16-720B HW2 Writeup

Nirmay Singaram

nsingara@andrew.cmu.edu

1 1.1

Q1.1

Let P_1, P_2 be the projection matrices representing the cameras C, C' respectively. P_1, P_2 are 3×4. Let $P_1[A_1|b_1]$ and $P_2 = [A_2|b_2]$. Since these are camera projections we know that A_1 and A_2 have determinant non-zero implies invertible.

We know that $x_1 = A_1 x_\pi + b_1$ and $x_2 = A_2 x_\pi + b_2$.

So $x_\pi = A_1^{-1}(x_1 - b_1)$ and

$x_2 = A_2(A_1^{-1}(x_1 - b_1)) + b_2 = A_2 A_1^{-1} x_1 - A_2 A_1^{-1} b_1 + b_2$

Thus our homography H will simply act on x_2 as above, i.e, by doing $A_2 A_1^{-1} x_1 - A_2 A_1^{-1} b_1 + b_2$ (which is an affine transformation) to get x_2. I won't put this in algebraic form since the question said we don't have to but clearly such an H exists.

So basically since projections are affine transformations, with the linear parts being invertible, we may undo these transformations. Then undoing P_1 then *doing* P_2 will make you go from x_1 to x_2 which is our homography H. □

2 1.2

Q1.2

1. $\mathbf{H}$ is a 3×3 matrix. Rearrange this into a column vector gives us that $\mathbf{h}$ is a 9×1 column vector. Then $\boldsymbol{A_i h} = 0$ is an equation in 9 variables, thus giving us 9 degrees of freedom.

Alternately one could also say that the 9th variable is determined by the first 8 variables and so there are actually 8 degrees of freedom.

2. 8 degrees of freedom would require 8 equations to solve, and each A_i supplies two equations (as we'll see in the next question A_i has two rows), and so each A_i (each point correspondence) gives us 2 equations where we need 8 in total, and so we need 4 point correspondences to solve for $\boldsymbol{h}$.

3. Fix an i. Then the equation of $x_1^i = H x_2^i$ gives us:

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

and so we get the equations:
$x_1 = h_{11}x_2 + h_{12}y_2 + h_{13}$
$y_1 = h_{21}x_2 + h_{22}y_2 + h_{23}$
$1 = h_{31}x_2 + h_{32}y_2 + h_{33}$
which gives us the equations:
$-h_{11}x_2 - h_{12}y_2 - h_{13} + h_{31}x_2x_1 + h_{32}y_2x_1 + h_{33}x_1 = 0$
$-h_{21}x_2 - h_{22}y_2 - h_{23} + h_{31}x_2y_1 + h_{32}y_2y_1 + h_{33}y_1 = 0$

Here we just used the third equation and plugged it in the first two as $x_1 = (1)x_1$ and $y_1 = (1)y_1$. This gives us the following matrix equation:

$$\begin{bmatrix} -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2x_1 & y_2x_1 & x_1 \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2y_1 & y_2y_1 & y_1 \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = 0$$

And so $\boldsymbol{A_i} = \begin{bmatrix} -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2x_1 & y_2x_1 & x_1 \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2y_1 & y_2y_1 & y_1 \end{bmatrix}$

4. A trivial solution of course would be that $\boldsymbol{h} = 0$.

If there were non trivial solutions (i.e $\boldsymbol{h} \neq 0$ such that $\boldsymbol{Ah} = 0$) then this would mean that $\boldsymbol{A}$ is *not* full rank, since the null space would be non-trivial (dimension ≥ 1) and by the rank nullity theorem, this would make the rank be at least 1 *less* than full rank. This would mean that one of the eigenvalues would be 0, $\boldsymbol{h}$ solutions to this equation would be the eigenvectors corresponding to the eigenvalue of 0.

3 1.4

Q1.4.1

As the intrinsic matrices for the cameras, we should have that $\boldsymbol{K_1}, \boldsymbol{K_2}$ are invertible (they are the non-augmented part of the affine transformation that is the camera). Moreover, rotation matrices always have inverses (the inverse to a θ rotation is a $-\theta$ rotation).

To invert the first projection we get that $(\boldsymbol{K_1}[\boldsymbol{I}0])^{-1} = [\boldsymbol{I}0]^{-1}K_1^{-1} = [\boldsymbol{I}0]^T K_1^{-1}$ and thus $\boldsymbol{X} = [\boldsymbol{I}0]^T K_1^{-1}\boldsymbol{x}_1$ and as $\boldsymbol{x}_2 = \boldsymbol{K}_2[\boldsymbol{R}0]\mathbf{X} = \boldsymbol{K}_2[\boldsymbol{R}0][\boldsymbol{I}0]^T K_1^{-1}\boldsymbol{x}_1$

And since $[\boldsymbol{R}0][\boldsymbol{I}0]^T = [\boldsymbol{R}0]$ we get that the homography is $\boldsymbol{H} = \boldsymbol{K}_2[\boldsymbol{R}0]K_1^{-1}\boldsymbol{x}_1$.

Q1.4.2

$\boldsymbol{H}^2 = (\boldsymbol{K}[\boldsymbol{R}0]\boldsymbol{K}^{-1})(\boldsymbol{K}[\boldsymbol{R}0]\boldsymbol{K}^{-1}] = \boldsymbol{K}[\boldsymbol{R}0]\boldsymbol{K}^{-1}\boldsymbol{K}[\boldsymbol{R}0]\boldsymbol{K}^{-1} = \boldsymbol{K}[\boldsymbol{R}^2 0]\boldsymbol{K}^{-1}$

If R is the rotation matrix by θ then $R^2 = R \times R$ is the rotation matrix by 2θ (since it rotates by θ, twice), and so $\boldsymbol{H}^2 = \boldsymbol{K}[\boldsymbol{R}0]\boldsymbol{K}^{-1}$ is the homography for rotation by 2θ (of a camera about its own center).

Q 1.4.3

For instance if one of the assumptions of the system doesn't hold- say, the true world points are not on a plane and are instead on a more complex shape (ex: a curved $2-d$ object like a cylinder). Or, say, if the camera moves via more complex actions than simply displacement and rotation, such as rotating about a point not at its center or on an axis not parallel to the plane, etc. In these scenarios our planar homography is not equipped to handle the scene image change since the changes are occuring in more dimensions than being handled by us.

Q1.4.4

Suppose we have a line in $3d$ given by the equation $p+tv$ where p is a point in $3d$ space and v is a direction vector, with $t \in \mathbb{R}$ determining points on the line.

Then we know using the linear/other properties of the projection matrix that $\boldsymbol{P}(p+tv) = \boldsymbol{P}(p) + \lambda \boldsymbol{P}(v) + q$ where λ is some new element of $\mathbb{R}$ that is affected by $\boldsymbol{P}$'s scaling. q is an element of $\mathbb{R}^2$ that represents the affine part of $\mathbb{P}$ and then $\boldsymbol{P}(p) + \lambda \boldsymbol{P}(v) + q$ is the equation of a straightline in $2d$ with point $\boldsymbol{P}(p) + q$ and direction vector $\boldsymbol{P}(v)$. So $\boldsymbol{P}$ sends a line to a line.

4 2.1

Q2.1.1

The Harris detector sort of like a convolution slides a 'mask' of a certain scale across the image, calculates the horizontal and vertical gradients and looks at the correlation between them to determine corner-ness.

In contrast, the FAST detector works by selecting single pixels in areas of interest, and checking the intensities of a circular region around it. A sufficient change in intensity (in either brighter or darker direction) would lead us to classify such a pixel as a corner.

FAST detectors are thus faster since we select single pixels rather than sliding 'masks' across the image. The linked paper supports this claim with FAST being faster than Harris.

