CV HW1

Lifan Yu (lifany)

September 2023

Question 1

1.1

For a real world point, suppose it has coordinate $\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$

Projection from 3D to 2D can be expressed as

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f, & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Where the r's are the rotations and t's are the translations. the matrix with f's is the projection onto camera plane.

On the plane Π, we can express the above projection as

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} h11 & h12 & h13 & h14 \\ h21 & h22 & h23 & h24 \\ h31 & h32 & h33 & h34 \end{bmatrix} \begin{bmatrix} X_\Pi \\ Y_\Pi \\ Z \\ 1 \end{bmatrix}$$

Plug in $Z = 0$

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = H' \begin{bmatrix} X_\Pi \\ Y_\Pi \\ 1 \end{bmatrix}$$

Here H' is 3 by 3. The same x_Π maps o two camera image points, therefore,

$$\lambda_1 \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = H_1 \begin{bmatrix} X_\Pi \\ Y_\Pi \\ 1 \end{bmatrix}$$

$$\lambda_2 \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = H_2 \begin{bmatrix} X_\Pi \\ Y_\Pi \\ 1 \end{bmatrix}$$

Combining the above expressions gives

$$\lambda_1 H_1^{-1} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \lambda_2 H_2^{-1} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \frac{\lambda_2}{\lambda_1} H_1 H_2^{-1} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

Let

$$H = \frac{\lambda_2}{\lambda_1} P_1 P_2^{-1}$$

and here H is a 3 by 3 matrix.
Thus there exists a 3 by 3 matrix H so that

$$x1 \equiv H x_2$$

1.2

-

(1) 8 (9 variables to solve, but with 1 constant scaling)
(2) 4 (1 point pair produces 2 equations)
(3)

$$\lambda \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

$$x_2 = \frac{\lambda x_2}{\lambda} = \frac{ax_1 + by_1 + c}{gx_1 + hy_1 + i}$$

$$y_2 = \frac{\lambda y_2}{\lambda} = \frac{dx_1 + ey_1 + f}{gx_1 + hy_1 + i}$$

For $x_1^i = [x_1^i, y_1^i, 1]$, the equations can be rearranged into

$$\begin{bmatrix} x_2^i & y_2^i & 1 & 0 & 0 & 0 & -x_1^i x_2^i & -x_1^i y_2^i & -x_1^i \\ 0 & 0 & 0 & x_2^i & y_2^i & 1 & -y_1^i x_2^i & -y_1^i y_2^i & -y_1^i \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

This can be denoted simply as

$$A_i h = 0$$

Where

$$A_i = \begin{bmatrix} x_2^i & y_2^i & 1 & 0 & 0 & 0 & -x_1^i x_2^i & -x_1^i y_2^i & -x_1^i \\ 0 & 0 & 0 & x_2^i & y_2^i & 1 & -y_1^i x_2^i & -y_1^i y_2^i & -y_1^i \end{bmatrix}$$

(4) Trivial solution $h = 0$.

A (n by n matrix) is not full rank because there is some non-trivial solution to Ah = 0, so its null space is not empty, and that Rank(A) + Dim(Null(A)) = n, therefore rank < n.

Its influence on singular values is that some singular values will be 0.

1.4.1

We are given

$$x_1 = K_1[I \;\; 0]X$$
$$x_2 = K_2[R \;\; 0]X$$

x_2 can be written as

$$x_2 = K_2R[I \;\; 0]X$$

Substitute $K_1^{-1}x_1 = [I \;\; 0]X_1$ into the above, we get

$$x_2 = K_2RK_1^{-1}x_1$$

Let $H = K_2RK_1^{-1}$, we then have

$$x \equiv Hx_2$$

1.4.2

$$x_1 = H^2 X_2 = h(HX_2)$$

Let

$$X' = HX_2$$

, meaning, rotating x_2 by θ gives us X'

We now have $x_1 = HX'$, meaning rotating X' by θ gives X_1, then X_1 is obtained by rotating X_2 by 2θ

1.4.3

Only when the scene is planar can the planar homography work well, that is, when objects are not too close to the camera and do not have a significant scene depth differences among them, and under this condition the objects can be considered to be on the same plane

With significant scene depth difference in objects close to the camera, planar homography works well only under pure rotation

