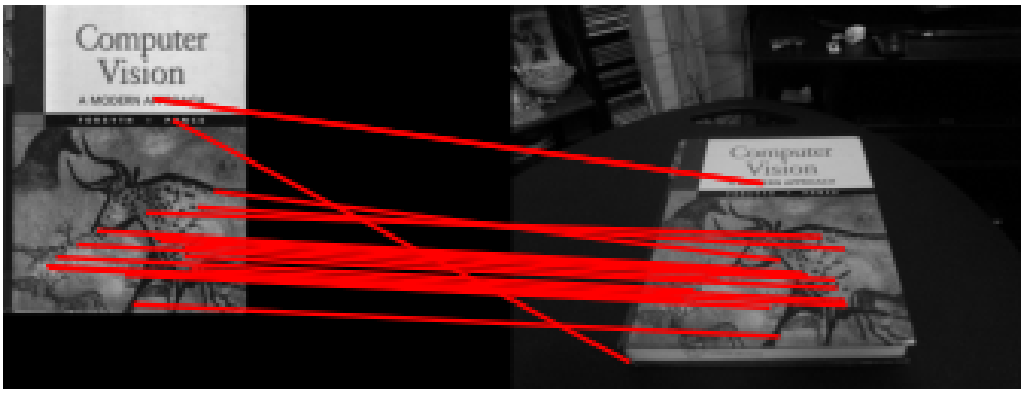
Question 2.1.5: Feature Matching and Parameter Tuning

First, we analyse keeping Ratio constant at 0.7. Sigma values tested for - 0.10, 0.15, 0.20, 0.25

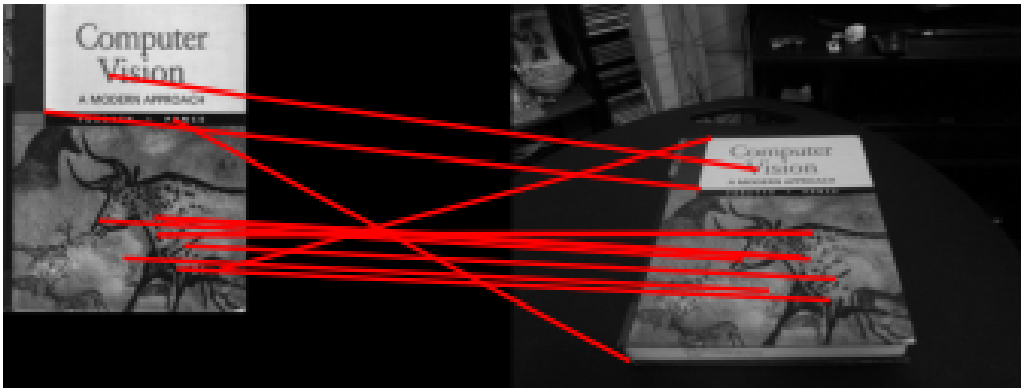
Figur 4: Visualizing Matched points. [Sigma = 0.10, Ratio = 0.7]



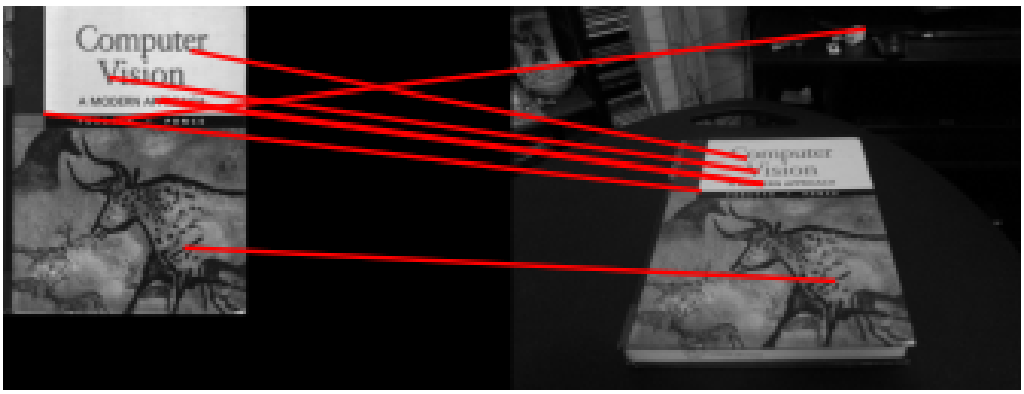
Figur 5: Visualizing Matched points. [Sigma = 0.15, Ratio = 0.7 (Default)]



Figur 6: Visualizing Matched points. [Sigma = 0.20, Ratio = 0.7]



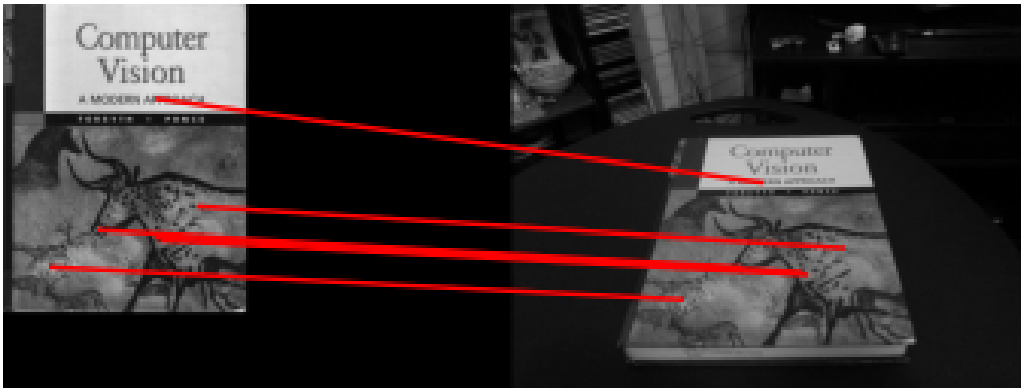
Figur 7: Visualizing Matched points. [Sigma = 0.25, Ratio = 0.7]



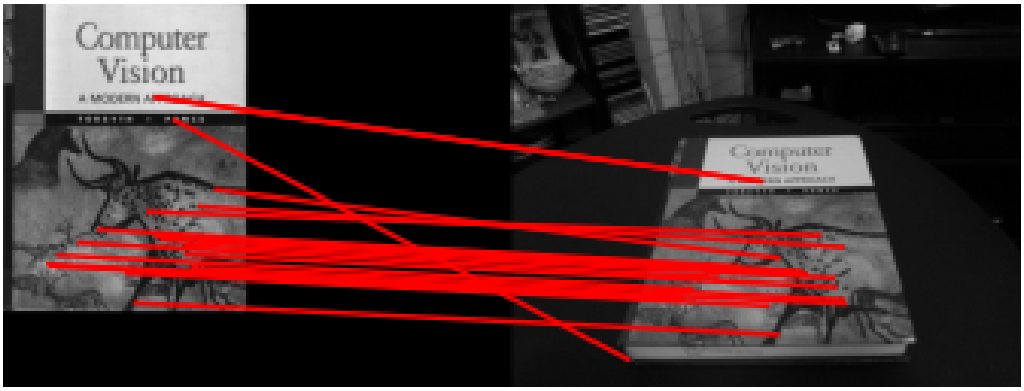
As one can observe from the images- keeping ratio constant, the number of matches decreases as sigma is increased.

Now, keeping sigma constant at 0.15, different ratios are tried. Ratio - 0.6, 0.7 and 0.8

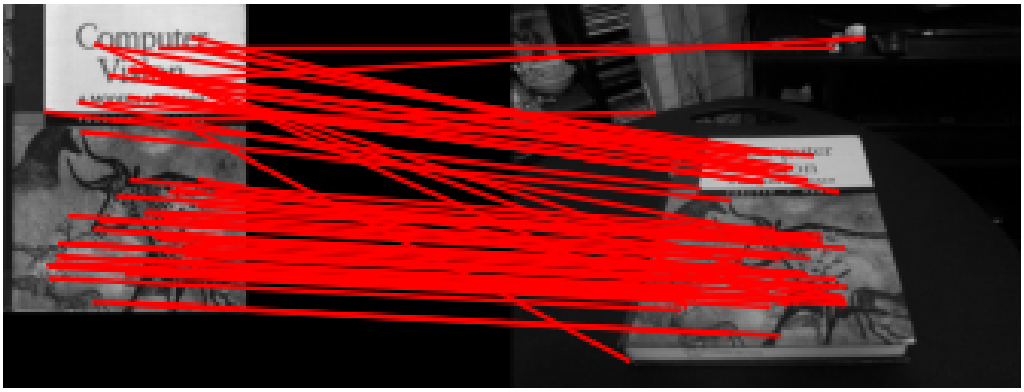
Figur 8: Visualizing Matched points. [Sigma = 0.15, Ratio = 0.6]



Figur 9: Visualizing Matched points. [Sigma = 0.15, Ratio = 0.7 (Default)]



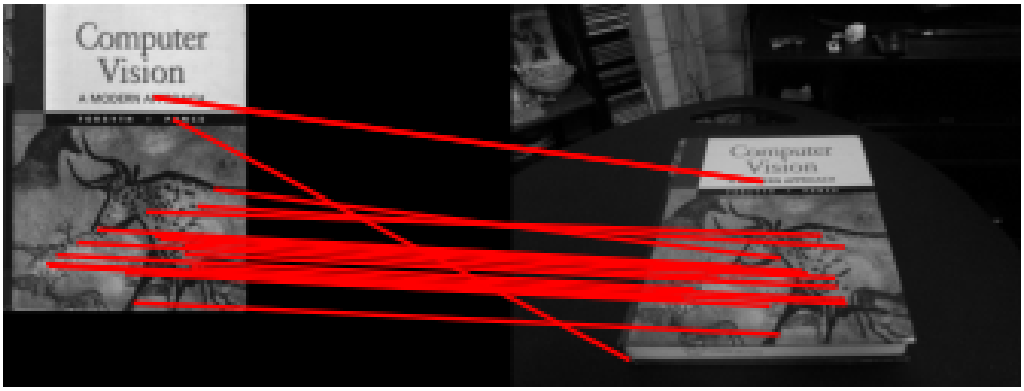
Figur 10: Visualizing Matched points. [Sigma = 0.15, Ratio = 0.8]