Q2.1.2

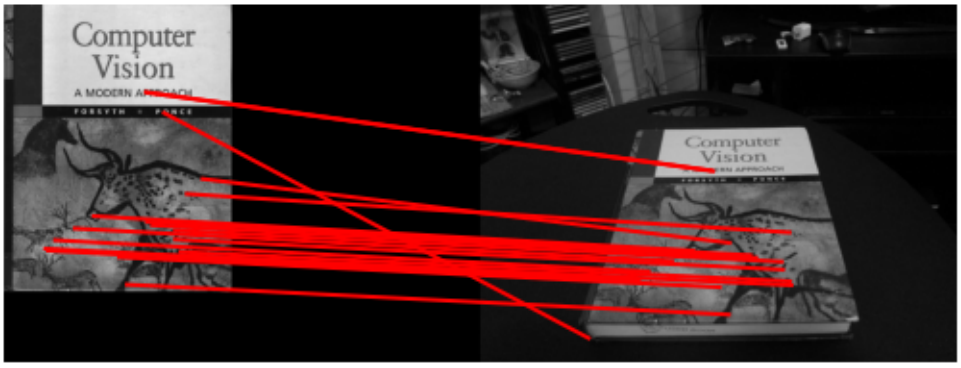
Given a selected pixel, BRIEF isolates a patch around this pixel. After applying Gaussian Filters to this patch, we may compare (via binary tests) the result to some stock set of point pairs. Say the set has size K. Then the results from this comparison will be stored in a binary vector of size K, which will act as the descriptor for said patch. The filter banks we used are not binary descriptors unlike the BRIEF ones so I don't think we can use them as descriptors.

Q2.1.3

BRIEF descriptors are binary vectors. The Hamming distance just requires a simple XOR addition to compute (since if two bits are different we either get $1 \oplus 0 = 1$ or $0 \oplus 1 = 1$ and so the the XOR addition will mark all the bits where the two vectors differ). XOR addition is very computationally efficient so it is desirable to Euclidean calculations in this sense.

More generally, the Euclidean distance is actually the square root of the Hamming distance, since the Euclidean distance will be $\sqrt{\sum_{(x_i-y_i)^2}}$ where $(x_i), (y_i)$ are the binary vectors. Each summand will be non-zero (and thus 1) only when x_i and y_i differ and $1^2 = 1 \implies$ that the interior of the square root also simply calculates the number of bits that are different and then applies a square root on top. So we're essentially finding the same thing with a less computationally efficient method.

Q2.1.4



The following is the code snippet for the file matchpics.Py:

```
import numpy as np
import cv2
import skimage.color
from helper import briefMatch
from helper import computeBrief
```

```
from helper import corner_detection

# 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'
        # 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
        descriptors_1, locs1 = computeBrief(I1, locs1)
        descriptors_2, locs2 = computeBrief(I2, locs2)
        # TODO: Match features using the descriptors
        matches = briefMatch(descriptors_1, descriptors_2, ratio)
        return matches, locs1, locs2
```

We also created an additional file that basically calls this function and applies it to the files in question. I called this file answer_generator.py

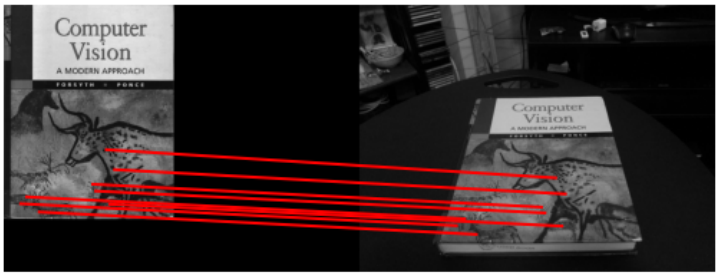
```
#This file mostly just executes questions/programs that I've been asked to fill out.
import cv2
from matchPics import matchPics
from helper import plotMatches
from opts import get_opts

opts = get_opts()
#I used the same image as given in the example
cv_frontal = cv2.imread('../data/cv_cover.jpg')
```

```
10  cv_obscured = cv2.imread('../data/cv_desk.png')
11
12  matches, loc1, loc2 = matchPics(cv_frontal, cv_obscured, opts)
13  plotMatches(cv_frontal, cv_obscured, matches, loc1, loc2)
```

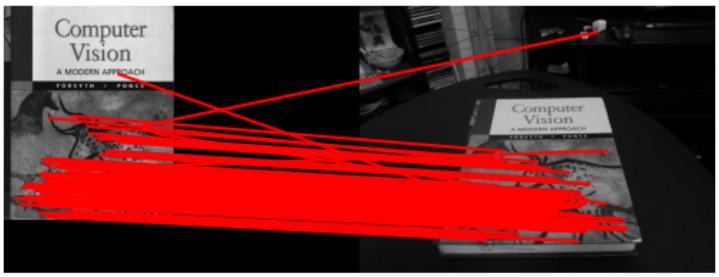
Q2.1.5

I chose sigmas from the set of $\{0.05, 0.15, 0.25\}$ and ratio from the set of $\{0.5, 0.7, 0.9\}$. This gave us a total of 9 different combinations which I show here:

$sigma = 0.05$, $ratio = 0.5$:

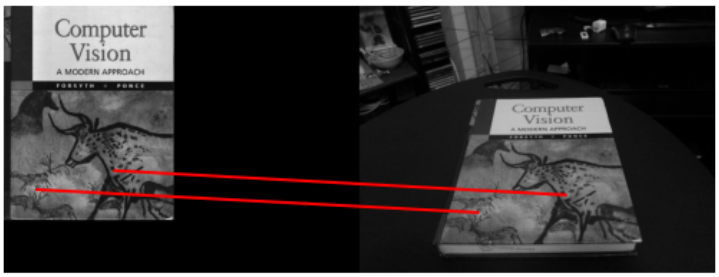


$sigma = 0.05$, $ratio = 0.7$:

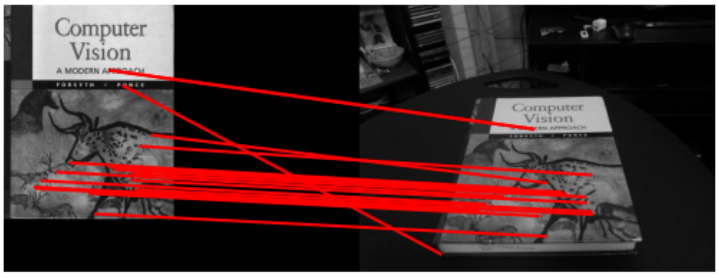


$sigma = 0.05$, $ratio = 0.9$:

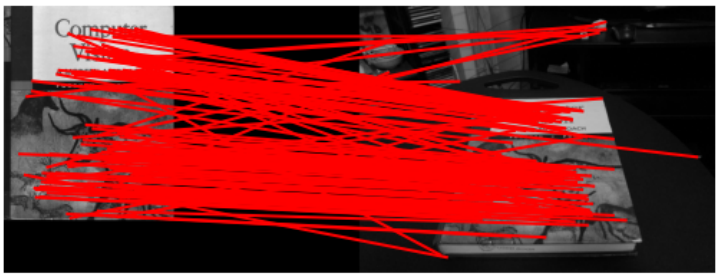
$sigma = 0.15$, $ratio = 0.5$:



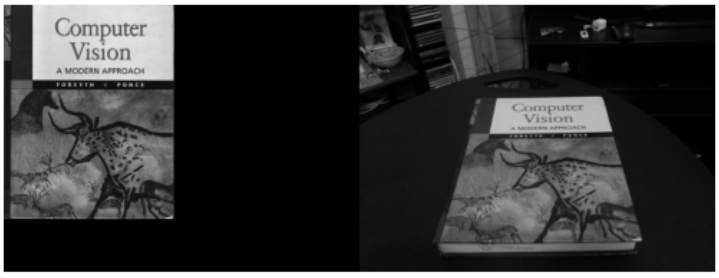
$sigma = 0.15$, $ratio = 0.7$:



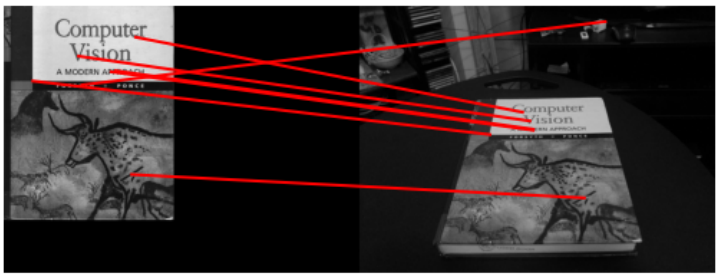
$sigma = 0.15$, $ratio = 0.9$:



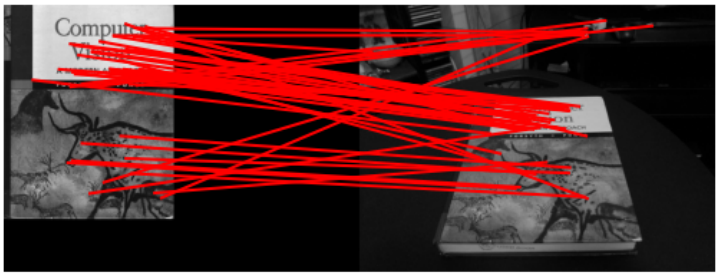
$sigma = 0.25$, $ratio = 0.5$:



$sigma = 0.25$, $ratio = 0.7$:



$sigma = 0.25$, $ratio = 0.9$:

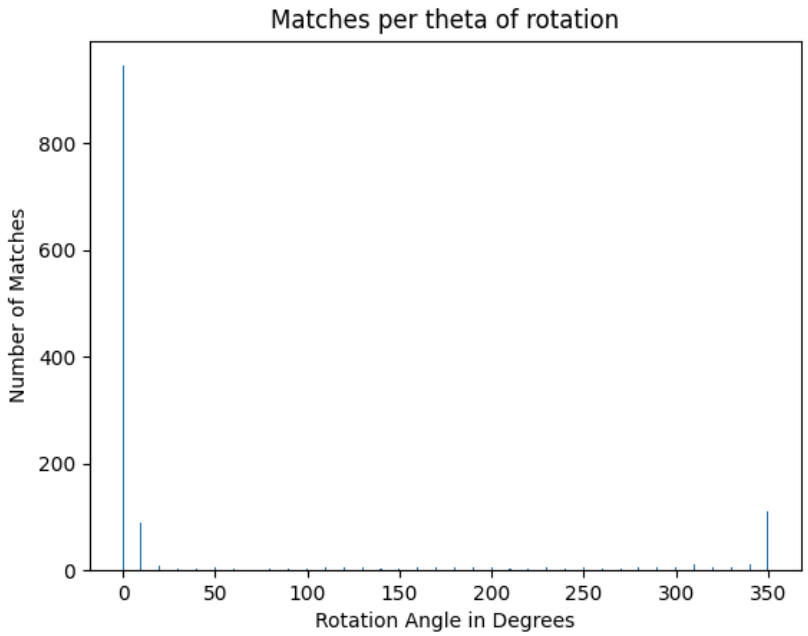


As we can see, increasing *sigma* leads to a lower number of matched points

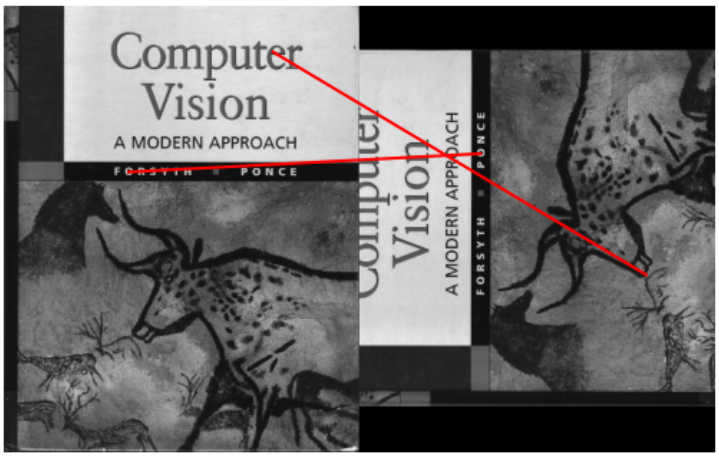
(because increasing *sigma* decreases corner detecting ability to begin with). On the other hand, increasing *ratio* increases the number of points matched.

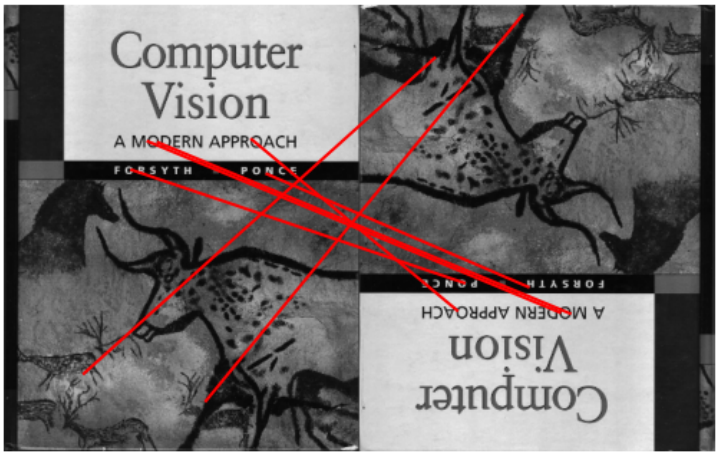
Q2.1.6

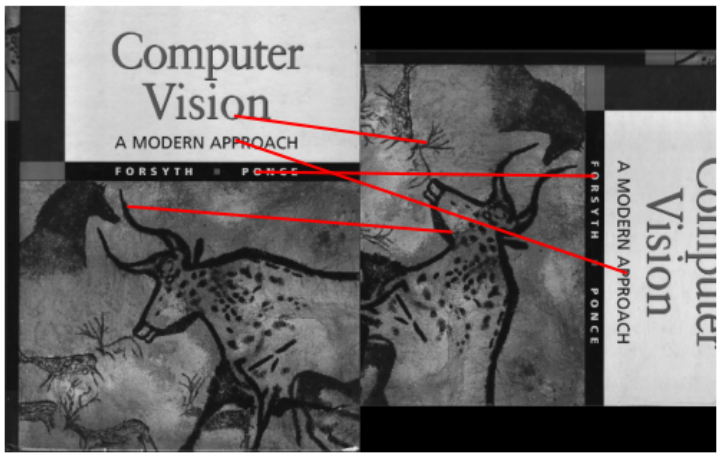
Here is the histogram (I don't know how to make the bins bigger):



And here are the matching thingies with three angles: $90°, 180°, 270°$ respectively:


Computer
Vision
A MODERN APPROACH
FORSYTH PONCE


Computer
Vision
A MODERN APPROACH
FORSYTH PONCE



As we can see, the BRIEF method deals very poorly with rotation. Even with a little rotation, the algorithm fails to be able to match corners on the same image very fast. This is because we create our binary descriptors by comparing patches with randomly chosen pixels. When we rotate the image, we obviously change the pixels/pixel intensities with which we compare the patch. This means that in each rotation, we get very different binary descriptors for the same patch, leading to difficulty in matching corners.

