Q1.1

Let X_{pi} be the real world point projected through P_1 and P_2

$X_{pi} = P_1X_1$

$X_{pi} = P_2X_2$

$P_1X_1 = P_2X_2$

Since P_1 and P_2 are 3×4, multiply P_1^T for inversion

$(P_1^TP_1)\, X_1 = (P_1^TP_2)\, X_2$

$X_1 = (P_1^TP_1)^{-1}(P_1^TP_2)\, X_2$

$X_1 = H_1X_2$

This proves there exists a homography H satisfying equation 1

Please note that H is a 3×3 matrix

Q1.2

1. Though h has 9 different elements, there are only 8 degrees of freedom because one element is used in normalization.
2. Since we have 8 different elements to be solved for, we need 4 point pairs.
3. From equation, we have

   $X_1 = H X_2$

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

   $X_1 = h_{11}x_2 + h_{12}y_2 + h_{13}$
   $Y_1 = h_{21}x_2 + h_{22}y_2 + h_{23}$
   $1 = h_{31}x_2 + h_{32}y_2 + h_{33}$

   $X_1 = (h_{11}x_2 + h_{12}y_2 + h_{13})/ (h_{31}x_2 + h_{32}y_2 + h_{33})$
   $Y_1 = (h_{21}x_2 + h_{22}y_2 + h_{23})/ (h_{31}x_2 + h_{32}y_2 + h_{33})$

   $h_{11}x_2 + h_{12}y_2 + h_{13} - (h_{31}x_2 + h_{32}y_2 + h_{33}) X_1 = 0$
   $h_{21}x_2 + h_{22}y_2 + h_{23} - (h_{31}x_2 + h_{32}y_2 + h_{33}) Y_1 = 0$

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

   $$A_i = \begin{bmatrix} x_2 & y_2 & 1 & 0 & 0 & 0 & -x_1x_2 & -x_1y_2 & -x_1 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -y_1x_2 & -y_1y_2 & -y_1 \end{bmatrix}$$

4. A is full rank if only trivial solution (h=0) exists, resulting in non-zero eigen values.

   When a non-trivial solution exists, A won't be full rank.
   Because of this, there would be at least one eigen value that is zero.

Q1.4.1

We know that,

$X_1 = K_1 [I\ 0] X$

$X_2 = K_2 [R\ 0] X$

$X_1 K_1^{-1} = [I\ 0] X$

$X_2 K_2^{-1} = R [I\ 0] X$

$X_2 K_2^{-1} = R X_1 K_1^{-1}$

$X_1 = (K_2^{-1} R^{-1} K_1) X_2$

$X_1 = H X_2$

This proves there exists a homography H

Q1.4.2

Let X_1 be the point on camera's plane at position 1.

Let X_2 be the point on camera's plane at position 2, after rotation of theta.

Let X_3 be the point on camera's plane at position 3, after rotation of theta.

The angle of rotation between the planes at position 1 and 3 is 2*theta.

$X_1 = H X_2$ -> Corresponds to rotation of theta -> Eq 1

$X_2 = H X_3$ -> Corresponds to rotation of theta -> Eq 2

H remains same as the camera intrinsic parameters are constant and its value is:

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

Subs Eq 2 in Eq 1, we get: $X_1 = H^2 X_3$

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} X \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} cos^2(\theta) - sin^2(\theta) & -2\sin(\theta)\cos(\theta) & 0 \\ 2\sin(\theta)\cos(\theta) & cos^2(\theta) - sin^2(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos(2\theta) & -\sin(2\theta) & 0 \\ \sin(2\theta) & \cos(2\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The general form is:

$X_1 = H_2 X_3$ -> Corresponds to rotation of 2*theta

By comparing, we get:

$H_2 = H^2$

This proves that H^2 corresponds to rotation of 2*theta.

Q1.4.3

Any arbitrary scene has depth. A planar homography does not take this into account. So, it fails mapping between different depth planes.

Q.1.4.4

Let **A** be a point in 3D space and the corresponding 2D point be **a**.

Let **P** be the projection matrix. The following equation projects 3D onto 2D

a = P A + a'

Solving for A we get,

A = P^{-1}(a – a') -> where direction of ray is **P^{-1} a** and the center is **-P^{-1}a**

This implies that P preserves lines algebraically.

Q2.1.1

FAST	**Harris Corner**
Pixelwise comparison	Patch comparison
Existence of sufficient number of continuous bright/dark pixels concludes a corner	Understanding ratio of lambda obtained through solving cornerness equation concludes a corner
Computationally less expensive but less robust to lighting and noise variations	Computationally more expensive as a lot of calculation is involved compared to FAST

Q2.1.2

BRIEF	**Filter Banks**
Binary strings produced through random pixel pair intensity comparison	Modern filter banks obtained by convolving image and filters
Fast	Computationally expensive
Capture local information	Capture global information

Filter banks can be used as descriptors but would have global info and less localised than BRIEF.

It takes more computation and storage as well. So, basically depends on the application.