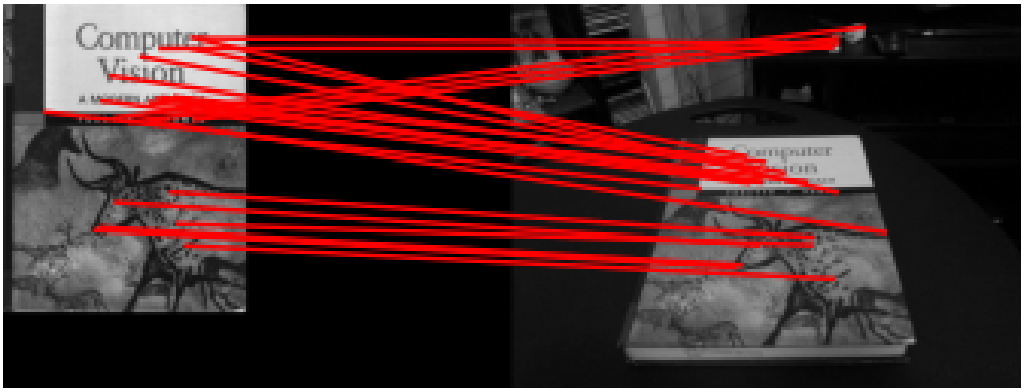
As observed in the images above - as ratio increases, the number of matches increases.

Thus, number of matches has an inverse relation with sigma and a direct relation with ratio. Further observations are made by increasing and decreasing both simultaneously -

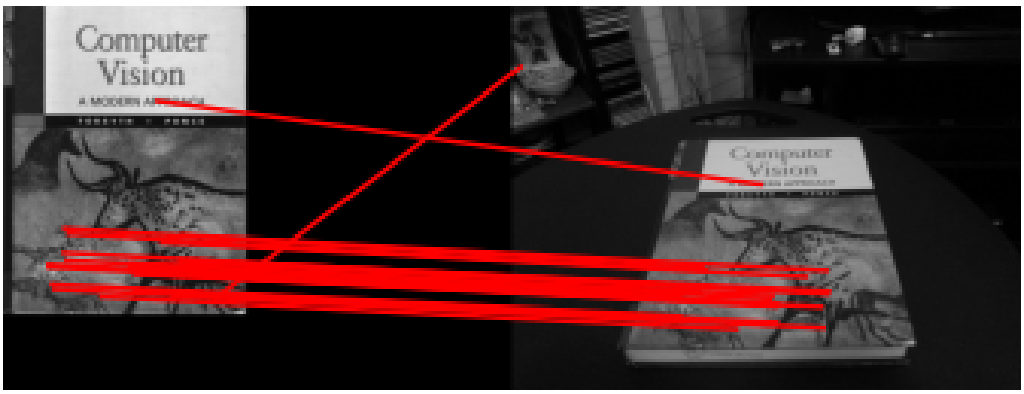
Figur 11: Visualizing Matched points. [Sigma = 0.15, Ratio = 0.7 (Default)]



Figur 12: Visualizing Matched points after increasing both [Sigma = 0.25, Ratio = 0.8)]



Figur 13: Visualizing Matched points. [Sigma = 0.10, Ratio = 0.6]



Reasoning-

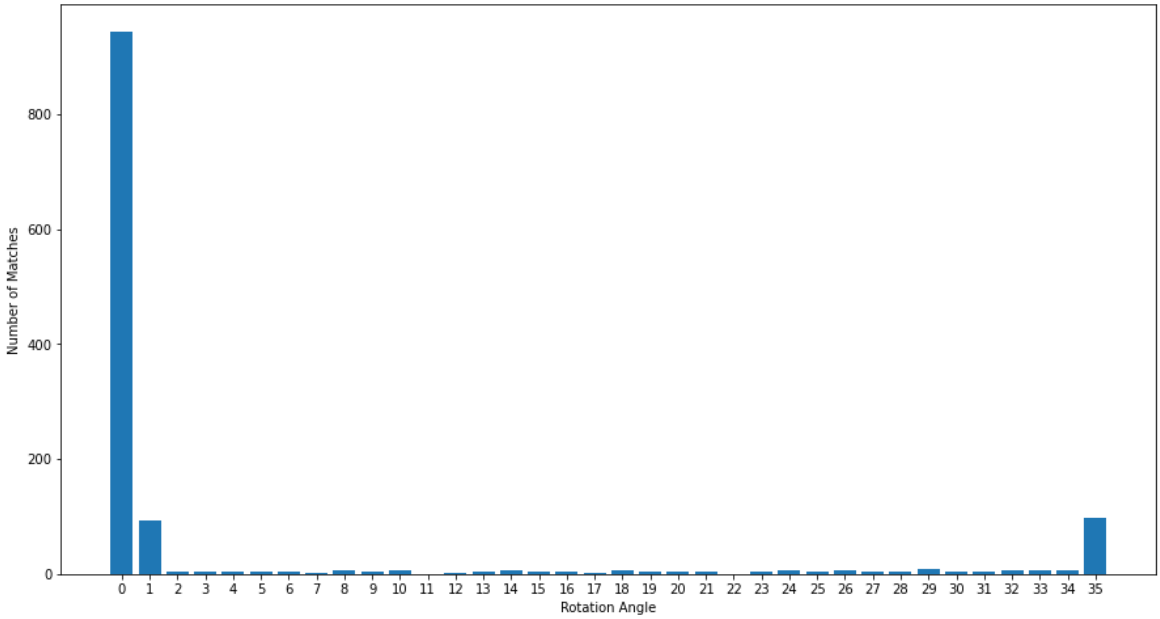
Ratio is maximum ratio of distances between first and second closest descriptor in the second set of descriptors (image 2). This threshold is useful to filter ambiguous matches between the two descriptor sets. As the max ratio is less, we want first closest descriptor match to be more closer compared to the second descriptor match so that we match with first closest descriptor with more confidence. Hence, the number of matches reduce on reducing Ratio. (direct relation)

sigma is the threshold used in deciding whether the pixels on the circle are brighter, darker or similar with respect to test pixel. Hence, decreasing threshold suggests lesser similar pixels, thereby increasing the count of corners detected. (inverse relation)

Question 2.1.6: BRIEF and Rotations

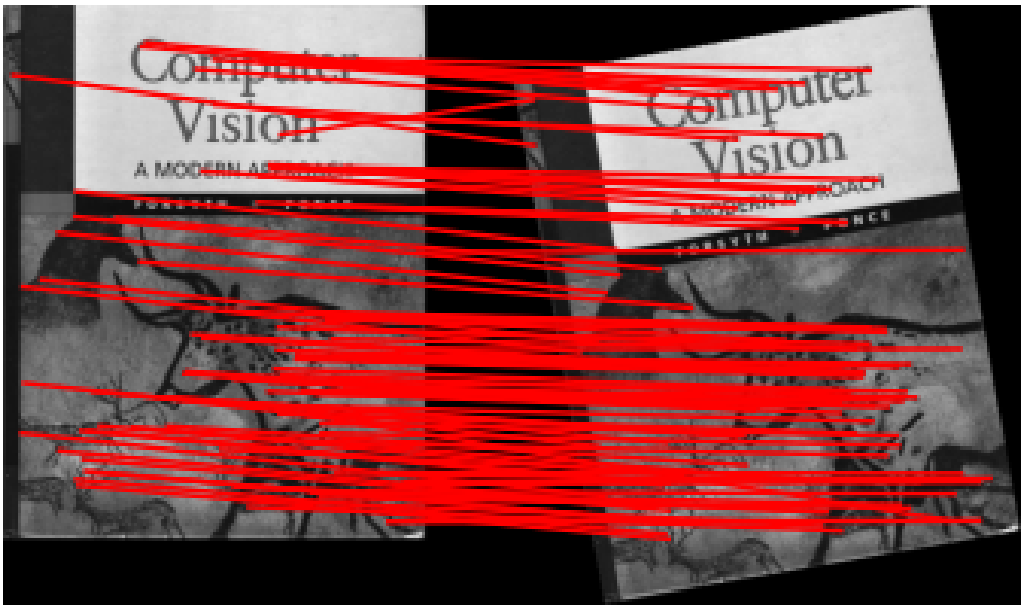
For default values in **opts.py**, i.e., sigma = 0.15 and ratio = 0.7 - the matches for the 36 rotation values are presented in a histogram below. (Sigma can be decreased and ratio can be increased to observe more matches)

Figur 14: Count of match points at different angles (angles = X-axis*10). (Histogram)

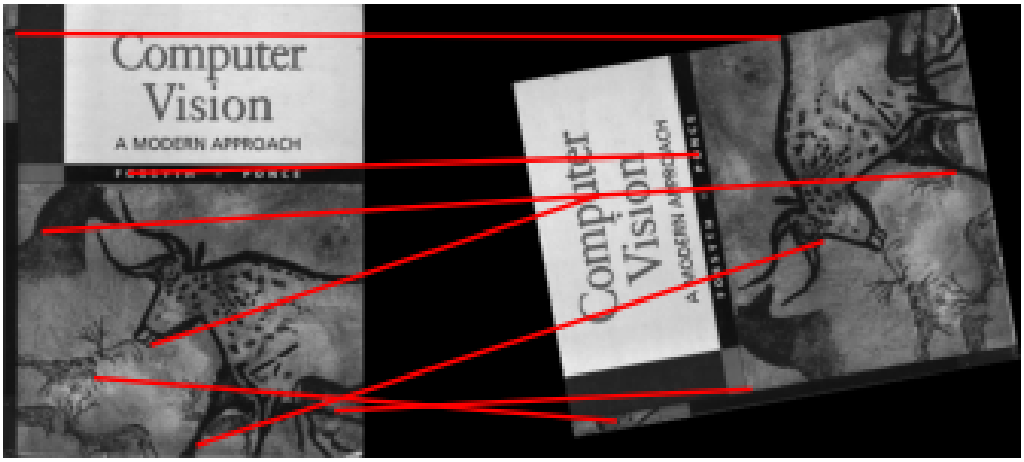


The visualization for 3 differently rotated orientations of the same image shown below. (Angles - 10, 100 and 200 degrees respectively).

