Q1.1

According to camera projection matrix:

$\lambda1*[x1\ y1\ 1]^T = H1*[X\ Y\ 1]^T$

$\lambda2*[x2\ y2\ 1]^T = H2*[X\ Y\ 1]^T$

Then:

$\lambda1*[x1\ y1\ 1]^T = H1*H2^{-1}*\lambda2[x2\ y2\ 1]^T$

$\lambda*[x1\ y1\ 1]^T = H1*H2^{-1}*[x2\ y2\ 1]^T$

Thus, $X1 = A1P = A1A2^{-1}P \equiv HX2$

Q1.2

1. h has 8 degrees of freedom. Because h is a column vector reshaped from H (3x3), and the DOB of H is 8. Thus, the DOF of h is also 8.

2. We need 8 points, which means 4 pairs to solve h.

3.

$X_1^i \equiv HX_2^i$

$[u1\ v1\ 1]^T \equiv [[h11\ h12\ h13], [h21\ h22\ h23], [h31\ h32\ h33]] * [u2\ v2\ 1]^T$

Then, we can use a constant n to transform the equation above into “=”:

$[n*u1\ n*v1\ n]^T = [[h11\ h12\ h13], [h21\ h22\ h23], [h31\ h32\ h33]] * [u2\ v2\ 1]^T$

In this way, we can begin calculate the derivation:

H31u2u1+h32v2u1+h33u1 = h11u2+h12v2+h13

H31u2v1+h32v2v1+h33v1 = h21u2+h22v2+h23

So if we arrange h_{ij} in the right, then the left should be A:

[[-u2 –v2 -1 0 0 0 u2u1 v2u1 u1], [0 0 0 –u2 –v2 -1 u2v1 v2v1 v1]] * h

In conclusion:

A = [[-u2 –v2 -1 0 0 0 u2u1 v2u1 u1], [0 0 0 –u2 –v2 -1 u2v1 v2v1 v1]]

4.

The trivial solution for h means all elements in h are zero, so h = [0 0 0 0 0 0 0 0 0]^T.

A is not full rank because its DOF is 8.

If the eigenvalues don’t contain 0, matrix has full rank. As for the eigenvectors, the matrix rank = n-number of eigenvectors corresponding to eigenvalue 0.

Q1.4.1

To prove: $K1[I\ 0]X \equiv H * K2[R\ 0]X$

$X1 = K1*[I\ 0]*X = K1*X$

$X2 = K2*[R\ 0]*X = K2*[R\ 0] * (X1/K1) = K2*R*K1^{-1}*X1$

Assume: $X1 = H * X2$

Then: $H = K1R^{-1}K2^{-1}$

In conclusion: H exists

Q1.4.2

From Q1.4.1 we can know H = $K1R^{-1}K2^{-1}$

So $H^2 = (K1R^{-1}K2^{-1}) * (K1R^{-1}K2^{-1}) = KRRK^{-1}$

R = [[cosθ –sinθ 0], [sinθ cosθ 0], [0 0 1]]

R*R = [[$\cos^2θ–\sin^2θ$ –2sinθcosθ 0], [2sinθcosθ $\cos^2θ–\sin^2θ$ 0], [0 0 1]]

= [[cos2θ –sin2θ 0], [sin2θ cos2θ 0], [0 0 1]] = R(2θ)

In conclusion:

H^2 = K R(2θ) K^{-1}, so H^2 is the homography corresponding to a rotation of 2θ.

Q1.4.3

The planar homography can falsely map points when there are many similar or repeated points among pictures from different views. Also, some space information can be ignored because z-axis is not taken into account in homography method.

Q1.4.4

Assume that we have three points in the 3-dimensional space:

A (x1, y1, z1), B (x2, y2, z2), C (x3, y3, z3)

Consider we have a simple projection matrix P:

[[1 0 0 0], [0 1 0 0], [0 0 0 0], [0 0 0 1]]

Then we project the line into 2D:

x = PX = [[1 0 0 0], [0 1 0 0], [0 0 0 0], [0 0 0 1]] * [[x1 x2 x3], [y1 y2 y3], [z1 z2 z3], [1 1 1]]

= [[x1 x2 x3], [y1 y2 y3], [0 0 0], [1 1 1]]

We can find the original x-axis and y-axis numbers are preserved in 2D space, all the z-axis values are set to 0. Thus, the original line in 3D is projected to the corresponding line in 2D.

Q2.1.1

Harris corner detector uses sliding window to find large change in intensity by moving window in any direction. Thus, this method needs many computations to find the corner point.

On the other hand, FAST detector selects an interest point and only consider a circle of 16 pixels around to check whether there exists 12 continuous pixels around which are brighter and darker than center pixel and threshold. Moreover, to make this process even faster, we can only exam four pixels at 1, 9 and 5, 13, because at least three of them should satisfy the condition if p is the corner.

To include, FAST detector is much less computationally intensive than Harris detector.

Q2.1.2

Filter banks use different filter to classify or smooth the whole picture, with little regarding to spatial information of pictures. BRIEF descriptor selects pairs between patches and compares their similarity, which is a more specific way. We can use difference of Gaussian filters as a descriptor.