Q2.1.3

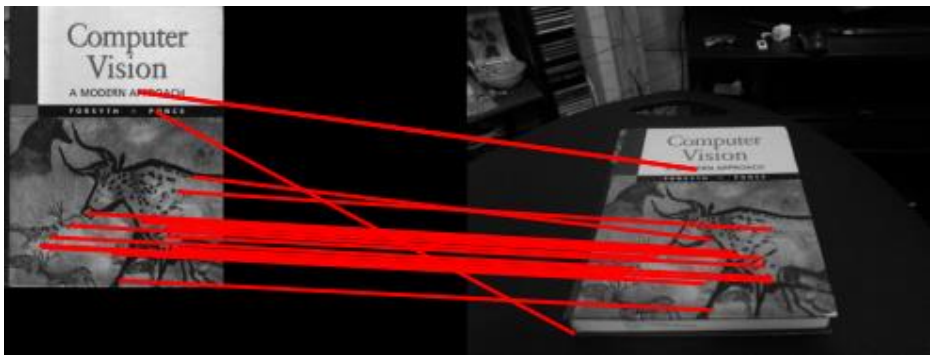
Hamming distance tells how many bits are different between the two binary vectors.

BRIEF represents image patch as a binary string. The value obtained through usage of Hamming tells us how close they are. The smaller the H distance, the more similar are the descriptors.

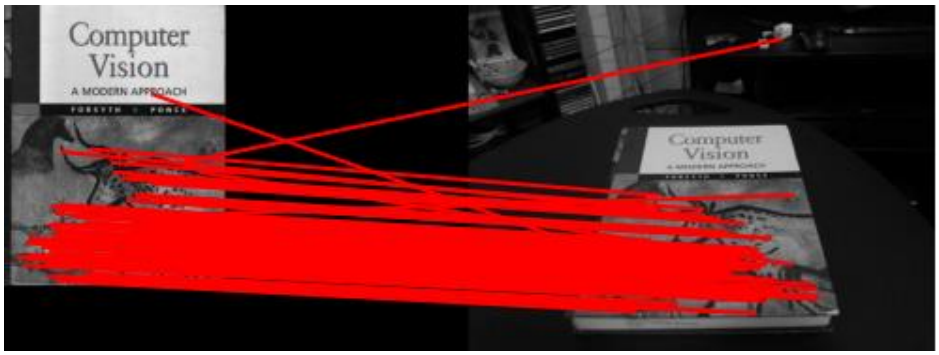
Calculating Hamming is faster than Euclidean. Small changes in illumination will flip a few bits which does not affect Hamming's distance that much in comparison to Euclidean, whose distance could vary a lot.

Q2.1.4

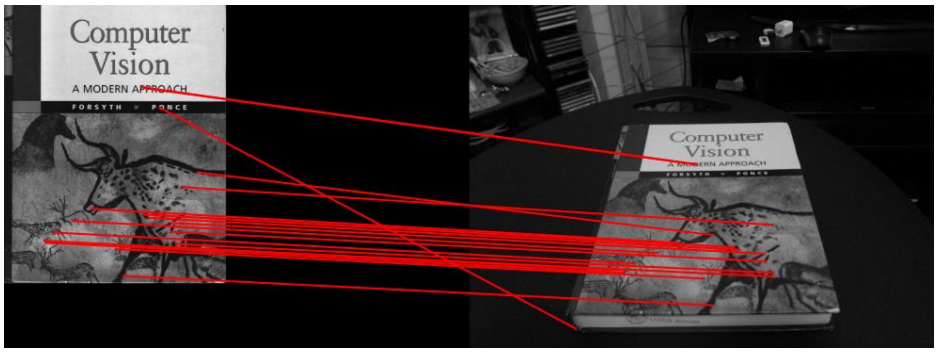
```
import numpy as np

import cv2
from helper import briefMatch
from helper import computeBrief
from helper import corner_detection

# Q2.1.4

def matchPics(I1, I2, opts):

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

        # TODO: Convert Images to GrayScale
        I1G = cv2.cvtColor(I1, cv2.COLOR_BGR2GRAY)
        I2G = cv2.cvtColor(I2, cv2.COLOR_BGR2GRAY)

        # TODO: Detect Features in Both Images
        fea1 = corner_detection(I1G, sigma)
        fea2 = corner_detection(I2G, sigma)

        # TODO: Obtain descriptors for the computed feature locations
        desc1, locs1 = computeBrief(I1G, fea1)
        desc2, locs2 = computeBrief(I2G, fea2)

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

        # Fliping y and x to x and y
        locs1 = np.fliplr(locs1)
        locs2 = np.fliplr(locs2)

        return matches, locs1, locs2
```