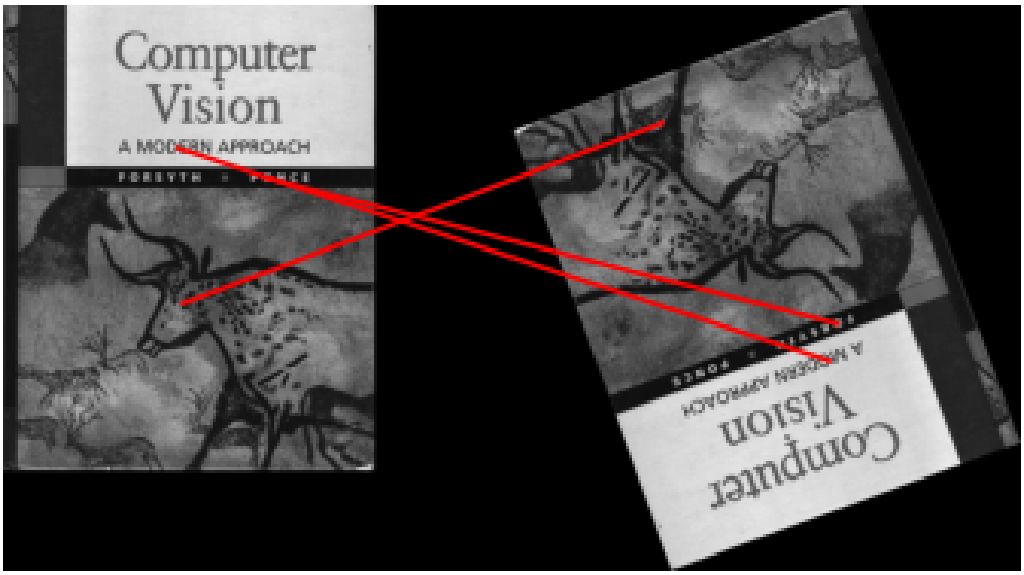
Figur 15: **Match points for Rotation Angle = 10**



Figur 16: **Match points for Rotation Angle = 100**

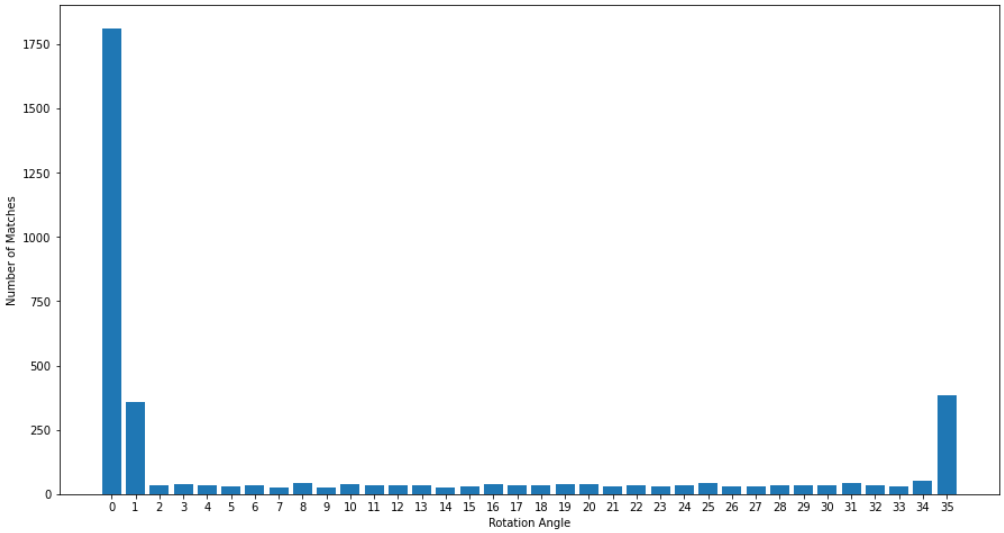


Figur 17: **Match points for Rotation Angle = 200**



Similar result is obtained irrespective of sigma and/or ratio changes (shown below).

Figur 18: Count of match points at different angles. Sigma reduced to 10 and Ratio increased to 0.8 - for more matches. (angles = X-axis*10)



This is bound to happen because BRIEF descriptor or the Binary Robust Independent Elementary Features method is **not** orientation or scale invariant. Thus, it works only for very small changes in orientation, but not for large values of rotation.

The code for calculating the above histogram - **briefRotTest.py** is shown below.

Figur 19: Code snippet of **briefRotTest.py**

```python
from opts import get_opts
import scipy
import matplotlib.pyplot as plt
from displayMatch import displayMatched

#Q2.1.6

def rotTest(opts):

    #Read the image and convert to grayscale, if necessary
    img = cv2.imread('../data/cv_cover.jpg')
    test_img = cv2.imread('../data/cv_cover.jpg')

    #if (len(img.shape)>=3):
    #    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    #if (len(test_img.shape)>=3):
    #    test_img = cv2.cvtColor(test_img, cv2.COLOR_BGR2GRAY)

    count=[]
    bins = np.arange(36)

    for i in range(36):
        print(i)
        rot_img = scipy.ndimage.rotate(test_img, angle=i*10)
        matches,_,_ = matchPics(img, rot_img, opts)
        if (i == 1 or i == 10 or i == 20):
            #displaying 2 images
            displayMatched(opts, img, rot_img)
        count.append(len(matches))

    #Display histogram
    fig, ax = plt.subplots(figsize=(15, 8))
    plt.bar(bins, count)
    ax.set_xticks(bins)
    ax.set_xticklabels(bins)
    plt.xlabel('Rotation Angle')
    plt.ylabel('Number of Matches')
    plt.show()
```

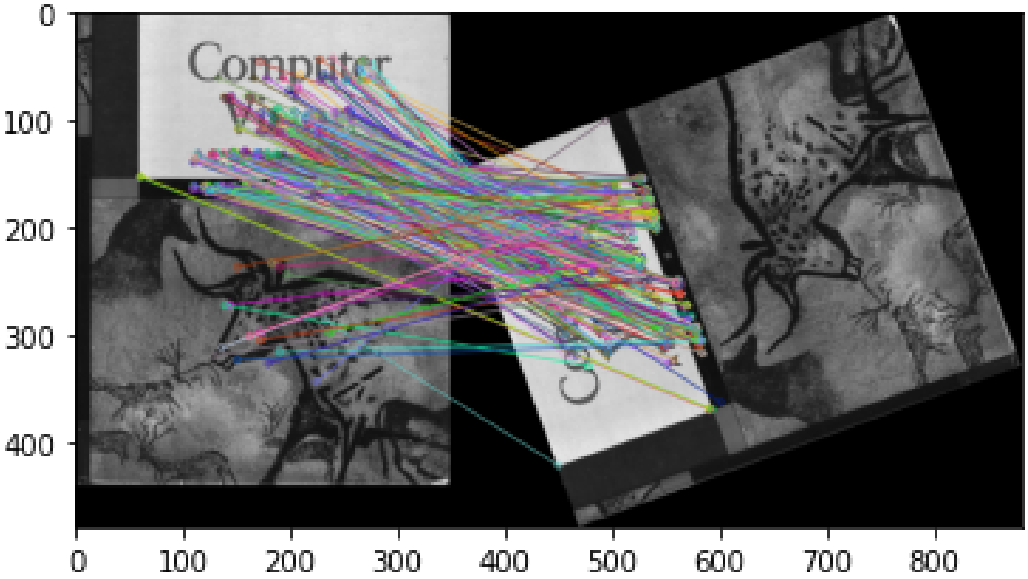
Q 2.1.7 : Improving Performance

2 - Rotation Invariance in BRIEF

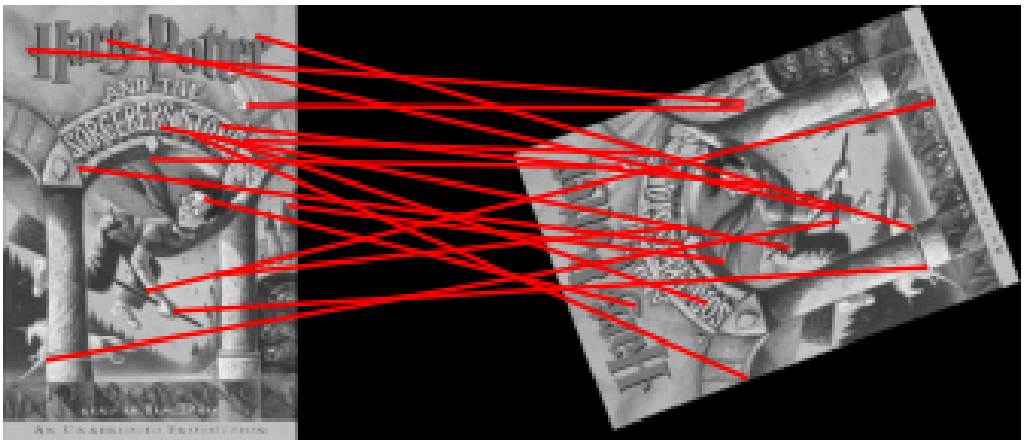
Rotating the images by big angles (=110 degrees here), we see how orb (Oriented FAST and Rotated BRIEF) work better than our Rotation Invariant versions coded.