Here is the code for this: To generate the histogram in briefRotTest.py:

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

#Q2.1.6
def rotTest(opts):
    opts = get_opts()
    num_of_matches = np.zeros(36)
    #Read the image and convert to grayscale, if necessary
    image = cv2.imread('../data/cv_cover.jpg')
```

```
    for i in range(36):
        theta = i*10
        #Rotate Image
        image_rotated = scipy.ndimage.rotate(image, theta, reshape = False)
        #Compute features, descriptors and Match features
        matches, loc_1, loc_2 = matchPics(image, image_rotated, opts)
        num_of_matches[i] = len(matches)
    #Update histogram
    x_axis = np.arange(0, 360, 10)
    plt.bar(x_axis, num_of_matches)
    plt.xlabel("Rotation Angle in Degrees")
    plt.ylabel("Number of Matches")
    plt.title("Matches per theta of rotation")
    #Display histogram
    plt.show()
if __name__ == "__main__":

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

And in order to generate the three example images/matchings, we used the answer_generator.py file again:

```
image = cv2.imread('../data/cv_cover.jpg')
image_90 = scipy.ndimage.rotate(image, 90, reshape = False)
image_180 = scipy.ndimage.rotate(image, 180, reshape = False)
image_270 = scipy.ndimage.rotate(image, 270, reshape = False)
matches, loc1, loc2 = matchPics(image, image_90, opts)
plotMatches(image, image_90, matches, loc1, loc2)
matches, loc1, loc2 = matchPics(image, image_180, opts)
plotMatches(image, image_180, matches, loc1, loc2)
matches, loc1, loc2 = matchPics(image, image_270, opts)
plotMatches(image, image_270, matches, loc1, loc2)
```

Q2.1.7

I don't know if I'll have time to implement this (doubt it) but I would use a keypoint detector that can determine orientation, use that to undo the angle change on our keypoint pairs so that I'm undoing rotation, and then apply BRIEF to the images that are now properly aligned with each other.

5 2.2

Q2.2.1 Basically just used eigenvalue and SVD decompositions as suggested in section 1, where A is from our answer in 1.2.3 and implemented them here:

```
def computeH(x1, x2):
#Q2.2.1
#Compute the homography between two sets of points
n = x1.shape[0] #n is the number of point pairs
A = np.zeros((2*n, 9)) #2*d because for each point pair we get A_i with two rows
#9 because as we saw in section 1 h has 9 elts

for i in range(n):
    #A_2i and A_{2i+1} should be the matrix A_i from section 1 and I'm just plugging in what I got from there
    A[2*i] = [-1*x1[i, 0], -1*x1[i, 1], -1, 0, 0, 0, x1[i, 0]*x2[i, 0], x2[i, 0]*x1[i, 1], x2[i, 0]]
    A[2*i + 1] = [0, 0, 0, -1*x1[i, 0], -1*x1[i, 1], -1, x2[i, 0]*x1[i, 0], x2[i, 1]*x1[i, 0], x2[i, 1]]

#as suggested in the handout now we just do svd
U, Sigma , V_transpose = np.linalg.svd(A)
eigenvalues= Sigma[-1]
eigenvectors = V_transpose[-1:]/V_transpose[-1, -1] #The solution is the last column of V, i.e, the last row of V_transpose
#We divided by the last elememnt of the vector for normalization purposes.
H2to1 = eigenvectors.reshape(3, 3)
return H2to1
```

Q2.2.2

I feel like technically this is an affine transform, not a linear one as the origin is sent somewhere else. In any case, we basically subtract each point by the mean, and then scale down by an appropriate scalar (which is sqrt 2 times the largest distance).

We do this:

```
def computeH_norm(x1, x2):
    #Q2.2.2
    #Compute the centroid of the points
    cx_1 = np.mean(x1[:, 0])
    cy_1 = np.mean(x1[:, 1])
```

```
    cx_2 = np.mean(x2[:, 0])
    cy_2 = np.mean(x2[:, 1])

    #Shift the origin of the points to the centroid
    #Normalize the points so that the largest distance from the origin is equal to sqrt(2)
    #I don't see the point of actually finding the new points; it suffices for me to find the transformations instead, because that's what I need to return, so I'm instead going to directly compute the T_i
    magnitudes_1 = np.zeros(x1.shape[0])
    magnitudes_2 = np.zeros(x2.shape[0])
    for i in range(x1.shape[0]):
        magnitudes_1[i] = np.sqrt((x1[i, 0] - cx_1)**2 + (x1[i, 1] - cy_1)**2)
    for i in range(x2.shape[0]):
        magnitudes_2[i] = np.sqrt((x2[i, 0] - cx_2)**2 + (x2[i, 1] - cy_2)**2)

    max_1 = np.max(magnitudes_1)
    max_2 = np.max(magnitudes_2)
    #Scale everything down by max and then multiply by sqrt(2) and we good
    scale_1 = np.sqrt(2)/max_1
    scale_2 = np.sqrt(2)/max_2
    translation_1 = np.array([[1, 0, -cx_1], [0, 1, -cx_2], [0, 0, 1]])
    translation_2 = np.array([[1, 0, -cx_2], [0, 1, -cy_2], [0, 0, 1]])
    scale_1_matrix = np.array([[scale_1, 0, 0], [0, scale_1, 0], [0, 0, 1]])
    scale_2_matrix = np.array([[scale_2, 0, 0], [0, scale_2, 0], [0, 0, 1]])
    #Similarity transform 1
    T_1 = scale_1@translation_1
    x1new = np.hstack((x1, np.ones(x1.shape[0], 1)))
    x1new = T_1@x1.T
    #Similarity transform 2
    T_2 = scale_2@translation_2
    x2new = np.hstack((x2, np.ones(x2.shape[0], 1)))
    x2new = T_2@x2.T
    #Compute homography
    H2to1_oftransf = computeH(x1new, x2new)
    #Denormalization
    #Using the formula given
    H2to1 = np.linalg.inv(T_2)@(H2to1_oftransf@T_1)
```

```
41      return H2to1
```

Q2.2.2

Here is the code snippet (I excluded library imports, etc as it was already a bit long):

```
1   def computeH_norm(x1, x2):
2       #Q2.2.2
3       #Compute the centroid of the points
4       cx_1 = np.mean(x1[:, 0])
5       cy_1 = np.mean(x1[:, 1])
6       cx_2 = np.mean(x2[:, 0])
7       cy_2 = np.mean(x2[:, 1])
8
9       #Shift the origin of the points to the centroid
10      #Normalize the points so that the largest distance from the
        ↪  origin is equal to sqrt(2)
11      #I don't see the point of actually finding the new points;
        ↪  it suffices for me to find the transformations instead,
        ↪  because that's what I need to return, so I'm instead
        ↪  going to directly compute the T_i
12      magnitudes_1 = np.zeros(x1.shape[0])
13      magnitudes_2 = np.zeros(x2.shape[0])
14      for i in range(x1.shape[0]):
15          magnitudes_1[i] = np.sqrt((x1[i, 0] - cx_1)**2 + (x1[i,
            ↪  1] - cy_1)**2)
16      for i in range(x2.shape[0]):
17          magnitudes_2[i] = np.sqrt((x2[i, 0] - cx_2)**2 + (x2[i,
            ↪  1] - cy_2)**2)
18
19      max_1 = np.max(magnitudes_1)
20      max_2 = np.max(magnitudes_2)
21      #Scale everything down by max and then multiply by sqrt(2)
        ↪  and we good
22      scale_1 = np.sqrt(2)/max_1
23      scale_2 = np.sqrt(2)/max_2
24      translation_1 = np.array([[1, 0, -cx_1], [0, 1, -cy_1], [0,
        ↪  0, 1]])
25      translation_2 = np.array([[1, 0, -cx_2], [0, 1, -cy_2], [0,
        ↪  0, 1]])
26      scale_1_matrix = np.array([[scale_1, 0, 0], [0, scale_1,
        ↪  0], [0, 0, 1]])
27      scale_2_matrix = np.array([[scale_2, 0, 0], [0, scale_2,
        ↪  0], [0, 0, 1]])
28      #Similarity transform 1
```

```
    T_1 = scale_1_matrix@translation_1
    x1new = np.hstack((x1, np.ones((x1.shape[0], 1))))
    x1new = T_1@x1new.T
    #Similarity transform 2
    T_2 = scale_2_matrix@translation_2
    x2new = np.hstack((x2, np.ones((x2.shape[0], 1))))
    x2new = T_2@x2new.T
    #Compute homography
    H2to1_oftransf = computeH(x1new, x2new)
    #Denormalization
    #Using the formula given
    H2to1 = np.linalg.inv(T_2)@(H2to1_oftransf@T_1)
    return H2to1
```