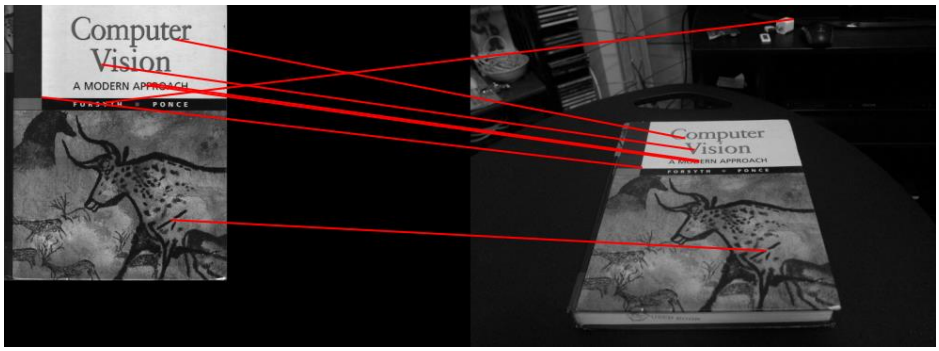
Q2.1.5



Sigma: 0.05 Ratio: 0.7

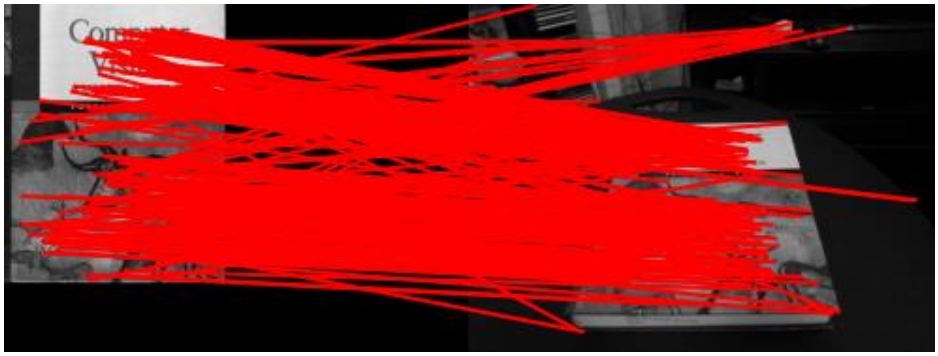


Sigma: 0.15 Ratio: 0.7

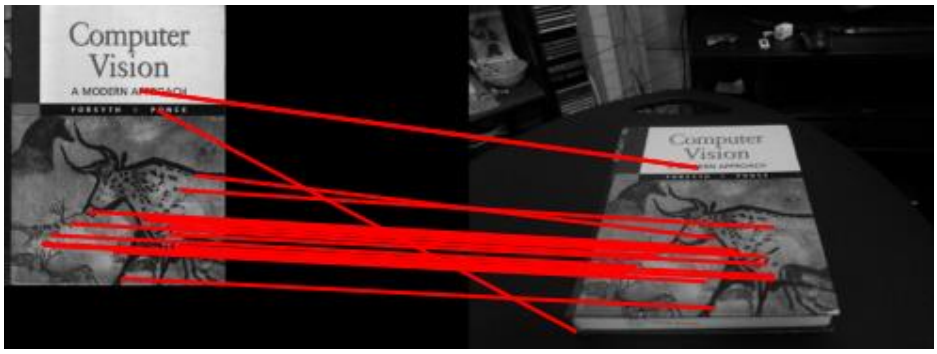


Sigma: 0.25 Ratio: 0.7

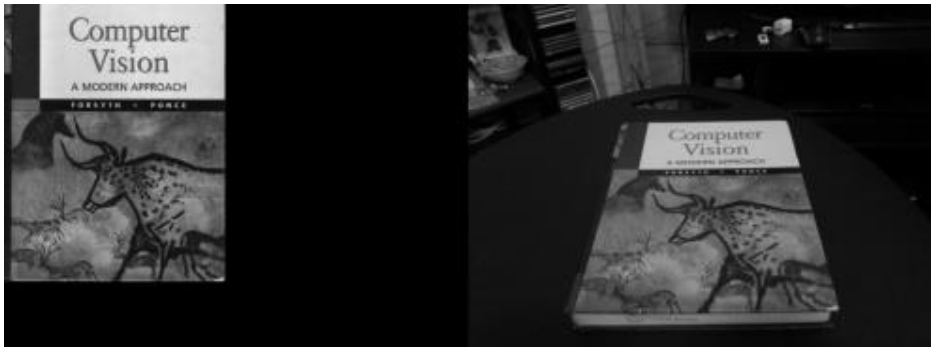
As sigma increases, the threshold increases resulting in fewer corners detected because of fewer continuous bright/dark pixels. An inverse relation exists between sigma and features matched.