Figur 20: **Matches for the rotated image by BRIEF**

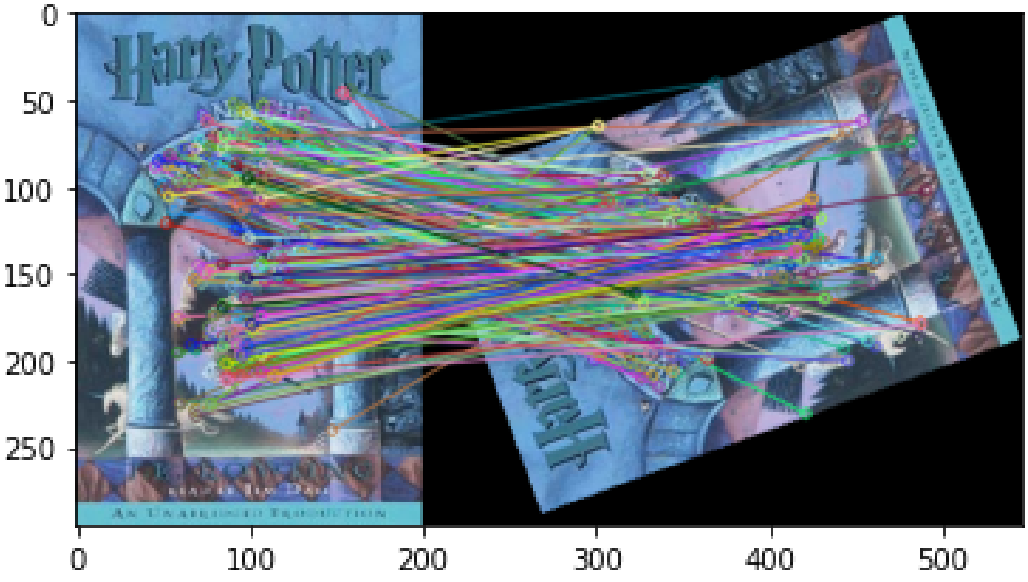
Figur 21: **Matches for the rotated image using ORB**



Figur 22: **Matches for the rotated image by BRIEF**



Figur 23: **Matches for the rotated image using ORB**



Code for implementing ORB:-

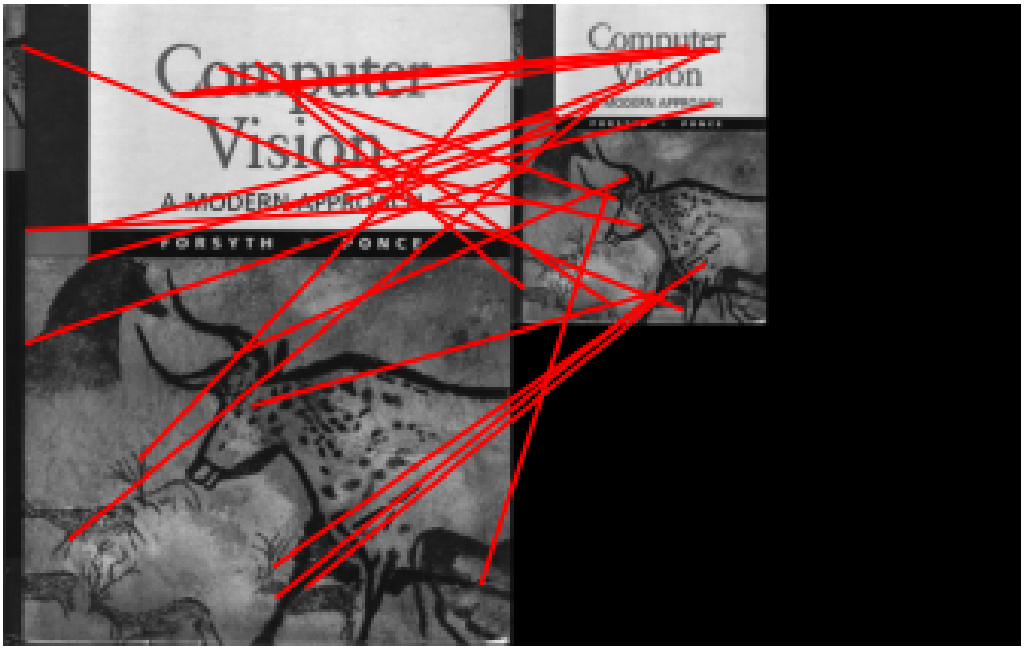
Figur 24: Code snippet of **test_orb.py**

```
import cv2
from opts import get_opts
import matplotlib.pyplot as plt
from displayMatch import displayMatched
import scipy

#Q2.1.7 - extra

def siftTest(opts):

    #Read the image and convert to grayscale, if necessary
    img = cv2.imread('../data/hp_cover.jpg')
    test_img = cv2.imread('../data/hp_cover.jpg')
    test_img = scipy.ndimage.rotate(test_img, angle=110)

    if (len(img.shape)>=3):
        img1 = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    if (len(test_img.shape)>=3):
        test_img1 = cv2.cvtColor(test_img, cv2.COLOR_BGR2GRAY)

    orb = cv2.ORB_create()
    kp1, desc1 = orb.detectAndCompute(img1,None)
    kp2, desc2 = orb.detectAndCompute(test_img1, None)

    # BFMatcher with default params
    bf = cv2.BFMatcher()

    matches = bf.match(desc1,desc2)
    img3 = cv2.drawMatches(img,kp1,test_img,kp2,matches,None,
                flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
    plt.imshow(img3),plt.show()

    displayMatched(opts, img, test_img)


if __name__ == "__main__":

    opts = get_opts()
    siftTest(opts)
```

2 - Scale Invariance in BRIEF

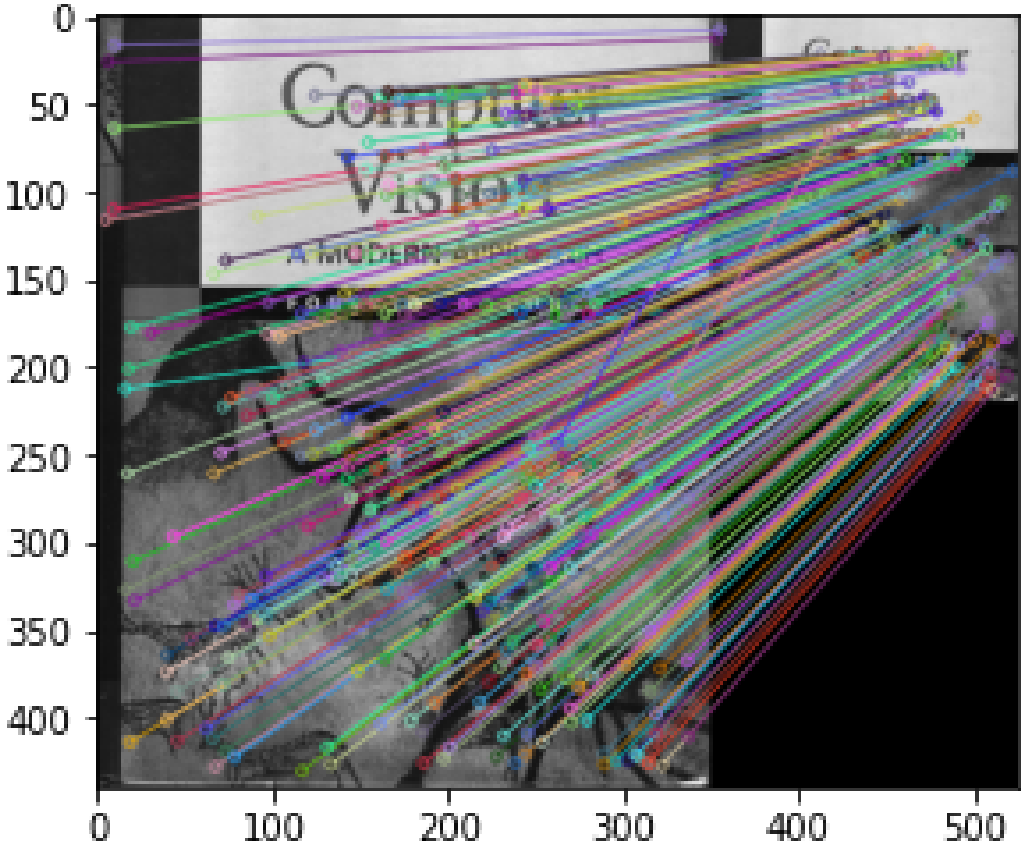
Taking the same picture at half the size and comparing gives us lesser and more incorrect matches. This is because of because we use FAST for corner detection along with BRIEF for descriptor.