Q2.2.3

Here is computeH_ransac

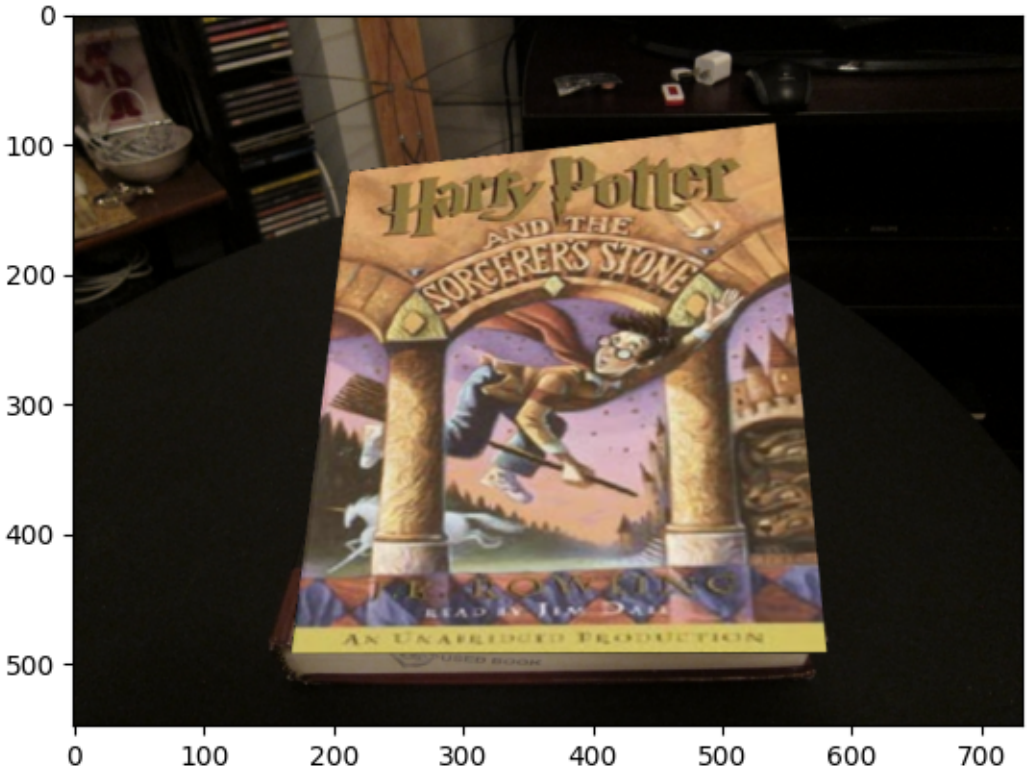
```
def computeH_ransac(locs1, locs2, opts):
    #Q2.2.3
    #Compute the best fitting homography given a list of
    ↪  matching points
    max\_iters = opts.max\_iters  # the number of iterations to
    ↪  run RANSAC for
    inlier\_tol = opts.inlier\_tol # the tolerance value for
    ↪  considering a point to be an inlier

    randompixels_1 = np.zeros((2, 4))
    randompixels_2 = np.zeros((2, 4))
    bestH2to1 = np.zeros((3, 3))
    x1 = locs1
    x2 = locs2
    x1_hom = np.hstack((x1, np.ones((x1.shape[0], 1))))
    x2_hom = np.hstack((x2, np.ones((x2.shape[0], 1))))
    inliers = -1
    for i in range(max\_iters):
        num_inliers = 0
        random_index = np.random.choice(locs1.shape[0], 4)
        randompixels_1 = locs1[random_index, :]
        randompixels_2 = locs2[random_index, :]
        H_norm = computeH_norm(randompixels_1, randompixels_2)

        for j in range(x2_hom.shape[0]):
            pred_x2 = H_norm@x1_hom[j].T
            pred_x2[0] = pred_x2[0]/pred_x2[2]
            pred_x2[1] = pred_x2[1]/pred_x2[2]
```

```
            err_1 = (x2_hom[j][0] - pred_x2[0])
            err_2 = (x2_hom[j][1] - pred_x2[1])
            err = [err_1, err_2]
            error = np.linalg.norm(err)
            if error <= inlier\_tol:
                num_inliers+=1

        if num_inliers > inliers:
            bestH2to1 = H_norm
            inliers = num_inliers
    return bestH2to1, inliers
```

Q2.2.4

In answer to step 4, basically we need to reshape the harry potter cover to fit "snugly" on top of the book cover.

Here is the result:



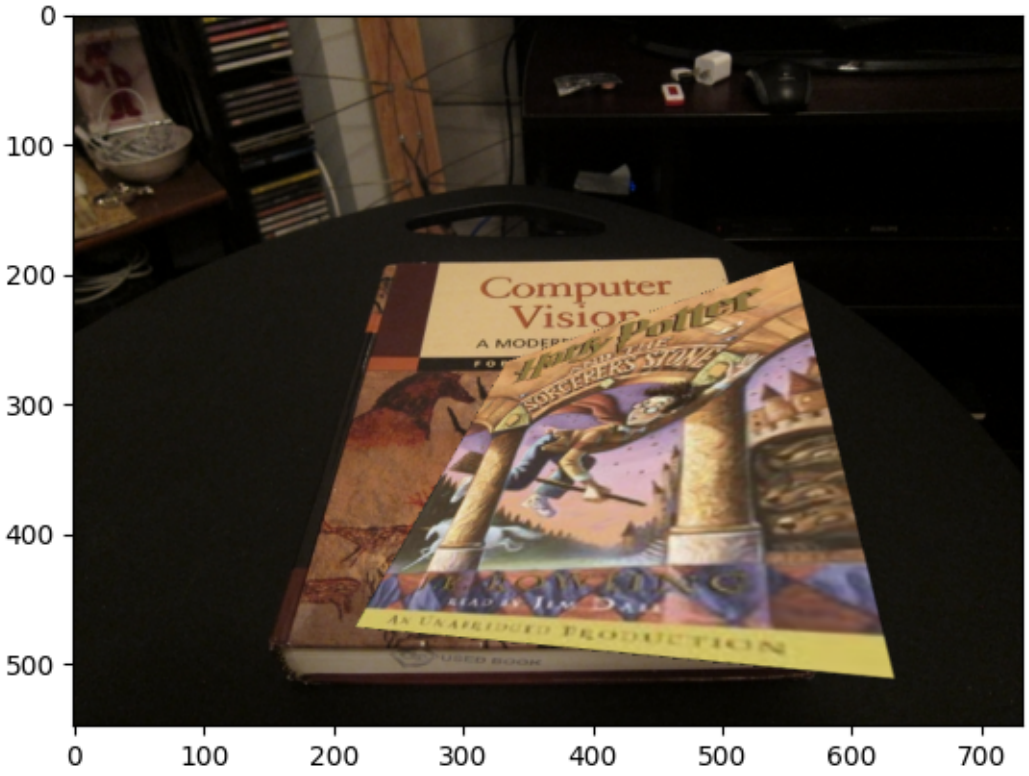
And here is the code:

```
# Import necessary functions
import matplotlib.pyplot as plt
from planarH import compositeH
from planarH import computeH_ransac
from matchPics import matchPics
# Q2.2.4

def warpImage(opts):
    cover = cv2.imread('../data/cv_cover.jpg')
    harry_potter = cv2.imread('../data/hp_cover.jpg')
    desk = cv2.imread('../data/cv_desk.png')
    matches, loc1, loc2 = matchPics(cover, desk, opts)

    #In answer to step 4, we need the harry potter cover to appear as the same size and perspective as the book cover.
    #This can be accomplished with an easy resize
    harry_potter_correct = cv2.resize(harry_potter, (cover.shape[1], cover.shape[0]))
    #Isolating the matches from the locations
    loc1 = loc1[matches[:, 0], 0:2]
    loc2 = loc2[matches[:, 1], 0:2]
    bestH2to1, inliers = computeH_ransac(loc1, loc2, opts)
    composite_img = compositeH(bestH2to1, harry_potter_correct, desk)
    plt.imshow(composite_img)
    plt.show()
```