Sigma: 0.15 Ratio: 1.4



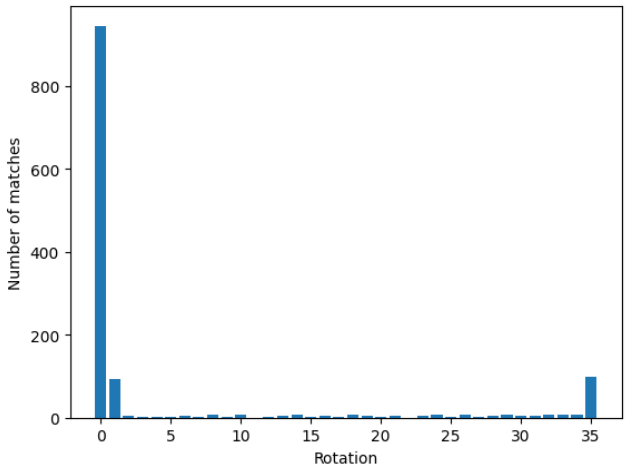
Sigma: 0.15 Ratio: 0.7



Sigma: 0.15 Ratio: 0.1

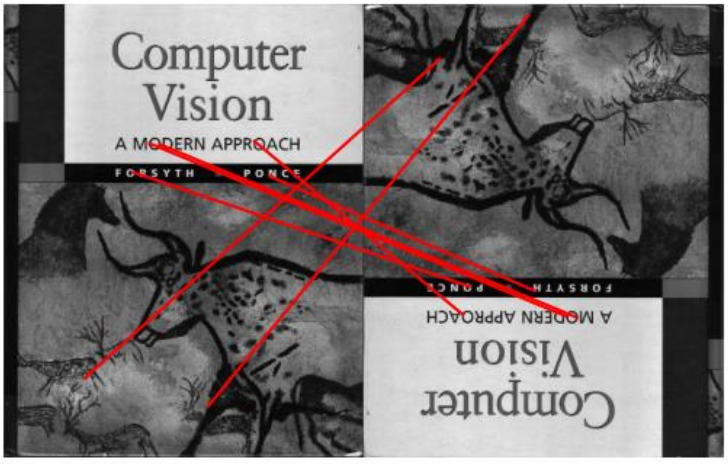
As ratio decreases, distance ratio decreases resulting in closeness of descriptors. Hence, less red lines. A direct relation exists between ratio and features matched.

Q2.1.6



Histogram

Rotation of 100 °



Rotation of 180 °

Rotation of 350 °

BRIEF computes random pixel pair's intensity values. When rotated, the descriptor value changes resulting in fewer matches.

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

#Q2.1.6

def rotTest(opts):

    # TODO: Read the image and convert to grayscale, if necessary
    I1G = cv2.imread('data/cv_cover.jpg')
    # I1G = cv2.cvtColor(I1G, cv2.COLOR_BGR2GRAY)

    x_axis_values = []
    y_axis_values = []

    for i in range(36):

        # TODO: Rotate Image
        R1 = scipy.ndimage.rotate(I1G, 10*i, axes=(1,0))

        # TODO: Compute features, descriptors and Match features
        matches, locs1, locs2 = matchPics(I1G, R1, opts)
        # plotMatches(I1G, R1, matches, locs1, locs2)

        # TODO: Update histogram
        x_axis_values.append(i)
        y_axis_values.append(matches.shape[0])

    pass

    # TODO: Display histogram
    plt.xlabel('Rotation')
    plt.ylabel('Number of matches')
    plt.bar(x_axis_values,y_axis_values)

if __name__ == "__main__":

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

Q2.2.1

```python
import numpy as np
import cv2
import random

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

    # Extract the x and y coordinates from the first set of points
    u1 = x1[:,0]
    v1 = x1[:,1]

    # Extract the x and y coordinates from the second set of points
    u2 = x2[:,0]
    v2 = x2[:,1]

    # Get the number of points
    index = x1.shape[0]

    for i in range(index):
        # Construct matrix A based on the homography equation
        A.append([u2[i], v2[i], 1, 0, 0, 0, -u1[i]*u2[i] , -u1[i]*v2[i], -u1[i]])
        A.append([0, 0, 0, u2[i], v2[i], 1, -v1[i]*u2[i], -v1[i]*v2[i], -v1[i]])

    # Convert list to array
    A = np.vstack(A)

    # Perform SVD
    U, S, V = np.linalg.svd(A)

    # Last column of V gives the solution
    eigen_vector = V.T[:,-1]

    H2to1 = eigen_vector.reshape(3,3)

    return H2to1
```

Q2.2.2

```
def computeH_norm(x1, x2):
    # TODO: Compute the centroid of the points
    u1 = np.array(x1[:,0])
    v1 = np.array(x1[:,1])

    u2 = np.array(x2[:,0])
    v2 = np.array(x2[:,1])

    u1_mean = np.mean(u1)
    v1_mean = np.mean(v1)

    u2_mean = np.mean(u2)
    v2_mean = np.mean(v2)

    # TODO: Shift the origin of the points to the centroid
    u1_hat = u1 - u1_mean
    v1_hat = v1 - v1_mean

    u2_hat = u2 - u2_mean
    v2_hat = v2 - v2_mean

    # TODO: Normalize the points so that the largest distance from the origin is equal to
sqrt(2)
    x1_dist = np.sqrt(np.square(u1_hat)+np.square(v1_hat))
    x2_dist = np.sqrt(np.square(u2_hat)+np.square(v2_hat))

    scale1 = np.sqrt(2)/np.max(x1_dist)
    scale2 = np.sqrt(2)/np.max(x2_dist)

    x1_scaled = np.array([u1_hat, v1_hat])*scale1
    x2_scaled = np.array([u2_hat, v2_hat])*scale2

    # TODO: Similarity transform 1 and 2
    S1 = np.array(([scale1, 0, 0],
                [0, scale1, 0],
                [0, 0, 1]))
    S2 = np.array(([scale2, 0, 0],
                [0, scale2, 0],
                [0, 0, 1]))
    L1 = np.array(([1, 0, -u1_mean],
                [0, 1, -v1_mean],
                [0, 0, 1]))
    L2 = np.array(([1, 0, -u2_mean],
                [0, 1, -v2_mean],
                [0, 0, 1]))
    T1 = S1@L1
    T2 = S2@L2
```

```
    # TODO: Compute homography
    H = computeH(x1_scaled.T, x2_scaled.T)

    # TODO: Denormalization
    H2to1 = np.linalg.inv(T1)@H@T2
    return H2to1
```