Figur 25: **Matches for half the size**



The implementation in the paper suggests the use of different scales of gaussians on downsampled group of images (octaves). To represent LoG, DoG is used as it is similar but faster to compute. The maxima/minima of the intensity pixel is selected - this is done by comparing each with its 3D neighbours. Using the Taylor series, interpolation is done and keypoints are found for matching. The BREIF descriptors are found for all the points at different scales and sizes. This helps improve the number of matches as well as their correctness. The result is shown below : -

Figur 26: **Matches for half the size using the method proposed in the Paper**



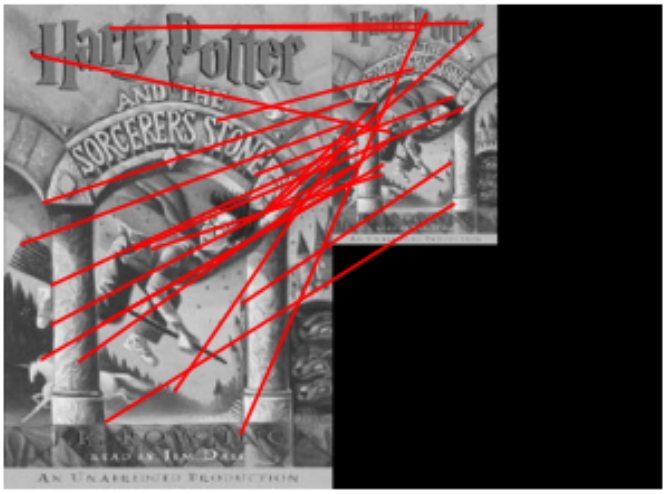
Code for the same:-

Figur 27: Code snippet of **test_sift.py**

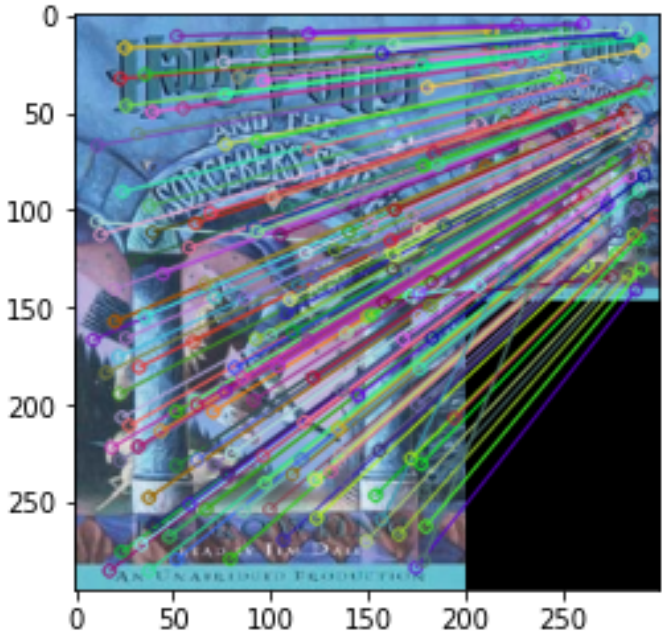
```python
from opts import get_opts
import matplotlib.pyplot as plt
from displayMatch import displayMatched

#Q2.1.7 - extra

def siftTest(opts):

    #Read the image and convert to grayscale, if necessary
    img = cv2.imread('../data/cv_cover.jpg')
    test_img = cv2.imread('../data/cv_cover.jpg')
    test_img = cv2.resize(test_img, (img.shape[1]//2,img.shape[0]//2))

    if (len(img.shape)>=3):
        img1 = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    if (len(test_img.shape)>=3):
        test_img1 = cv2.cvtColor(test_img, cv2.COLOR_BGR2GRAY)

    sift = cv2.SIFT_create()
    kp1, desc1 = sift.detectAndCompute(img1,None)
    kp2, desc2 = sift.detectAndCompute(test_img1, None)

    # BFMatcher with default params
    bf = cv2.BFMatcher()
    matches = bf.knnMatch(desc1,desc2,k=2)
    # Apply ratio test
    pts = []
    for m,n in matches:
        if m.distance < 0.75*n.distance:
            pts.append([m])

    img3 = cv2.drawMatchesKnn(img,kp1,test_img,kp2,pts,None,
                flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
    plt.imshow(img3),plt.show()

    displayMatched(opts, img, test_img)
```

Another Example for Harry Potter Book:

Figur 28: **Matches for half the size**



Figur 29: **Matches for half the size using the method proposed in the Paper**



Question 2.2: Homography Computation

Q 2.2.1 : Computing the Homography

Figur 30: Code snippet of **computeH in planarH.py**

```python
import numpy as np
import cv2
from displayMatch import displayMatched
from opts import get_opts

def computeH(x1, x2):
    #Q2.2.1
    #Compute the homography between two sets of points

    r,c = np.shape(x1)
    A = []
    for i in range(r):
        r1 = [-x2[i,0] , -x2[i,1] , -1, 0, 0, 0, x1[i,0]*x2[i,0] , x1[i,0]*x2[i,1] , x1[i,0]]
        r2 = [ 0, 0, 0, -x2[i,0] , -x2[i,1] , -1, x1[i,1]*x2[i,0], x1[i,1]*x2[i,1]  , x1[i,1]]
        A.append(r1)
        A.append(r2)

    A = np.asarray(A)
    u, s, vh = np.linalg.svd(A, full_matrices=True)

    # take last row of V.T because s val lowest there.
    ev = vh[-1,:]/vh[-1,-1]  #### Dont forget to scale idiot
    H2to1 = np.reshape(ev,(3,3))

    ###    Checking -
    ###    z = np.asarray([-2,1,1]).reshape(-1,1)
    ###    print(H2to1, H2to1@z)

    return H2to1
```

Q 2.2.2 : Homography Normalization

Figur 31: Code snippet of **computeH_norm in planarH.py**

```python
def computeH_norm(x1, x2):
    #Q2.2.2
    #Compute the centroid of the points
    r,c = np.shape(x1)

    #print(x1,x2)

    cx1 = np.mean(x1,axis=0) # 2 values - x_mean and y_mean
    cx2 = np.mean(x2,axis=0)

    #Shift the origin of the points to the centroid
    x1_dash = x1-cx1
    x2_dash = x2-cx2

    #Normalize the points so that the largest distance from the origin is equal to sqrt(2)
    dist_x1 = np.max(np.linalg.norm(x1_dash, axis = 1)) #to find max distance
    dist_x2 = np.max(np.linalg.norm(x2_dash, axis = 1))

    # Scaling Factor
    sf_x1 = np.sqrt(2)/dist_x1
    sf_x2 = np.sqrt(2)/dist_x2
    x1_dash = x1_dash*sf_x1
    x2_dash = x2_dash*sf_x2

    #Similarity transform 1
    T1 = np.asarray([[sf_x1, 0, -sf_x1*cx1[0]], [0, sf_x1, -sf_x1*cx1[1]], [0,0,1]])
    x1 = np.hstack((x1,np.ones(r).reshape(-1,1)))

    #Similarity transform 2
    T2 = np.asarray([[sf_x2, 0, -sf_x2*cx2[0]], [0, sf_x2, -sf_x2*cx2[1]], [0,0,1]])
    x2 = np.hstack((x2,np.ones(r).reshape(-1,1)))

    #Compute homography
    H2to1 = computeH(x1_dash,x2_dash)
    H2to1 = np.linalg.inv(T1) @ H2to1 @ T2

    ### Checking
    ### print(H2to1)
    ### z = np.asarray([-1,1,1]).reshape(-1,1)
    ### print(H2to1@z)

    return H2to1
```

Q 2.2.3 : Implement RANSAC

Figur 32: Code snippet of **computeH_ransac in planarH.py**

```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

    inliers = []
    in_co = 0
    bestH2to1 = None
    for i in range(max_iters): # max_iters
        idx = np.arange(0,len(locs1))
        np.random.shuffle(idx)
        ptsel = idx[0:4]
        H2to1 = computeH_norm(locs1[ptsel], locs2[ptsel])
        locs2_dash = np.hstack((locs2,np.ones(len(locs2)).reshape(-1,1))) #homogenous form
        locs1_est = (H2to1@locs2_dash.T)[0:2,:] #column vector of estimate found
        dist = np.linalg.norm(locs1.T-locs1_est, axis = 0)
        inlier = (dist<inlier_tol).astype(int)
        inlier_count = np.sum(inlier) #total number of inlier
        if inlier_count>in_co :
            inliers = inlier
            bestH2to1 = H2to1
            in_co = inlier_count

    if len(inliers) > 1 :
        index_good_pts = [i for i,j in enumerate(inliers) if j==1]
        if (len(index_good_pts)>=4): # recalculate for better values as suggested in class
            bestH2to1 = computeH_norm(locs1[index_good_pts], locs2[index_good_pts])
    else:
        bestH2to1 = np.eye(3)

    return bestH2to1, inliers
```