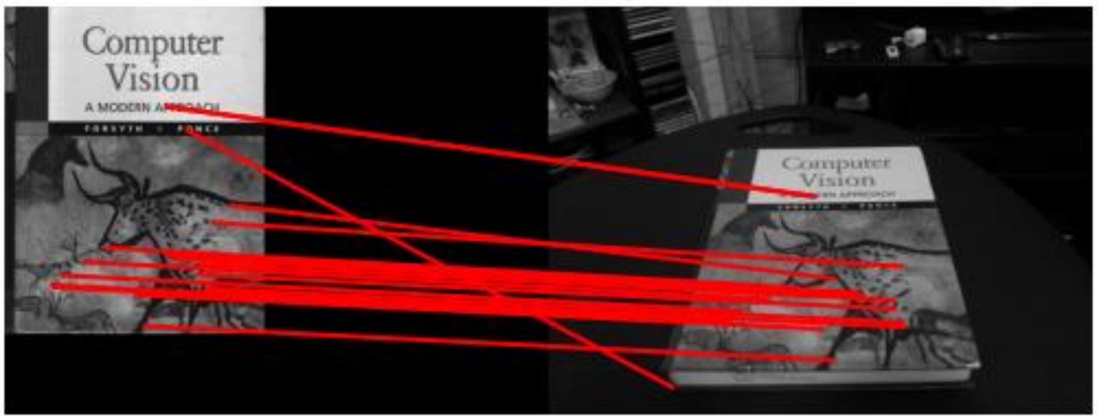
Q2.1.3

Hamming distance calculates the number of different bit positions between two binary strings. So we can first transform BRIEF descriptor to binary strings first, then compute the relative distances between points and find the closest one. The nearest neighbor calculates distances between the target point and all other points, then finds the point pair with the smallest distance. The most common way to calculate distance in nearest neighbor is Euclidean distance.

Euclidean distance computation is slower than Hamming distance, because Hamming distance calculate based on binary strings and XOR operation, which is super-fast and fit for bottom hardware of computers to operate.

Q2.1.4

Sigma = 0.15, ratio = 0.7

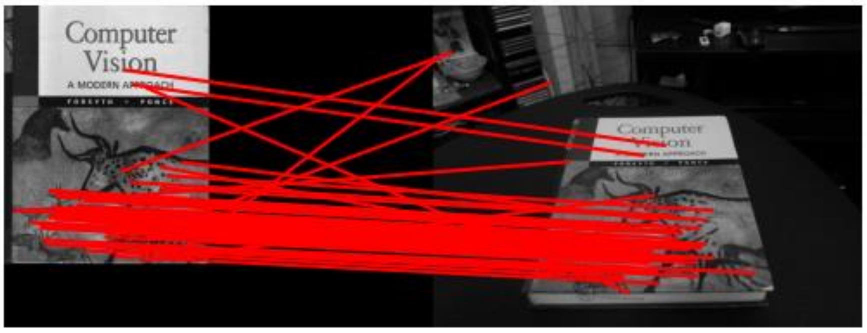


```
import numpy as np
import cv2
import skimage.color
from helper import briefMatch
from helper import computeBrief
from helper import corner_detection
from helper import plotMatches
from opts import get_opts
# Q2.1.4


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

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

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

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

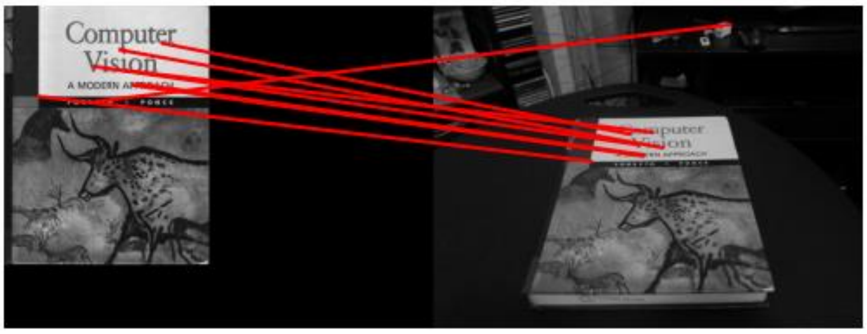
```
feature detector'

    matches, locs1, locs2 = None, None, None

    # TODO: Convert Images to GrayScale
    I1 = cv2.cvtColor(I1, cv2.COLOR_BGR2GRAY)
    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
    I1_desc, locs1 = computeBrief(I1, locs1)
    I2_desc, locs2 = computeBrief(I2, locs2)