Q2.2.3

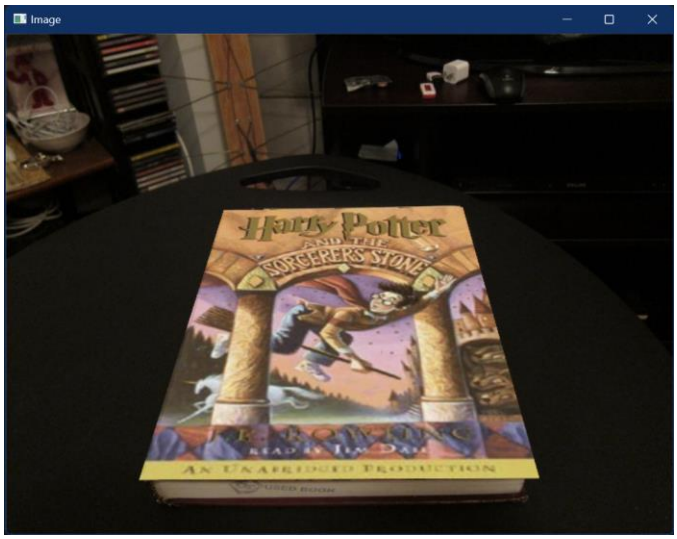
```
def computeH_ransac(locs1, locs2, opts):
    max_iters = opts.max_iters  # the number of iterations to run RANSAC for
    inlier_tol = opts.inlier_tol # the tolerance value for considering a point
to be an inlier

    # Initializing variables
    bestCount = 0
    inlierScore = np.zeros(locs1.shape[0], dtype=int)

    for i in range(max_iters):
        # Generating 4 random points
        randomPoints = np.array(random.sample(range(locs1.shape[0]),4))

        # Selecting 4 points from locs1 and locs2
        randomLocs1 = locs1[randomPoints]
        randomLocs2 = locs2[randomPoints]

        H = computeH_norm(randomLocs1,randomLocs2)

        # Homogenising
        locs1_hom = np.hstack((locs1, np.ones((locs1.shape[0], 1))))
        locs2_hom = np.hstack((locs2, np.ones((locs2.shape[0], 1))))

        # Estimating locs2
        locs1_estimated = H@locs2_hom.T
        locs1_estimated_norm = (locs1_estimated / locs1_estimated[2, :]).T

        # Computing error between original and estimated points
        inliers = locs1_hom - locs1_estimated_norm
        error = np.linalg.norm(inliers, axis = 1)

        # Counting inliers
        score = np.where(error < inlier_tol, 1, 0)
        count = np.sum(score)

        if count > bestCount:
            inlierScore = score
            bestCount = count

    # Computing best homography
    inliers_list = np.array([locs1[inlierScore==1], locs2[inlierScore==1]])
    bestH = computeH_norm(inliers_list[0], inliers_list[1])

    return bestH, score
```

Q2.2.4

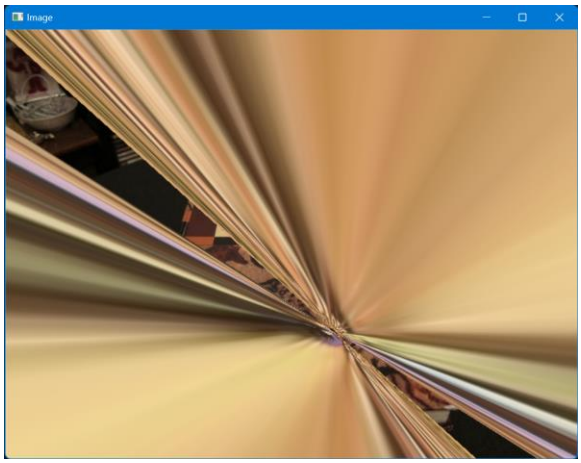
```
import numpy as np
import cv2
import skimage.color
from opts import get_opts
import matplotlib.pyplot as plt

from matchPics import matchPics
from planarH import compositeH
from planarH import computeH_ransac


def warpImage(opts):
    I1 = cv2.imread('data\cv_cover.jpg')
    I2 = cv2.imread('data\cv_desk.png')

    I3 = cv2.imread('data\hp_cover.jpg')
    I3 = cv2.resize(I3, (I1.shape[1], I1.shape[0]))

    matches, locs1, locs2 = matchPics(I1,I2,opts)

    locs1 = locs1[matches[:, 0]]
    locs2 = locs2[matches[:, 1]]

    H,_ = computeH_ransac(locs1, locs2, opts)

    output = compositeH(H, I3, I2)

    cv2.imshow('Image',output)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
    pass

if __name__ == "__main__":

    opts = get_opts()
    warpImage(opts)
```

The reason for improper warping is due to difference in cv_cover and hp_cover sizes. Resizing would solve the issue