Q 2.2.4: Automated Homography Estimation and Warping

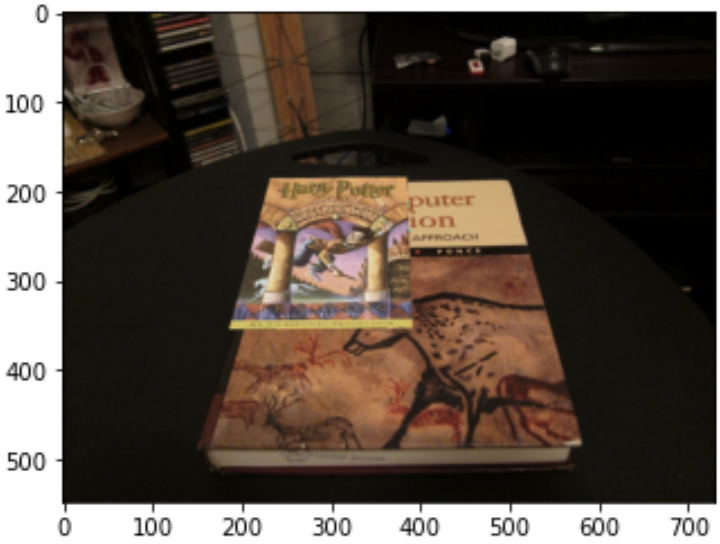
Figur 33: Code snippet of **Part 1 and 2: HarryPotterize.py**

```
def warpImage(opts):
    cv_cover = cv2.imread('../data/cv_cover.jpg')
    cv_desk = cv2.imread('../data/cv_desk.png')
    hp_cover = cv2.imread('../data/hp_cover.jpg')
    ### print(cv_cover.shape,cv_desk.shape,hp_cover.shape)

    matches, locs1, locs2 = matchPics(cv_desk, cv_cover, opts)
    #plotMatches(cv_desk, cv_cover, matches, locs1, locs2) ### already prints only true matches

    locs1 = locs1[matches[:,0]]
    locs1 = locs1[:,[1,0]]
    locs2 = locs2[matches[:,1]]
    locs2 = locs2[:,[1,0]]
    bestH2to1, inliers = computeH_ransac(locs1, locs2, opts) #tell us how to go from locs2 to locs1
    ### H2to1 goes from cover to desk  #plt.imshow(image2)

    hp_cover = cv2.resize(hp_cover, (cv_cover.shape[1], cv_cover.shape[0]))
    comp_img = compositeH(bestH2to1, hp_cover, cv_desk)
    comp_img = cv2.cvtColor(comp_img, cv2.COLOR_BGR2RGB)

    plt.imshow(comp_img)
```

Figur 34: Code snippet of **Part 3 and 5: compositeH from Planar.py**

```
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_template = np.ones((template.shape[0], template.shape[1]), dtype=np.uint8)
    mask_img = np.zeros((img.shape[0], img.shape[1]), dtype=np.uint8)

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

    mask_temp_warp = cv2.warpPerspective(mask_template, H2to1, (mask_img.shape[1], mask_img.shape[0]))
    #print(mask_temp_warp.shape) ### 480 640
    #print(mask_temp_warp) ##white box on black (0)

    warp_not = cv2.bitwise_not(mask_temp_warp)//255
    warp_not = np.stack([warp_not,warp_not,warp_not])
    warp_not = np.transpose(warp_not, (1,2,0))

    background = img*warp_not
    composite_img = background + composite_img

    return composite_img
```

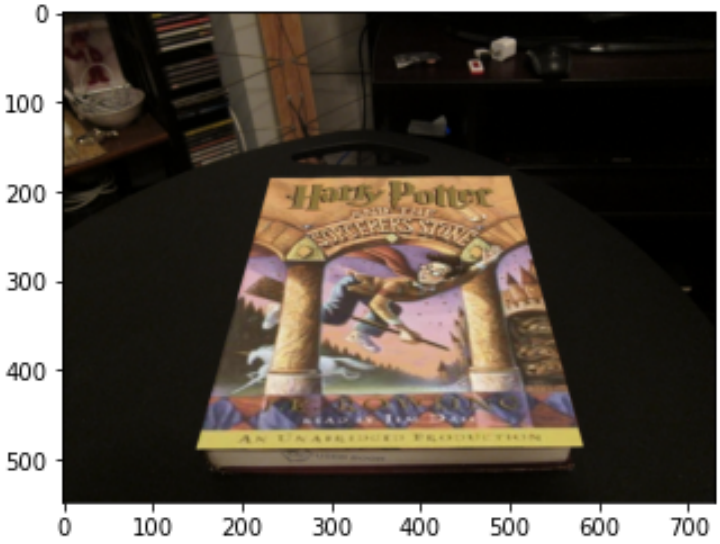
Figur 35: **Part 4: Correct Warping but different Size**



The warping is not filling the entire space of the cover on the desk. This is because the images **hp_cover** and **cv_cover** are of different sizes.

Easy fix to this would be resize the **hp_cover** image to the dimensions of **cv_cover** image. Results can be seen below.

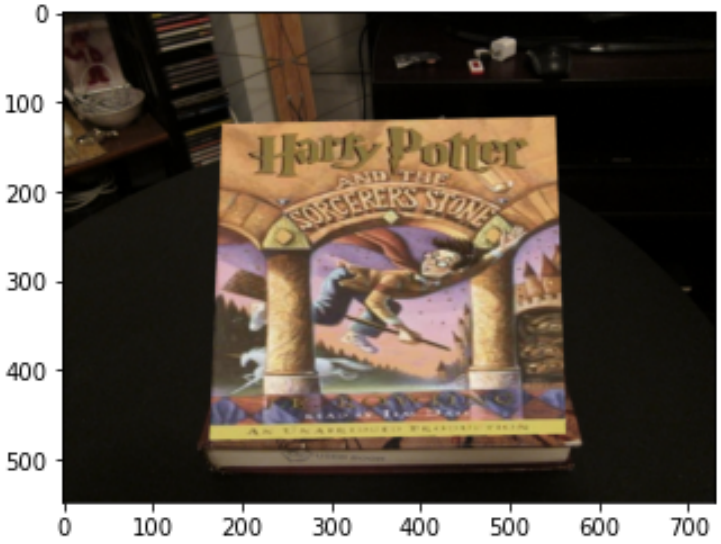
Figur 36: **Part 4 and 6: Correct Warping with modified hp_cover**