    # TODO: Match features using the descriptors
    matches = briefMatch(I1_desc, I2_desc, ratio)

    return matches, locs1, locs2


if __name__ == "__main__":
    opts = get_opts()
    im1 = cv2.imread('../data/cv_cover.jpg')
    im2 = cv2.imread('../data/cv_desk.png')
    matches, locs1, locs2 = matchPics(im1, im2, opts)
    plotMatches(im1, im2, matches, locs1, locs2)
```

Q2.1.5

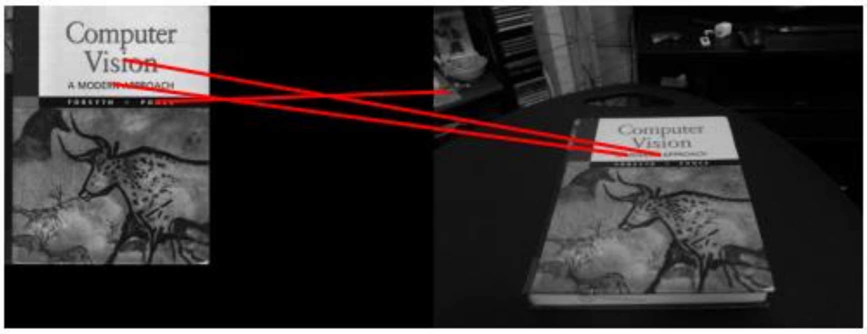
Sigma = 0.1, ratio = 0.7



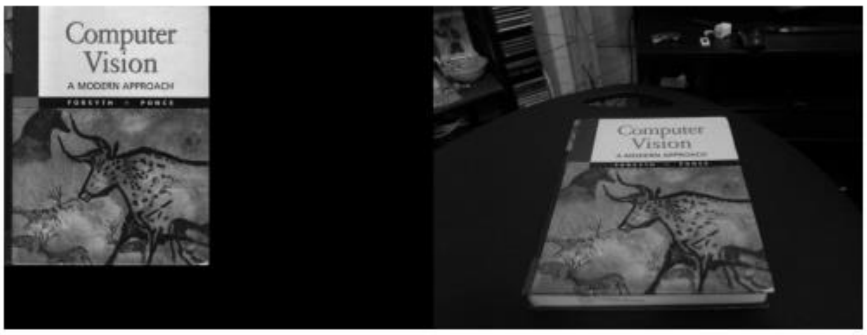
Sigma = 0.3, ratio = 0.7



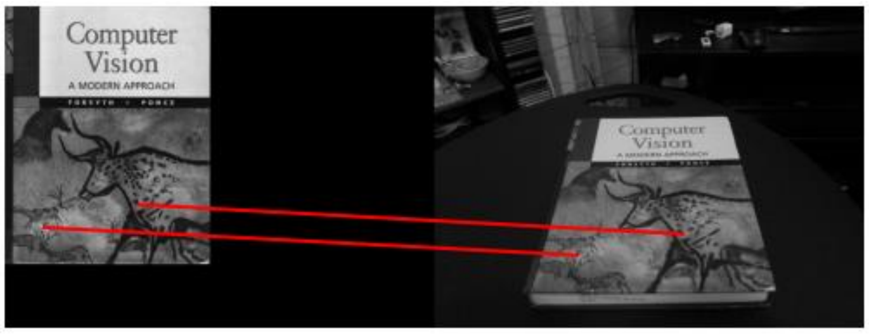
Sigma = 0.5, ratio = 0.7



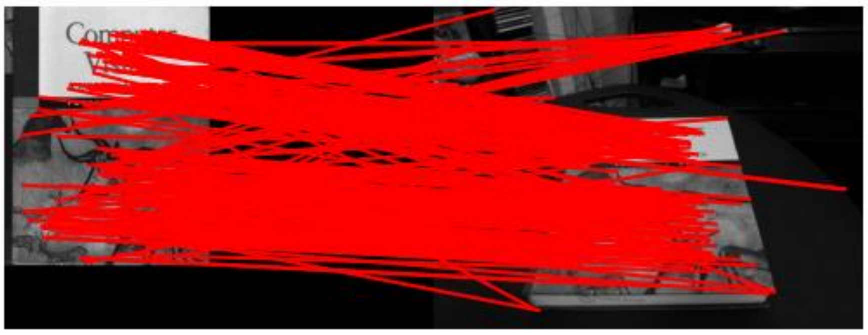
Sigma = 0.15, ratio = 0.1



Sigma = 0.15, ratio = 0.5



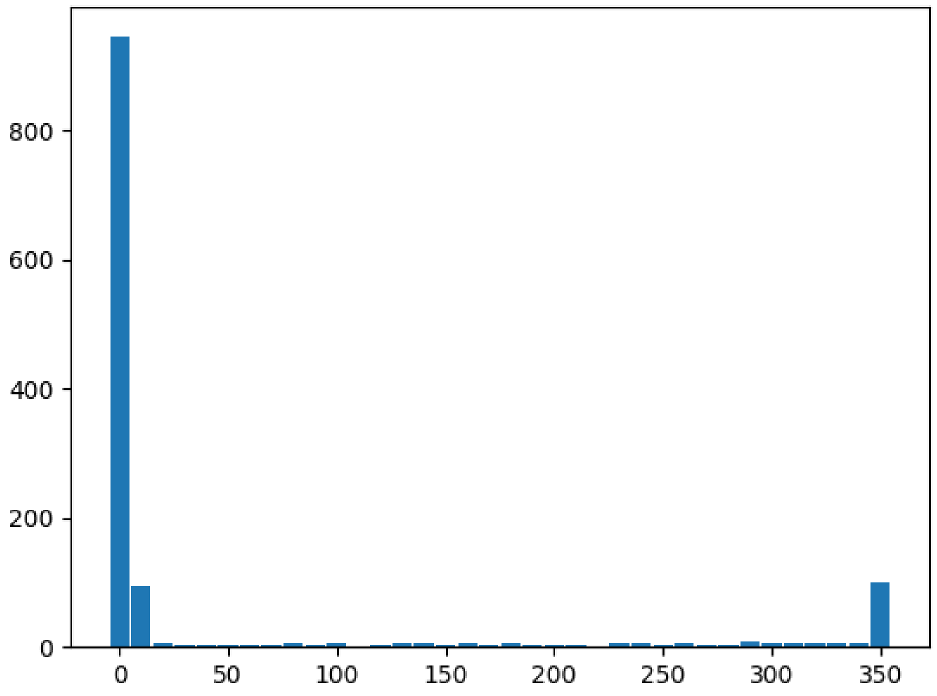
Sigma = 0.15, ratio = 1



We tuned sigma from 0.1 to 0.5 and ratio from 0.1 to 1. It can be noticed that bigger sigma will reduce the number of matches, and also will slightly decrease matching accuracy.

On the other hand, bigger ratio will affect matching number and precision dramatically. When ratio equals to 0.1, there are no matches between pictures. However, there are huge amount of matches when ratio equals 1 and it is clearly that many of them are poor matches.

Q2.1.6



We can see from the figure above that number of matches is the highest when the picture is not rotated. However, the number of matches decreases dramatically when picture rotating, and it remains very low between 10 degrees to 340 degrees. One reason for this result may be BRIEF is not invariant to rotation because BRIEF replies on orientation, which can be changed in rotation process.