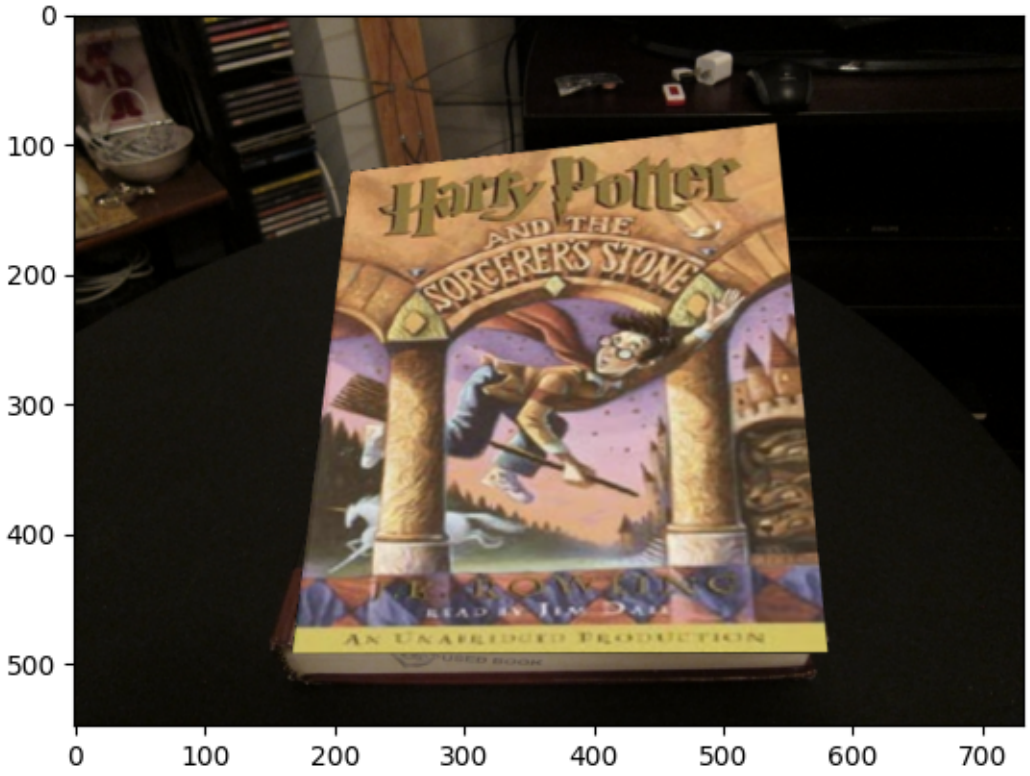
Q2.2.5

Let us pick max_iters from $\{250, 500, 750\}$ and $inlier\_tol$ from $\{1, 2, 3\}$. This gives us 9 possibilities:

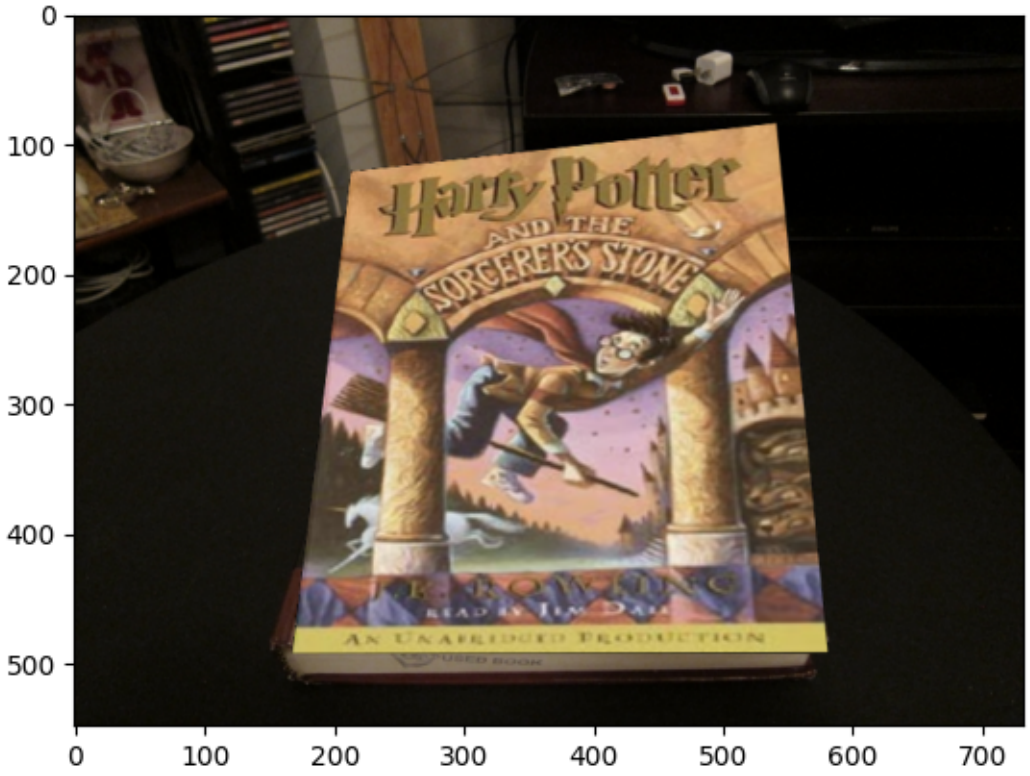
$max\_iter = 250$, $inlier\_tol = 1$:



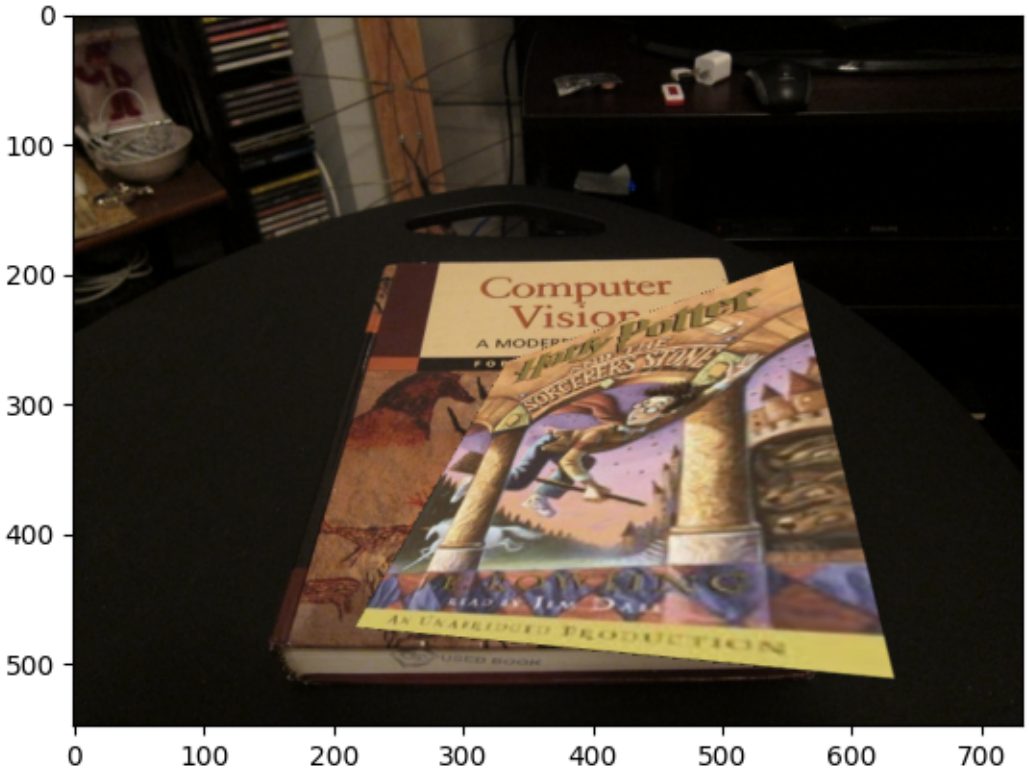
$max\_iter = 250$, $inlier\_tol = 2$:



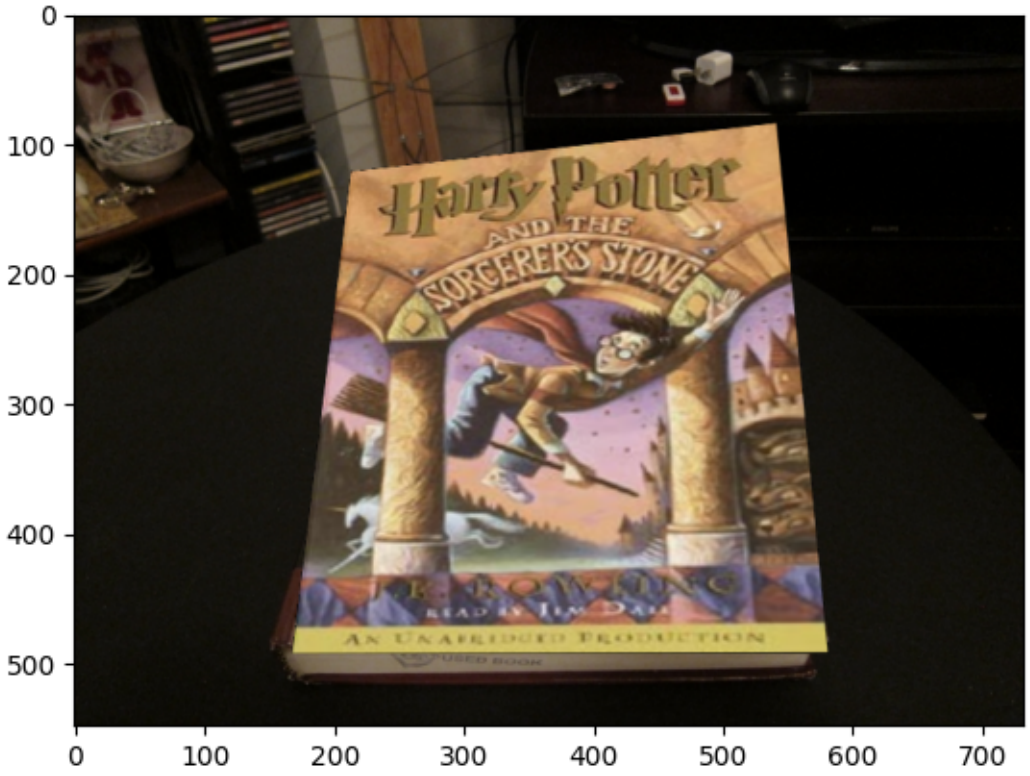
$max\_iter = 250$, $inlier\_tol = 3$:



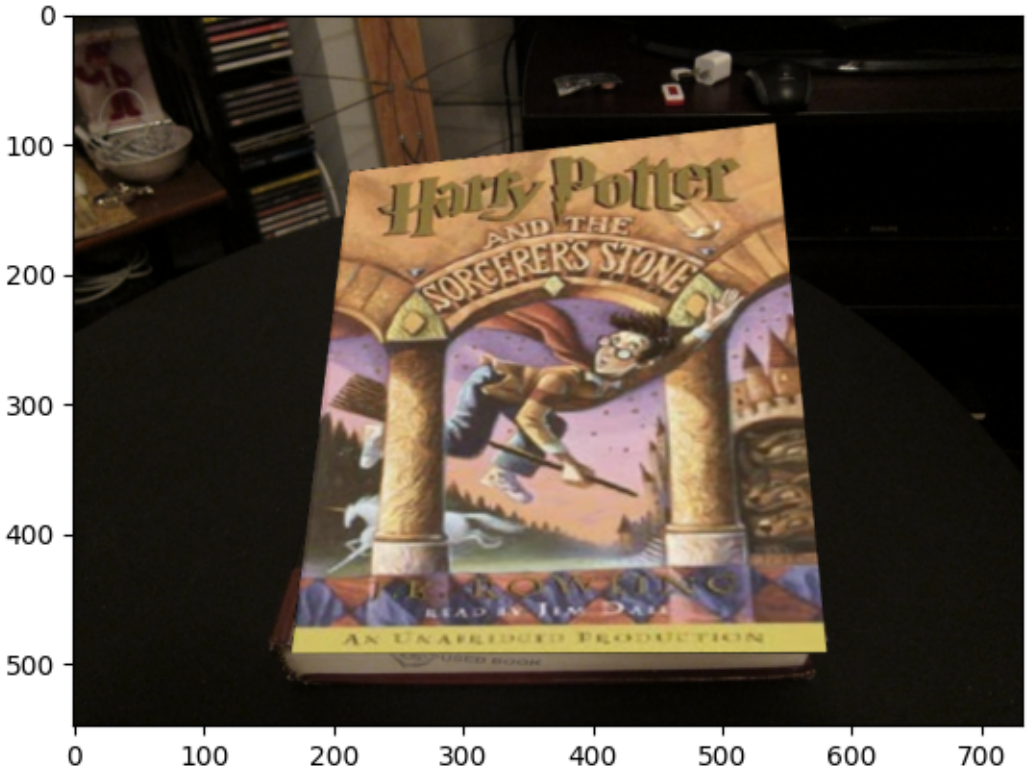
$max\_iter = 500$, $inlier\_tol = 1$



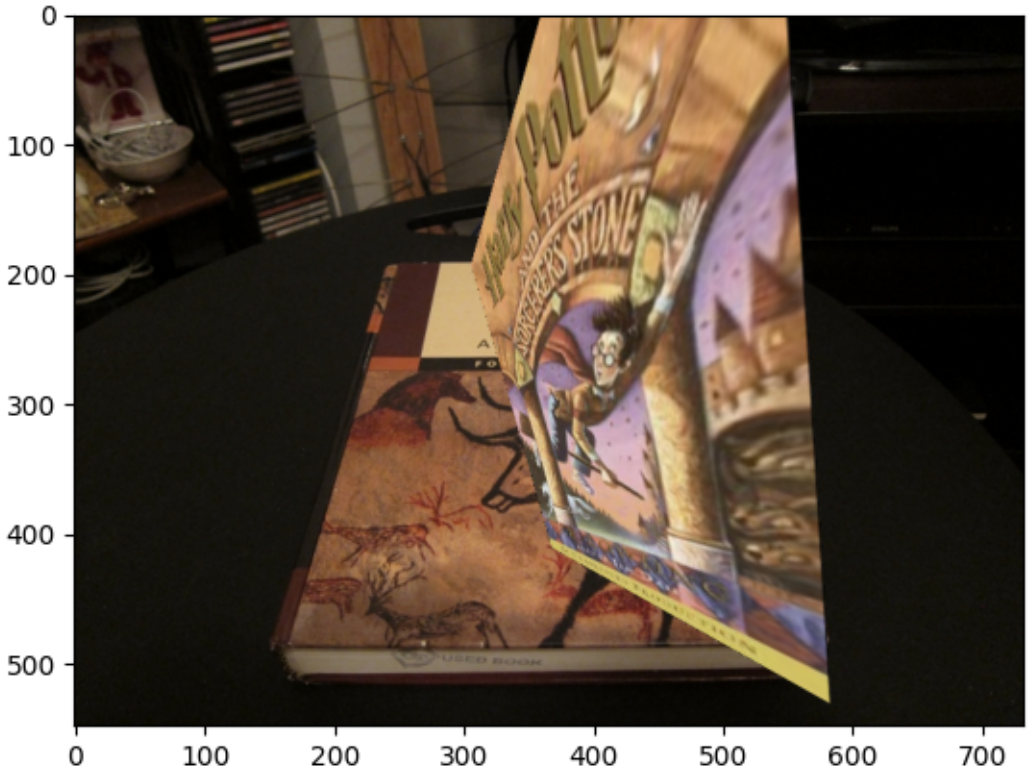
$max\_iter = 500$, $inlier\_tol = 2$



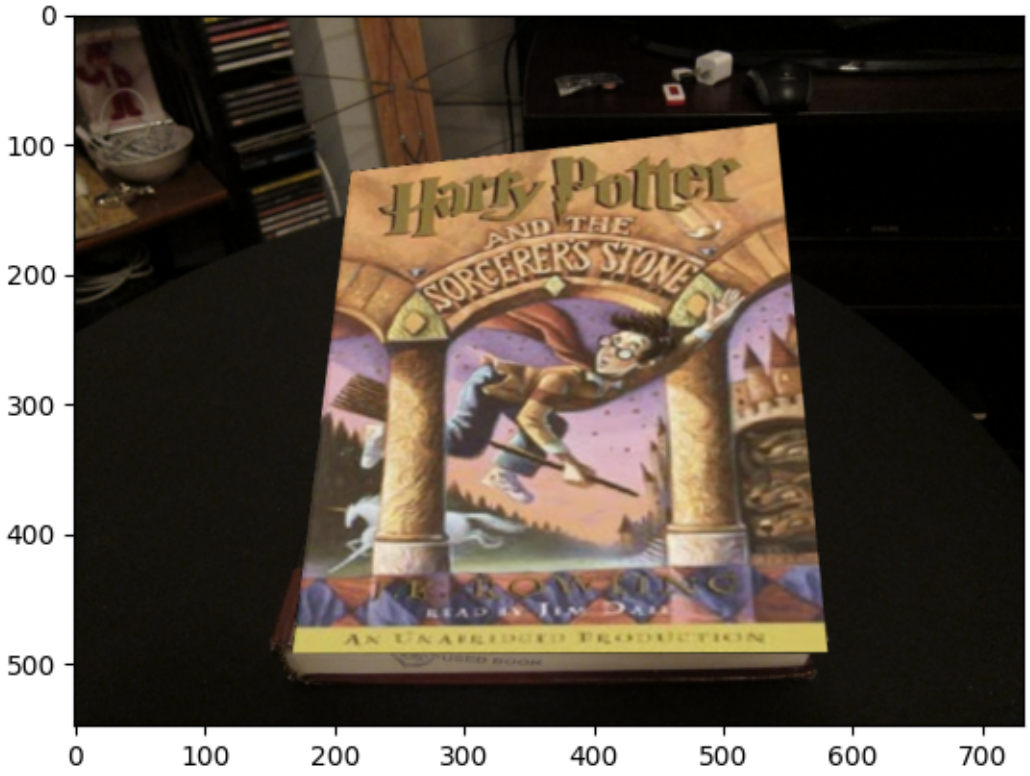
$max\_iter = 500$, $inlier\_tol = 3$



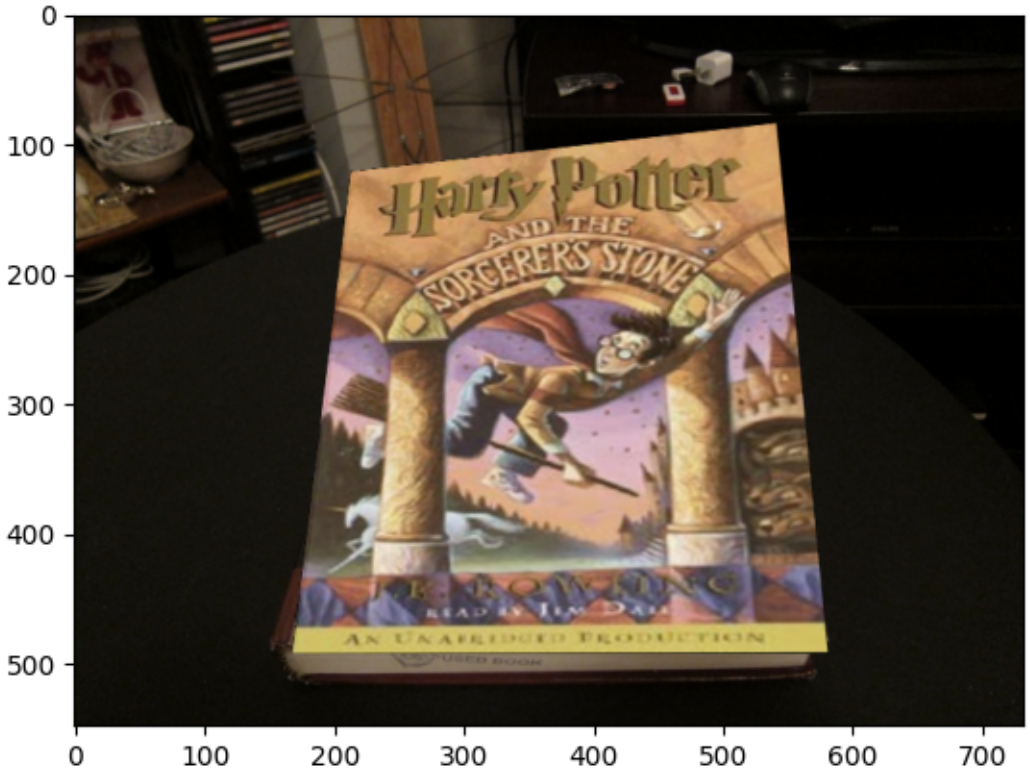
$max\_iter = 750$, $inlier\_tol = 1$



$max\_iter = 750$, $inlier\_tol = 2$



$max\_iter = 750$, $inlier\_tol = 3$



It seems the tolerance has the biggest effect on the quality with lower tolerance leading to worse results. Higher iterations typically leads to better results.

6 3

Sounds super fun but unfortunately didn't have the time to run code for several hours and I just can't get multiprocessing to work. Hopefully the extra credit questions cover what I lost here.

7 4

Here's the code:

```
1  import numpy as np
2  import cv2
```

```
3
4   # Import necessary functions
5   import matplotlib.pyplot as plt
6   from planarH import compositeH
7   from planarH import computeH_ransac
8   from matchPics import matchPics
9   from displayMatch import displayMatched
10  import skimage.io
11  import skimage.color
12  import scipy
13  from opts import get_opts
14
15  opts = get_opts()
16  left = cv2.imread("../data/pano_left_mine.jpg")
17  right = cv2.imread("../data/pano_right_mine.jpg")
18  left = cv2.resize(left, (1457, 1080))
19  right = cv2.resize(right, (1457, 1080))
20  #I figured out the padding basically by trial and error
21  right = cv2.copyMakeBorder(right, right.shape[1] -
    ↪  left.shape[1], 0, int(0.6*left.shape[1]), 0,
    ↪  cv2.BORDER_CONSTANT, value = 0)
22  matches, loc1, loc2 = matchPics(left, right, opts)
23  #displayMatched(opts, left, right)
24  loc1 = loc1[matches[:, 0], 0:2]
25  loc2 = loc2[matches[:, 1], 0:2]
26  bestH2to1, inliers = computeH_ransac(loc1, loc2, opts)
27  composite_img = compositeH(bestH2to1, left, right)
28  plt.imshow(composite_img)
29  plt.show()
```

Here are the left and right images I used respectively:

And we got:

I could probably have chosen better padding to reduce blackspace and have the fit be less angular but I was running out of time.