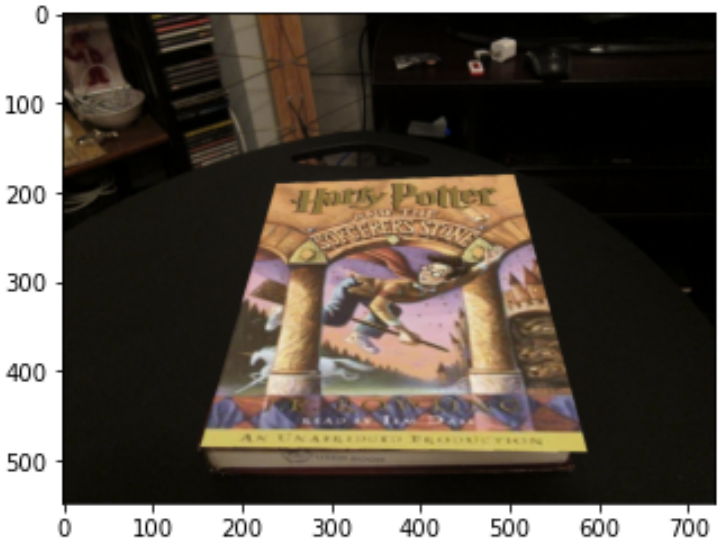
Q 2.2.5: RANSAC Parameter Tuning

1. Keeping **max_iters** constant at 500, we change **inlier_tol** - 0.5, 1, 2 (default), 15.

Figur 37: **max_iters** = 500, **inlier_tol** = 0.5

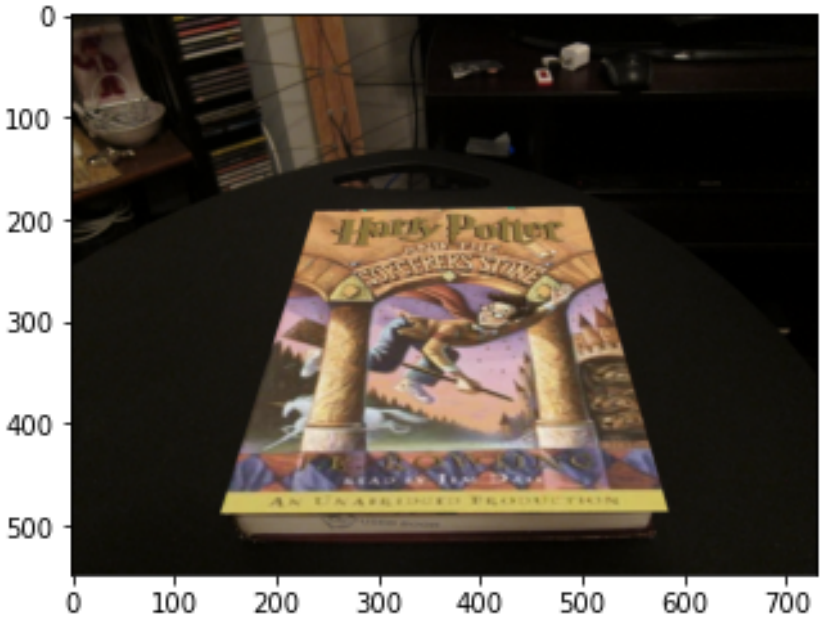


Figur 38: **max_iters** = 500, **inlier_tol** = 1.0

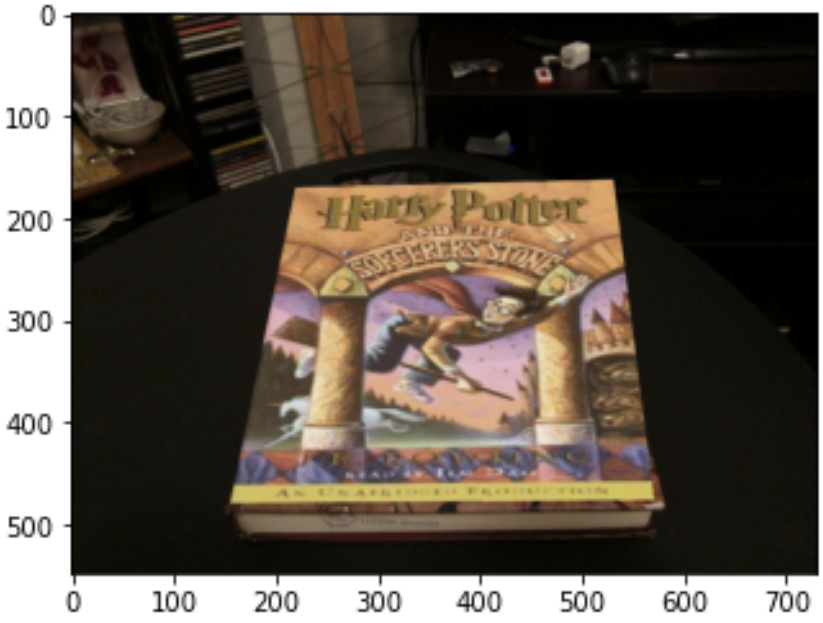


For very low value of tolerances - I wasn't able to find Homography. Even if I did (tol = 0.5),it was bad. As tolerance increased the fit seemed to get better (0.5 to 2). However, as seen in tolerances = 15, the output quality again deteriorates. This is probably because we have to keep the tolerance at an optimum - such that it is able to find the correct Homography as well as ensure noisy point-pairs do not get included.

Figur 39: **max_iters** = 500, **inlier_tol** = 2.0

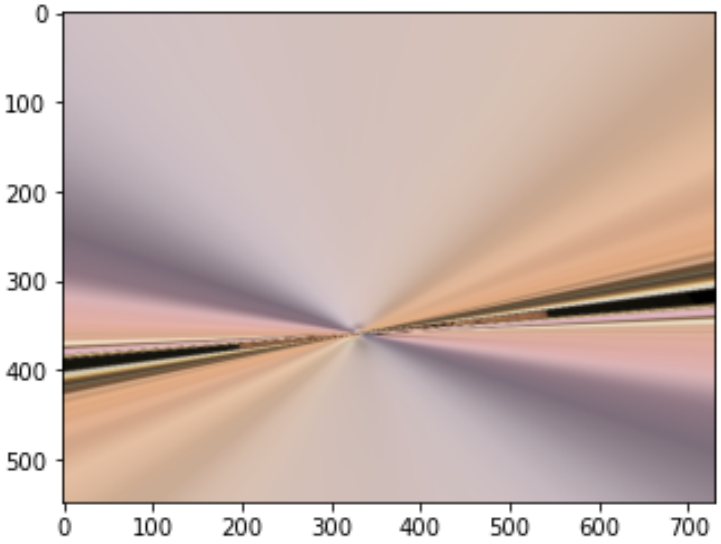


Figur 40: **max_iters** = 500, **inlier_tol** = 15.0 (Raised cover height on top right)

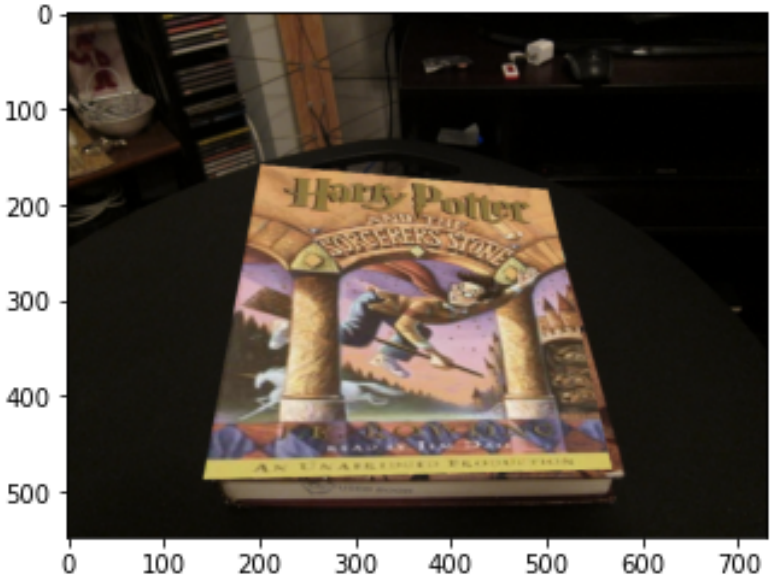


2. Keeping **inlier_tol** constant at 1.0, we change **max_iters** - 30, 250, 500(seen above) and 1000.

Figur 41: **max_iters** = 30, **inlier_tol** = 2.0

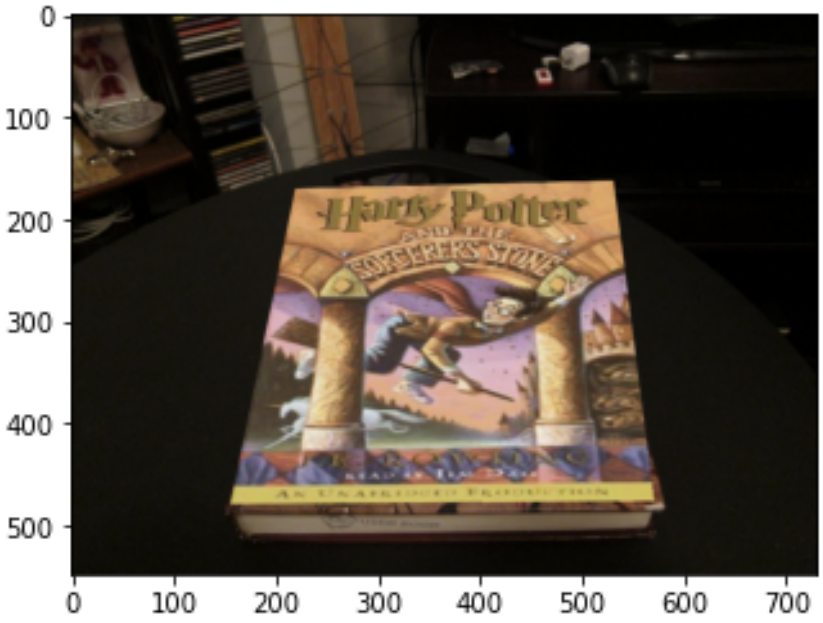


Figur 42: **max_iters** = 250, **inlier_tol** = 2.0

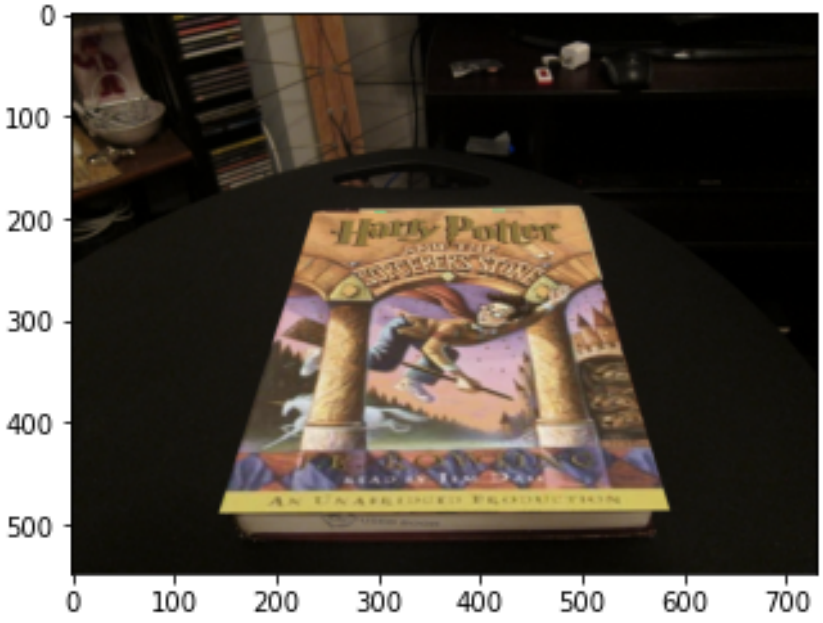


With more iterations for the same tolerance it is observed that the cover has a higher chance of warping correctly. For very low iterations, results can be very bad (see iter - 30). Iterations increases the number of times the pairs of matching points checked for valid homography, thereby increasing the probability to find a correct solution. Iterations take time and hence should to optimised as per the probability of correct value is required.