```python
def rotTest(opts):
    # Read the image and convert to grayscale, if necessary
    im = cv2.imread('../data/cv_cover.jpg')
    his = []
    for i in range(36):
        # Rotate Image
        im_r = ndimage.rotate(im, 10*i)
        # Compute features, descriptors and Match features
        matches, locs1, locs2 = matchPics(im, im_r, opts)
        # Update histogram
        his.append(matches.shape[0])
        pass

    # Display histogram
    degrees = [i*10 for i in range(36)]
    plt.bar(degrees, his, width=9)
    plt.show()
```

Q2.2.1

```
    def computeH(x1, x2):
    # Q2.2.1
    # Compute the homography between two sets of points
    A = np.zeros((x1.shape[0]*2, 9))
    for i in range(x1.shape[0]):
        u1, v1 = x1[i,0], x1[i,1]
        u2, v2 = x2[i,0], x2[i,1]
        # [xi, yi, 1, 0, 0, 0, -xi*ui, -yi*ui, -ui]
        A[i*2, :] = [u2, v2, 1, 0, 0, 0, -u2*u1, -v2*u1, -u1]
        # [0, 0, 0, xi, yi, 1, -xi*vi, -yi*vi, -vi]
        A[i*2+1, :] = [0, 0, 0, u2, v2, 1, -u2*v1, -v2*v1, -v1]

    # remove inf or NAN to prevent SVD couldn't converge
    A[np.isinf(A)] = 0
    A[np.isnan(A)] = 0
    u, s, vh = np.linalg.svd(A)
    # Vector(s) with the singular values, within each vector sorted
in descending order.
    # So the eig_vector corresponding to the min eig_value lies in
the last of vh
    min_eigVector = vh[-1]
    H2to1 = min_eigVector.reshape(3, 3)
    return H2to1
```

Q2.2.2

```python
def computeH_norm(x1, x2):
    # Q2.2.2
    # Compute the centroid of the points
    x1_centerX = np.average(x1[:, 0])
    x1_centerY = np.average(x1[:, 1])
    x2_centerX = np.average(x2[:, 0])
    x2_centerY = np.average(x2[:, 1])

    # Shift the origin of the points to the centroid
    x1_ori, x2_ori = np.zeros((x1.shape[0], 2)),
np.zeros((x2.shape[0], 2))
    x1_ori[:, 0] = x1[:, 0] - x1_centerX
    x1_ori[:, 1] = x1[:, 1] - x1_centerY
    x2_ori[:, 0] = x2[:, 0] - x2_centerX
    x2_ori[:, 1] = x2[:, 1] - x2_centerY

    # Normalize the points so that the largest distance from the
origin is equal to sqrt(2)
    x1_maxDist = np.max(np.linalg.norm(x1_ori, axis=1))
    x2_maxDist = np.max(np.linalg.norm(x2_ori, axis=1))
    s1, s2 = np.sqrt(2)/x1_maxDist, np.sqrt(2)/x2_maxDist
    T1 = [[s1, 0, -s1*x1_centerX], [0, s1, -s1*x1_centerY], [0, 0,
1]]
    T2 = [[s2, 0, -s2*x2_centerX], [0, s2, -s2 * x2_centerY], [0, 0,
1]]

    # Similarity transform 1
    x1_h = np.concatenate((x1, np.ones((x1.shape[0], 1))), axis=1)
    x1_norm = np.zeros((x1.shape[0], 2))
    for i in range(x1.shape[0]):
        x1_norm[i] = np.matmul(T1, x1_h[i])[0:2]

    # Similarity transform 2
    x2_h = np.concatenate((x2, np.ones((x2.shape[0], 1))), axis=1)
    x2_norm = np.zeros((x2.shape[0], 2))
    for i in range(x2.shape[0]):
        x2_norm[i] = np.matmul(T2, x2_h[i])[0:2]

    # Compute homography
    H2to1_h = computeH(x1_norm, x2_norm)
    # Denormalization
    H2to1 = np.linalg.inv(T1) @ H2to1_h @ T2
    return H2to1
```

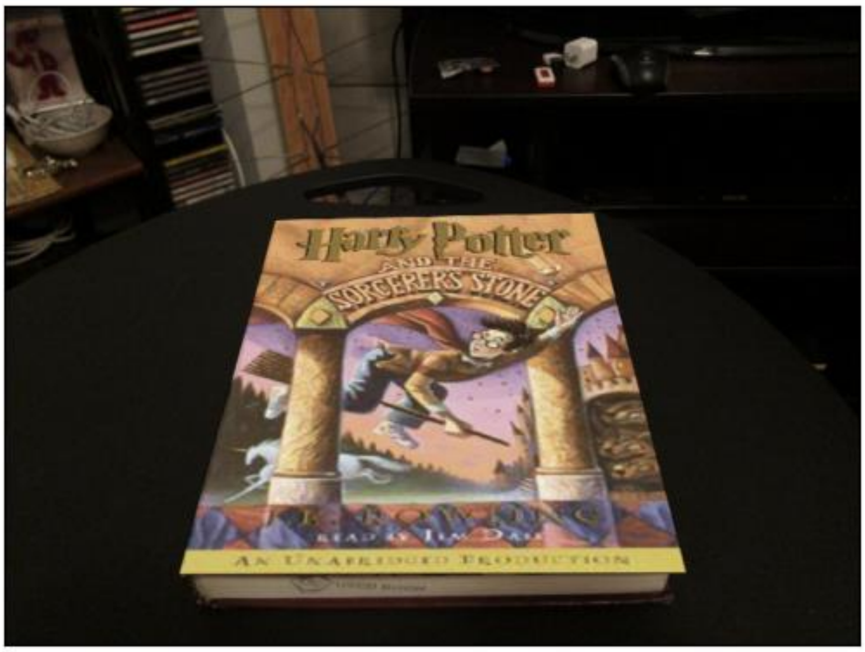
Q2.2.3