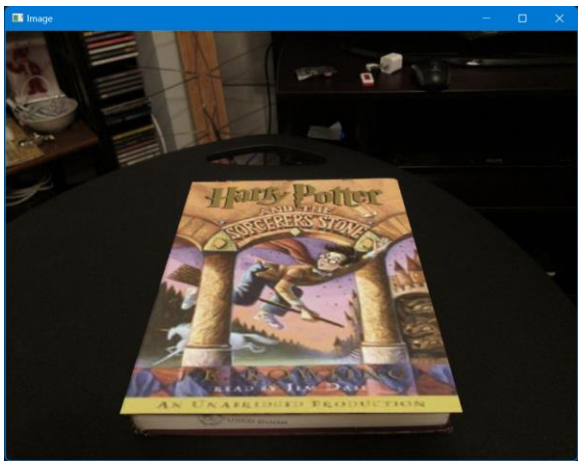
```
def compositeH(H2to1, template, img):

    #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.

    # Inverting Homography for warping
    H2to1 = np.linalg.inv(H2to1)

    # TODO: Create mask of same size as template
    mask = np.ones_like(template)

    # TODO: Warp mask by appropriate homography
    warpedMask = cv2.warpPerspective(mask,H2to1,(img.shape[1], img.shape[0]))

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

    # TODO: Use mask to combine the warped template and the image
    img[np.nonzero(warpedMask)] = warpedTemplate[np.nonzero(warpedMask)]

    return img
```



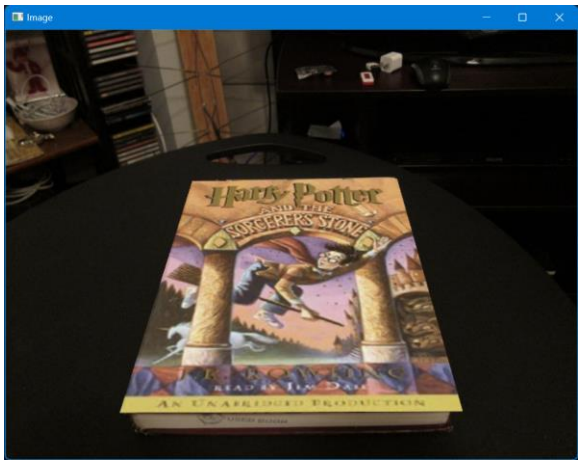
Q2.2.5



max_iters: 5

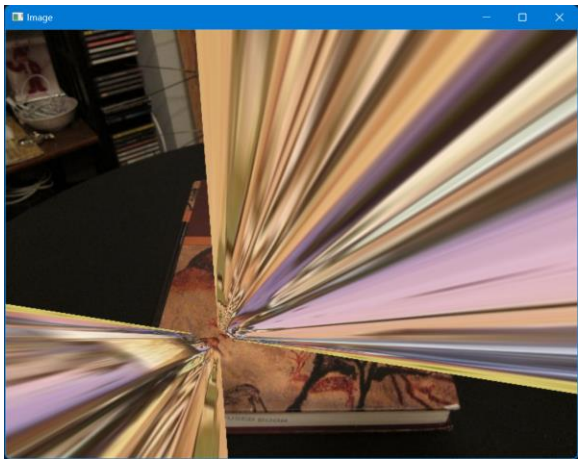


max_iters: 50

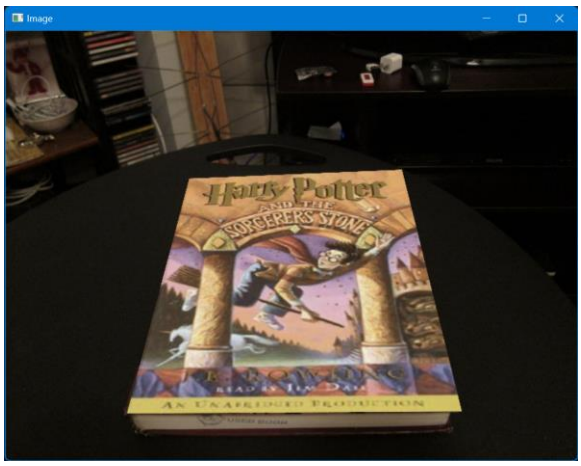


max_iters: 500

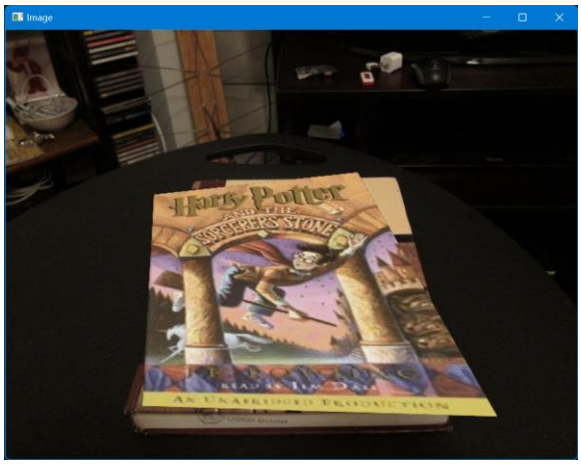
RANSAC needs to compute certain minimum iterations to find best possible homography. Once the threshold is crossed, RANSAC redundantly calculates other homographys.