Figur 43: **max_iters** = 1000, **inlier_tol** = 2.0



Figur 44: **max_iters** = 1000, **inlier_tol** = 5.0 (Best Results)



Question 3: Creating your AR Application

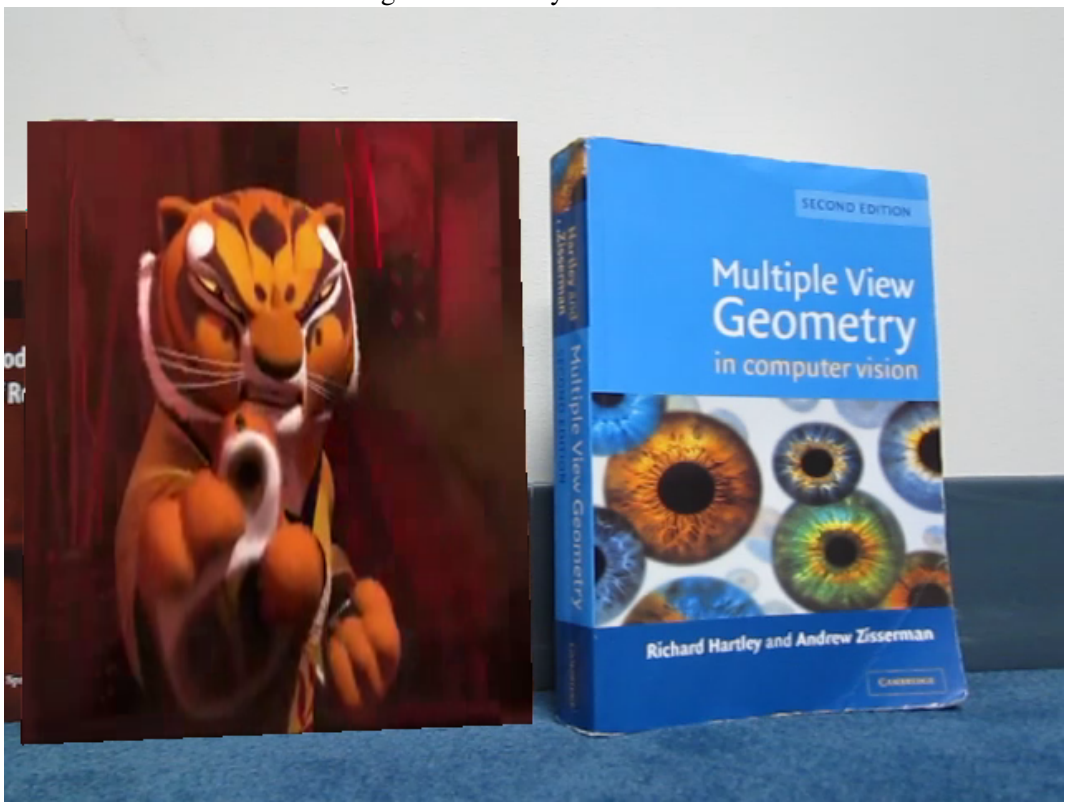
Question 3.1: Incorporating Video

Figur 45: Code snippet of **ar.py**

```
#Write script for Q3.1

book = loadVid("../data/book.mov")
shape_book = np.shape(book)

cv_cover = cv2.imread('../data/cv_cover.jpg')
shape_cover = cv_cover.shape

ars = loadVid("../data/ar_source.mov")
ars = ars[:,45:315,:,:] ## Removes the black strips from top and bottom
shape_ars = np.shape(ars)
### the range of 40 to 320 is found by the for loop below
###for i in range(shape_ars[1]):
    ###print(i, ars[1,i,100,1])

    # 0.8 15 1000 2
centerx = shape_ars[2]//2
x = centerx - shape_cover[1]/2

ars = ars[:,:, int(x):int(x+shape_cover[1]),:] ### central region extracted from ars
shape_ars = np.shape(ars)

f = min(shape_ars[0],shape_book[0])
frame_size = (shape_book[2],shape_book[1])

out = cv2.VideoWriter('../result/ar.avi',cv2.VideoWriter_fourcc('M','J','P','G'), 30, frame_size)

for i in range(f):
    print(i)
    ars_frame = cv2.resize(ars[i,:,:,:], (cv_cover.shape[1], cv_cover.shape[0]))
    ars_frame_gray = cv2.cvtColor(ars_frame,cv2.COLOR_BGR2GRAY)
    book_frame = book[i,:,:,:]
    book_frame_gray = cv2.cvtColor(book_frame,cv2.COLOR_BGR2GRAY)
    cv_cover_gray = cv2.cvtColor(cv_cover,cv2.COLOR_BGR2GRAY)

    matches, locs1, locs2 = matchPics(book_frame, cv_cover, opts)
    #plotMatches(cv_desk, cv_cover, matches, locs1, locs2) ### already prints only true matches

    locs1 = locs1[matches[:,0]]
    locs1 = locs1[:,[1,0]]
    locs2 = locs2[matches[:,1]]
    locs2 = locs2[:,[1,0]]
    bestH2to1, inliers = computeH_ransac(locs1, locs2, opts) #tell us how to go from locs2 to locs1

    comp_img = compositeH(bestH2to1, ars_frame, book_frame)

    dst = cv2.cvtColor(comp_img, cv2.COLOR_BGR2RGB)
    #plt.imshow(dst)

    #path = '../images/'+str(i)+'.png'
    #plt.imsave(path, dst)
    out.write(dst[:,:,::-1])

out.release()
```

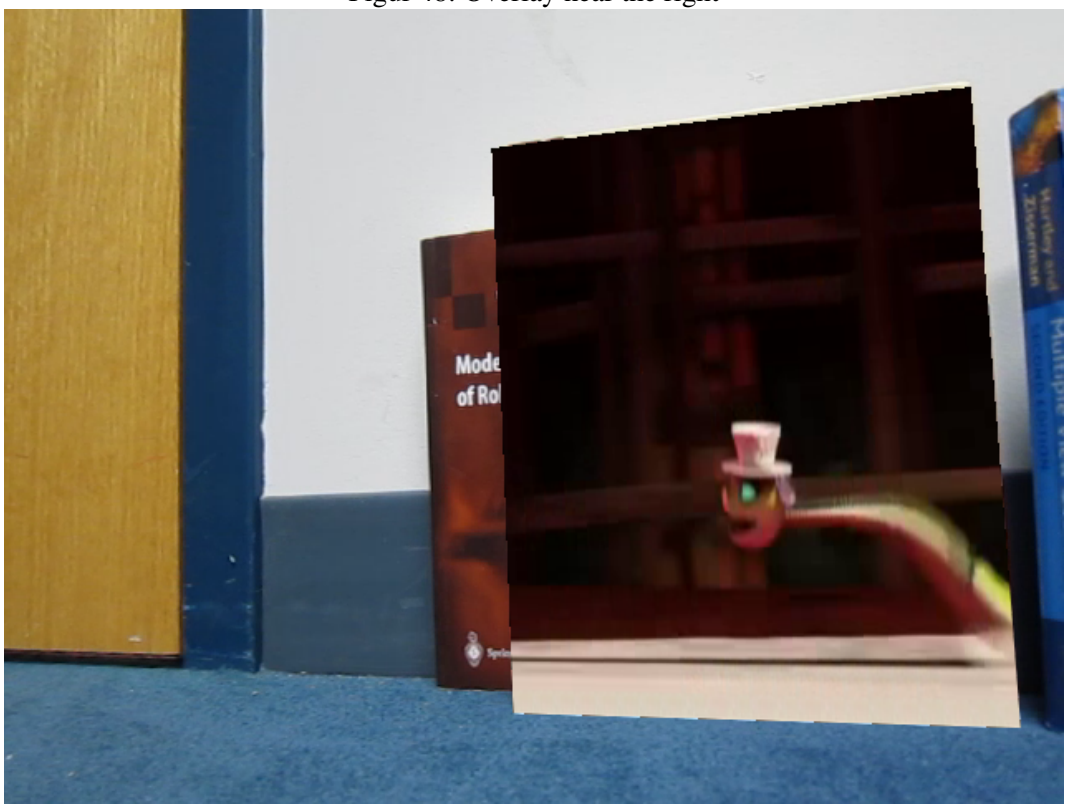
The video **ar.avi** has been added to the file submitted. The 3 Images for **left**, **right** and **center** are attached below. -

Figur 46: Overlay near the left



Figur 47: Overlay near the center

Figur 48: Overlay near the right



Question 3.2: AR Real Time

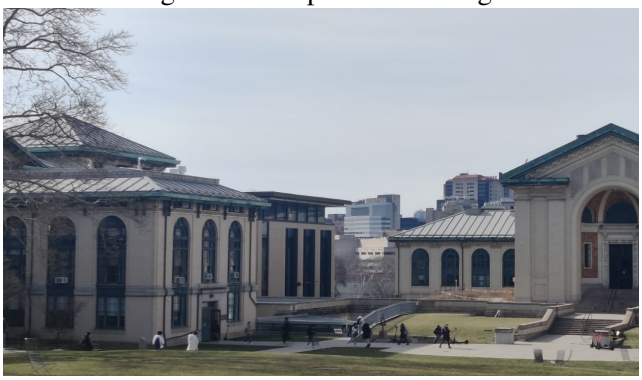
Question 4: Panorama

Figur 49: Code snippet of **panorama.py**

```python
import numpy as np
import cv2
import matplotlib.pyplot as plt
from planarH import computeH_ransac
from planarH import compositeH
from opts import get_opts
from matchPics import matchPics
from helper import plotMatches

# Import necessary functions

# Q4
opts = get_opts()

pl = cv2.imread('../data/pano_left_cmu.jpeg')
pr = cv2.imread('../data/pano_right_cmu.jpeg')

matches, locs1, locs2 = matchPics(pl, pr, opts)
locs1 = locs1[matches[:,0]]
locs1 = locs1[:,[1,0]]
locs2 = locs2[matches[:,1]]
locs2 = locs2[:,[1,0]]

bestH2to1, inliers = computeH_ransac(locs1, locs2, opts)

r,c = int(2*pl.shape[1]),int(1.5*pl.shape[0])

dst1 = cv2.warpPerspective(pr, bestH2to1, (r,c))
dst1[0:pl.shape[0],0:pl.shape[1]]=pl
cv2.imwrite('../data/output_cmu.jpg',dst1)
plt.imshow(dst1[:,:,::-1])
```

My Images are as follows:-

Figur 50: Left part of the image



Figur 51: Right part of the image

Figur 52: My Panorama created

By using the warping and coordinates of the right image, we can find the exact dimensions for the image to remove the black portion.

Figur 53: Left and Right part of the image

Figur 54: Panorama created