```
    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
    # the tolerance value for considering a point to be an inlier
    inlier_tol = opts.inlier_tol

    inliers = np.zeros(locs1.shape[0])
    max_inliers = float('-inf')     # the max number of inliers
    bestH2to1 = np.zeros((3, 3))
    predict_h = np.zeros((locs1.shape[0], 3))
    predict = np.zeros((locs1.shape[0], 2))     # predict points
coordinates
    for i in range(max_iters):
        # at least 4 point pairs needed to compute H
        ran_rows = np.random.choice(locs1.shape[0], 4)
        tmp_h = computeH_norm(locs1[ran_rows, :], locs2[ran_rows, :])

        for j in range(locs1.shape[0]):
            predict_h[j, :] = np.matmul(tmp_h, np.append(locs2[j, :],
1))
            predict[j, :] = (predict_h[j, :]/predict_h[j, 2])[0:2]

        # remove possible inf or NAN caused by divide 0
        predict[np.isinf(predict)] = 0
        predict[np.isnan(predict)] = 0
        bias = np.linalg.norm(locs1 - predict, axis=1)
        index = np.where(bias <= inlier_tol)
        if index[0].shape[0] > max_inliers:
            max_inliers = index[0].shape[0]
            for point in index:
                inliers[point] = 1
            bestH2to1 = tmp_h

    return bestH2to1, inliers
```

Q2.2.4

4.

We can see that although the hp_cover.jpg doesn't fill up the same space as cv_desk.jpg. I think the main reason is that their image sizes are different. So I plan to resize hp_cover.jp into the size of cv_desk.jpg, then they should be overlapped better.



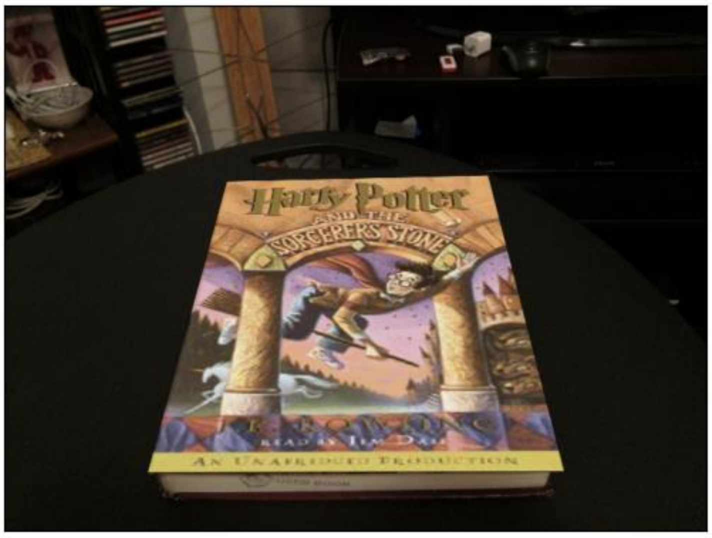
```
def compositeH(H2to1, template, img):

    # Create a composite image after warping the template image on
top
    # of the image using the homography

    # Note that the homography we compute is from the image to the
template;
    #x_template = H2to1*x_photo
    # For warping the template to the image, we need to invert it.

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

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

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

```
    # Use mask to combine the warped template and the image
    index = np.all(warp_mask, axis=2)      # find the index we need
replace (warp_mask=1)
    img[index, :] = warp_template[index, :]
    composite_img = img

    return composite_img
```

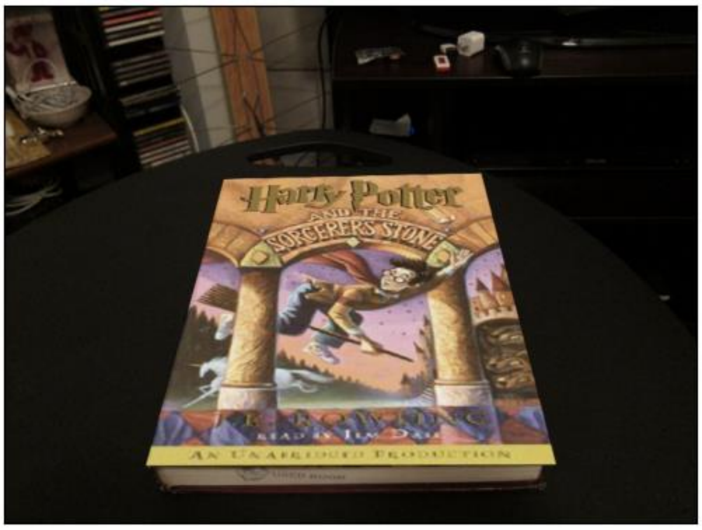
```
def warpImage(opts):
    opts = get_opts()
    cv_cover = cv2.imread('../data/cv_cover.jpg')
    cv_desk = cv2.imread('../data/cv_desk.png')
    hp_cover = cv2.imread('../data/hp_cover.jpg')
    matches, locs1, locs2 = matchPics.matchPics(cv_cover, cv_desk,
opts)

    # flip the RGB channel to print original picture in the result
    hp_cover = np.flip(hp_cover, axis=2)
    cv_desk = np.flip(cv_desk, axis=2)

    # scaling
    x1, x2 = np.zeros(locs1.shape), np.zeros(locs2.shape)
    x1[:, 0] = locs1[:, 1] * hp_cover.shape[1] / cv_cover.shape[1]
    x1[:, 1] = locs1[:, 0] * hp_cover.shape[0] / cv_cover.shape[0]
    x2[:, 0], x2[:, 1] = locs2[:, 1], locs2[:, 0]
    locs1, locs2 = x1[matches[:, 0]], x2[matches[:, 1]]

    # Use matching locs1 and locs2 to compute H
    bestH2to1, inliers = planarH.computeH_ransac(locs1, locs2, opts)

    # Use H to transform hp cover, with same output size as cv desk
    composite = planarH.compositeH(bestH2to1, hp_cover, cv_desk)
    plt.imshow(composite)
    # hide x-axis and y-axis
    ax = plt.gca()
    ax.axes.xaxis.set_visible(False)
    ax.axes.yaxis.set_visible(False)
    plt.show()
```