inlier_tol: 0.01



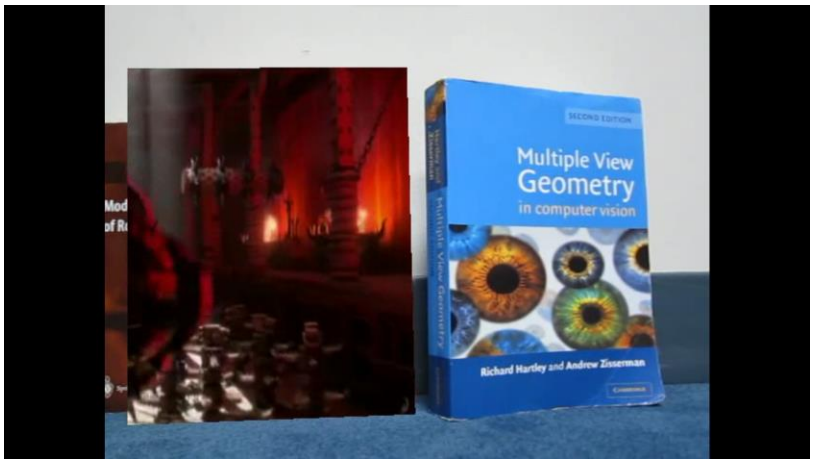
inlier_tol: 0.1



inlier_tol: 25

An optimum inlier tolerance is required.
Having it too low makes it difficult to warp as there would be fewer similar descriptors to match.
Having it too high makes it difficult to warp as there would be many similar descriptors to match.

3.1



Left

Right

Center

https://drive.google.com/file/d/1-vPXETNFUzmVRY3p1ZOjX6VHjtyftFhd/view?usp=sharing

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from opts import get_opts
from helper import loadVid
from matchPics import matchPics
from planarH import compositeH
from planarH import computeH_ransac
opts = get_opts()

# Load videos and cover image
ar = loadVid('C:/D/CMU/Courses/F23/16-820/hw1/data/ar_source.mov')
book = loadVid('C:/D/CMU/Courses/F23/16-820/hw1/data/book.mov')
cv_cover = cv2.imread('C:/D/CMU/Courses/F23/16-820/hw1/data/cv_cover.jpg')

output_video = cv2.VideoWriter('C:/D/CMU/Courses/F23/16-
820/hw1/data/ar_result.avi', cv2.VideoWriter_fourcc('F','M','P','4'), 30,
(book.shape[2], book.shape[1]))

for i in range(ar.shape[0]):
    print('Frame:', i)

    # Extracting current frames
    ar_image = ar[i]
    book_image = book[i]

    # Removing black padding
    ar_image_without_black_padding = ar_image[45:310,:,:]

    # Resizing AR to match cover
    ratio = cv_cover.shape[0]/ar_image_without_black_padding.shape[0]
    resize_width = int(ar_image_without_black_padding.shape[1]*ratio)
    resize_height = int(ar_image_without_black_padding.shape[0]*ratio)
    resized_ar = cv2.resize(ar_image_without_black_padding, (resize_width,
resize_height))

    # Cropping AR
    ar_crop = ar_image_without_black_padding[:,(int(resize_width/2)-
int(cv_cover.shape[1]/2)):(int(resize_width/2)+int(cv_cover.shape[1]/2)),:]
    resized_ar = cv2.resize(ar_crop, (cv_cover.shape[1], cv_cover.shape[0]))

    matches, locs1, locs2 = matchPics(cv_cover,book_image,opts)

    # Skip frame if not enough matches
    if len(matches) < 5:
        print('Skipping frame', i)
        continue
```

```
    # Extracting locations of matches
    locs1 = locs1[matches[:, 0], :]
    locs2 = locs2[matches[:, 1], :]

    H,_ = computeH_ransac(locs1, locs2, opts)

    output = compositeH(H, resized_ar, book_image)

    output_video.write(output)

output_video.release()
```

3.2

Without ORB, the average frame rate was 0.24.

Found ORB has decent matching rate and required less time for compute.

https://arxiv.org/ftp/arxiv/papers/1710/1710.02726.pdf

https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html

https://github.com/methylDragon/opencv-python-reference/blob/master/02%20OpenCV%20Feature%20Detection%20and%20Description.md

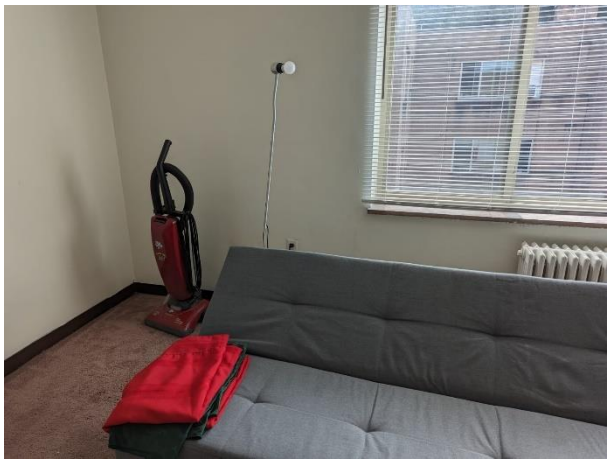
After ORB, the average frame rate was greater than 30

```
import numpy as np
import cv2
from helper import loadVid
from planarH import compositeH
import time

def matchPics(I1, I2):
    orb = cv2.ORB_create()

    locs1, desc1 = orb.detectAndCompute(I1, mask=None)
    locs2, desc2 = orb.detectAndCompute(I2, mask=None)