Q2.2.5

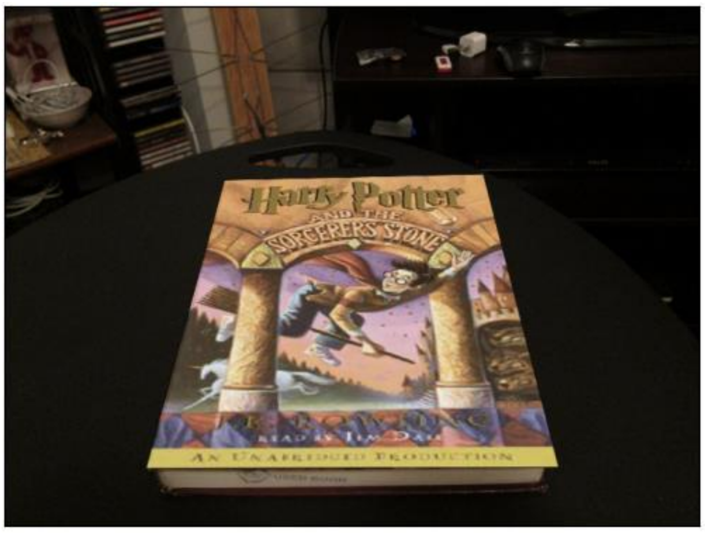
max iters=200, inlier tol =2.0:



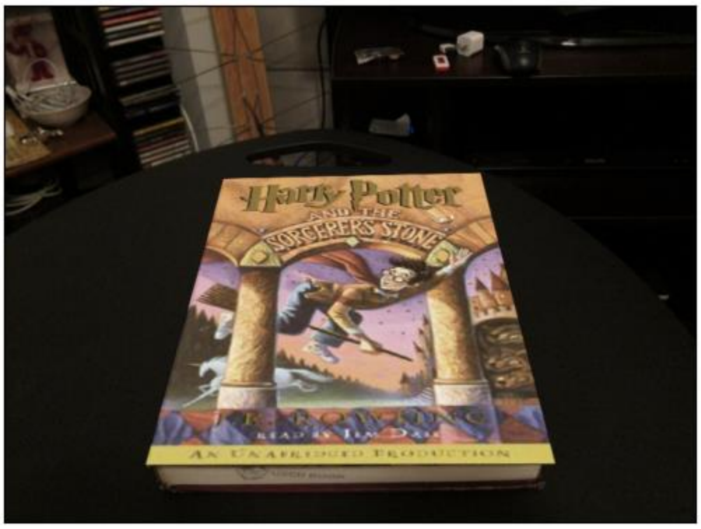
max iters=1000, inlier tol =2.0:



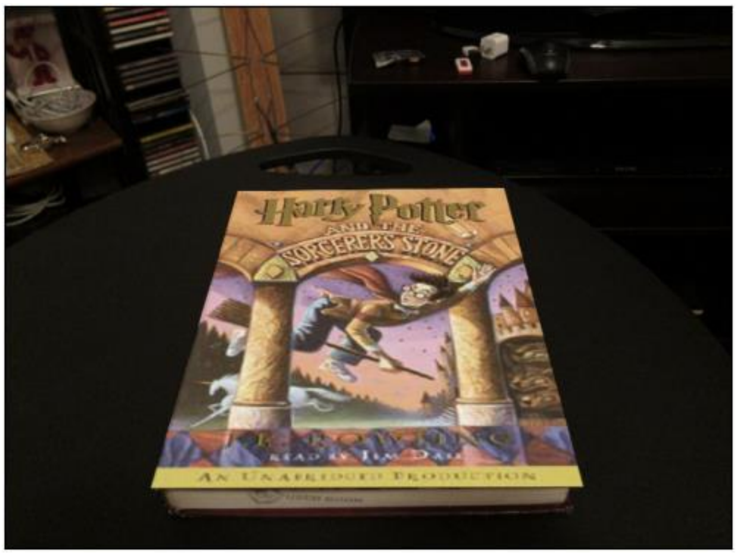
max iters=2000, inlier tol =2.0:



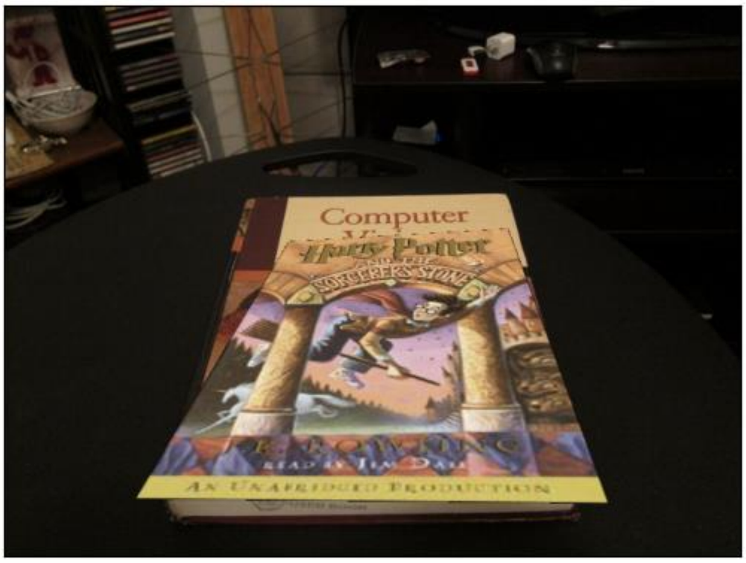
max iters=5000, inlier tol =2.0:



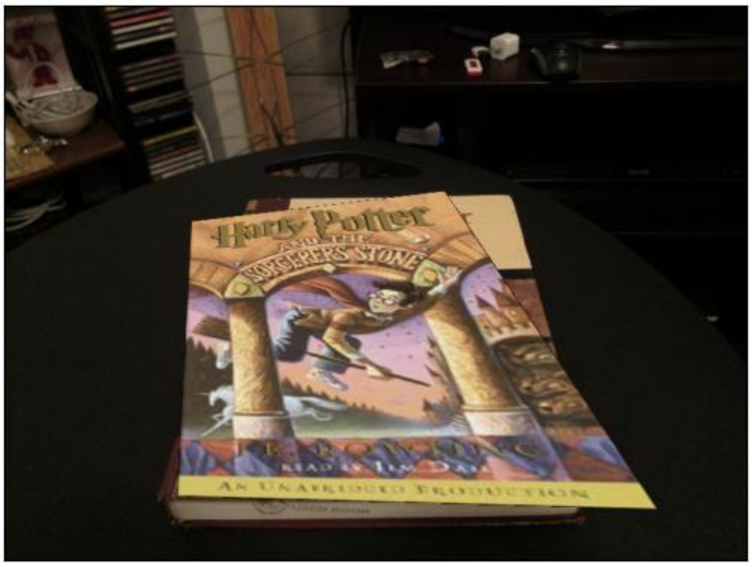
max iters=5000, inlier tol =1.0:



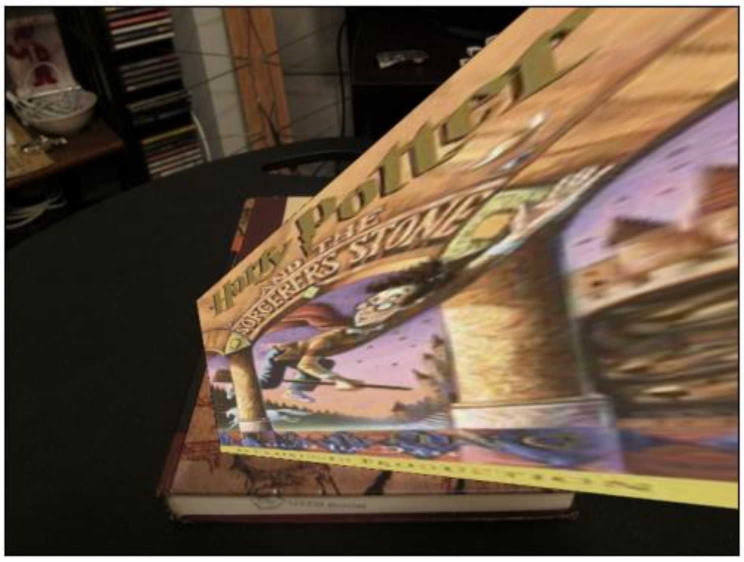
max iters=5000, inlier tol =5.0:



max iters=5000, inlier tol =10.0:



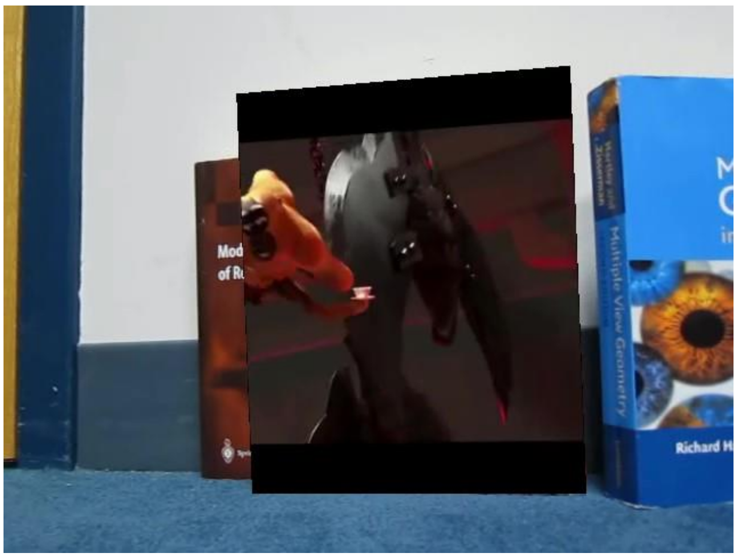
max iters=5000, inlier tol =50.0:



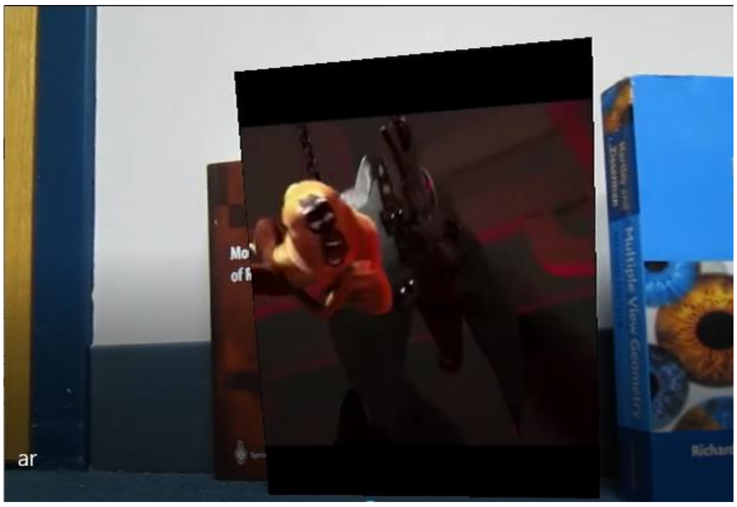
Conclusions:
Max iterations seem don't have huge impact on the final result. We tuned it from 200 to 5000 and the differences between them are very slight. One reason behind this may be the RANSAC algorithm can find the best homography matrix within small numbers of iteration. Thus, the final result will not change significantly after surpassing a certain iteration number.
On the other hand, inlier totals have a big weight on the final result. This parameter regulates the max tolerance for a point to be considered as an inlier, so the bigger it is the more inliers will be returned. Therefore, we can see from the above results, the overlapping effects turn to become worse when inlier totals exceed 5.0. In practical, it would be better to set inlier total within 5.0.

Q3.1

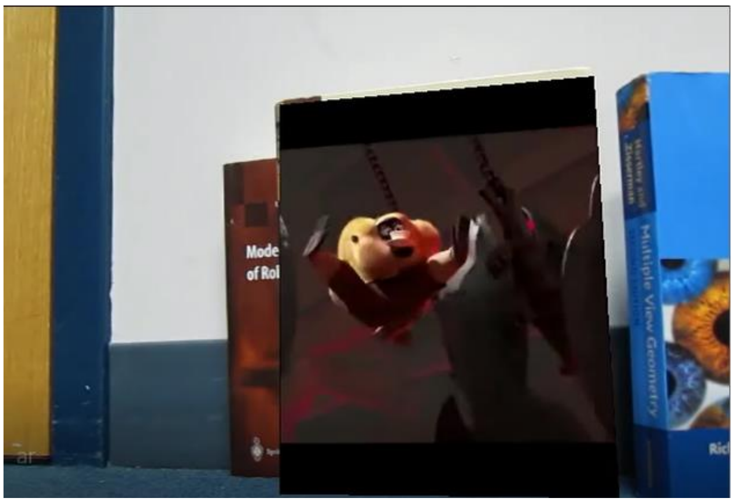
Center:



Left:



Right:



Google Drive link:

https://drive.google.com/file/d/1znG_1a5tmqgCR_phIkyasX32cfHW_HZs/view?usp=sharing

```python
import numpy as np
import cv2
from helper import loadVid
from opts import get_opts
import matchPics
import planarH
import multiprocessing
import os
import datetime

opts = get_opts()
source = loadVid('../data/ar_source.mov')
book = loadVid('../data/book.mov')
cv_cover = cv2.imread('../data/cv_cover.jpg')
output = cv2.VideoWriter('../result/ar.avi',
cv2.VideoWriter_fourcc(*'DIVX'), 25, (book[0].shape[1],
book[0].shape[0]))
output_length = min(len(source), len(book))


# Q3.1
def ar_process():
    output_frame = np.zeros((source.shape[1],
source.shape[1]*cv_cover.shape[1]//cv_cover.shape[0]))
    offset = (source.shape[2] - output_frame.shape[1])//2
    list_args = []
    for i in range(output_length):    # use the shorter length for
your video
        source_frame = source[i]
        book_frame = book[i]
        # crop the source frame
        output_frame = source_frame[:,
offset:offset+output_frame.shape[1], :]
        list_args.append((book_frame, output_frame, i))

    p = multiprocessing.Pool(processes=8)
    p.map(single_process, list_args)

    for i in range(output_length):
        composite = np.load('../result/'+str(i)+'.npy')
        os.remove('../result/'+str(i)+'.npy')
        output.write(composite)
    output.release()
```

```python
def single_process(args):   # similar as warpImage()
    book_frame, output_frame, index = args
    matches, locs1, locs2 = matchPics.matchPics(cv_cover, book_frame,
opts)

    # scaling
    x1, x2 = np.zeros(locs1.shape), np.zeros(locs2.shape)
    x1[:, 0] = locs1[:, 1] * output_frame.shape[1] /
cv_cover.shape[1]
    x1[:, 1] = locs1[:, 0] * output_frame.shape[0] /
cv_cover.shape[0]
    x2[:, 0], x2[:, 1] = locs2[:, 1], locs2[:, 0]
    locs1, locs2 = x1[matches[:, 0]], x2[matches[:, 1]]

    # Use matching locs1 and locs2 to compute H
    bestH2to1, inliers = planarH.computeH_ransac(locs1, locs2, opts)

    # Use H to transform hp cover, with same output size as cv desk
    composite = planarH.compositeH(bestH2to1, output_frame,
book_frame)
    np.save('../result/'+str(index), composite)


if __name__ == "__main__":
    start = datetime.datetime.now()
    ar_process()
    end = datetime.datetime.now()
    print('Time:', (end-start).total_seconds()//60, 'mins')
```