    # https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html
    # Given: 1. For ORB, use NORM_HAMMING 2.Keep crossCheck=True for best
match
    bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
    matches = bf.match(desc1,desc2)

    # Extracting locations of matches after converting DMatch objects
    points1 = np.array([cv2.KeyPoint_convert([locs1[match.queryIdx]]) for
match in matches])
    points2 = np.array([cv2.KeyPoint_convert([locs2[match.trainIdx]]) for
match in matches])

    return points1, points2

def main():
    # Load videos and cover image
    ars = loadVid('C:/D/CMU/Courses/F23/16-820/hw1/data/ar_source.mov')
    books = loadVid('C:/D/CMU/Courses/F23/16-820/hw1/data/book.mov')
    cv_cover = cv2.imread('C:/D/CMU/Courses/F23/16-820/hw1/data/cv_cover.jpg')

    # Removing black padding
    ar = ars[:,45:310,:,:]
```

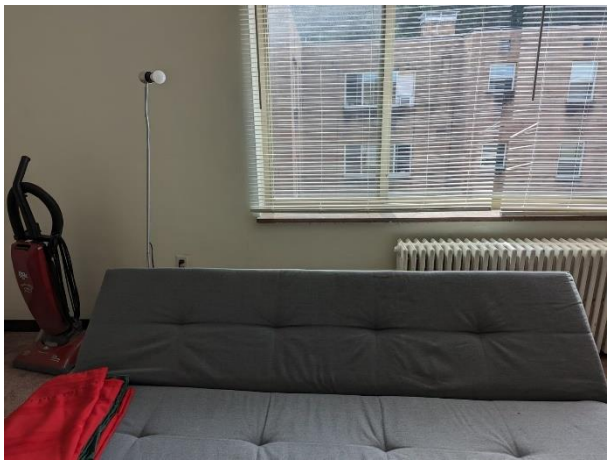
```
    # Resizing AR to match cover
    ratio = cv_cover.shape[0]/ar[0].shape[0]
    resize_width = int(ar[0].shape[1]*ratio)

    initial_time = time.time()
    previous_time = 0

    for i in range(ar.shape[0]):
        book = books[i]

        ar_crop = ar[i][:,(int(resize_width/2)-
int(cv_cover.shape[1]/2)):(int(resize_width/2)+int(cv_cover.shape[1]/2)),:]
        resized_ar = cv2.resize(ar_crop,
dsize=(cv_cover.shape[1],cv_cover.shape[0]))

        locs1, locs2 = matchPics(cv_cover,book)

        H,_ = cv2.findHomography(locs2, locs1, cv2.RANSAC, 6.0)

        composite_img = compositeH(H, resized_ar, book)

        cv2.imshow('composite_img', composite_img)
        cv2.waitKey(1)

        current_time = time.time() - initial_time
        time_difference = current_time - previous_time
        frame_rate = 1 / time_difference
        print('Frame rate:', frame_rate)
        previous_time = current_time

    cv2.destroyAllWindows()

if __name__ == '__main__':
    main()
```

4

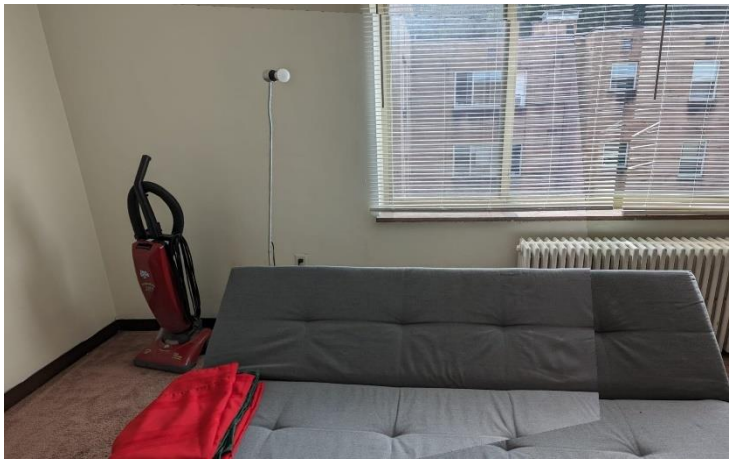
Left



Right



Output



```python
import numpy as np
import cv2
from opts import get_opts
import matplotlib.pyplot as plt
from matchPics import matchPics
from planarH import compositeH
from planarH import computeH_ransac
opts = get_opts()

# Q4
left = cv2.imread('C:/D/CMU/Courses/F23/16-820/hw1/data/lft.jpeg')
right = cv2.imread('C:/D/CMU/Courses/F23/16-820/hw1/data/rght.jpeg')

# Add padding
rightPadded = cv2.copyMakeBorder(right, 0, 0, right.shape[1]//5, 0,
cv2.BORDER_CONSTANT)

matches, locs1, locs2 = matchPics(left,rightPadded,opts)

locs1 = locs1[matches[:, 0], :]
locs2 = locs2[matches[:, 1], :]

H,_ = computeH_ransac(locs1, locs2, opts)

output = compositeH(H, left, rightPadded)

cv2.imwrite('C:/D/CMU/Courses/F23/16-820/hw1/data/test.jpg',output)
```

Collaborated with ddhrafan and dagupta2