1.4.4

Suppose we have the following line in 3D space

$$X = X_1 + (1-\lambda)(X_2 - X_1), \quad \lambda \in \mathbb{R}, X_i \in \mathbb{R}^3$$

$$X = (1+\lambda)\begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \end{bmatrix} - \lambda \begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \end{bmatrix} = \begin{bmatrix} (X_1 - X_2)\lambda + X_1 \\ (Y_1 - Y_2)\lambda + Y_1 \\ (Z_1 - Z_2)\lambda + Z_2 \end{bmatrix}$$

We have a projection matrix P that projects the 3D space points X_i onto 2D space points x_i

$$x = PX = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} X = \begin{bmatrix} fX \\ fY \\ Z \end{bmatrix} = \begin{bmatrix} f\frac{X}{Z} \\ f\frac{Y}{Z} \\ 1 \end{bmatrix}$$

For any two different points $M_1(X_1, Y_1, Z_1)$, $M_2(X_2, Y_2, Z_2) \in \mathbb{R}^3$ on the line represented by X, we have

$$M_1 = X_1 + (1-\lambda_1)(X_2 - X_1)$$
$$M_2 = X_2 + (1-\lambda_2)(X_2 - X_1)$$

Their projections onto a 2D plane is $m_1(x_1, y_1), m_2(x_2, y_2) \in \mathbb{R}^2$ expressed by

$$m_1 = PM_1, m_2 = PM_2$$

$$\frac{X_2 - X_1}{Y_2 - Y_1} = \frac{f\frac{X_2}{Z_1} - f\frac{X_1}{Z_1}}{f\frac{Y_2}{Z_2} - f\frac{Y_1}{Z_2}} = \lambda' \frac{(X_1 - X_2)\lambda_2 + X_1 - [(X_1 - X_2)\lambda_1 + X_1)]}{(Y_1 - Y_2)\lambda_2 + Y_1 - [(Y_1 - Y_2)\lambda_2 + Y_1)]} = \frac{(X_1 - X_2)(\lambda_2 - \lambda_1)}{(Y_1 - Y_2)(\lambda_2 - \lambda_1)}$$

Because the two points are different, $\lambda_2 \neq \lambda_1$, we get

$$\frac{X_1 - X_2}{Y_1 - Y_2}$$

Because $X_1 \neq X_2$ and $Y_1 \neq Y_2$, the result is a constant real number. It satisfies the definition of a 2D line that for any 2 points $(x_1, y_1), (x)2, y_2)$, the slope is a constant. Thus is preserves lines by projecting a 3D line to a 2D line.

Question 2

2.1.1 FAST Detector

Comparison:

FAST detector looks at the brightness of the point of interest's surrounding points (a circle of 16 pixels). While for Harris corner detector, it takes a small window of pixels, shift it by small amounts, and focus on the brightness difference in sum of squared difference for each pixel.

To make the algorithm fast, FAST checks four points outside the circle for their brightness. There needs to be at least 3 points with brightness level significantly different from that of the point of interest in order for a further detection of **n** contiguous darker/lighter pixels in the circle, n usually set to 12. While for Harris corner detector, it tries to find windows where the SSD is significant in all directions.

Disadvantages:

1. FAST then has a rather strict requirement that the corner needs to be less than a quarter big int he circle, which might not work well in a lot of cases such as a very perfect corner which takes up exactly a quarter of the circle, so it might miss some specific types of corners detected by Harris corner detector.

2. FAST has a parameter n (for contiguous brighter/darker pixels) which creates inconvenience. When n is too small, there might be a lot more points of interest detected than necessary.

3. Additionally, FAST might not work well when noises exist, because noises will destroy original areas of contiguous pixels with the same intensity, while Harris corner detector can perform better in noised images.

Advantage:

1. An advantage of FAST over Harris might be that it considers the precise and contiguous surroundings of a point, thus can more accurately and precisely identify corners (or identify more corners) than Harris.

2.1.1 BRIEF Descriptor

BRIEF descriptor is a binary descriptor that describes the image by sampling point pairs and computing their intensity difference. These randomly sampled points are initialized only once and used for each image and patch. For each point pair, if the first pixel's intensity is larger than the second, it puts a 1 in the string of the descriptor, else it puts a 0. The distance between two binary descriptors is the number of different bits. The comparison result is recorded as either 1 or 0.

Filter bank filters out desired features, thus can denoise, blur or enhance images, and yes, filter banks can be used to create descriptors. For example, the GIST descriptor is computed by convolving the image with Gabor filters, and producing feature maps, and using the averaged values of feature maps make up the descriptor.

2.1.3 Matching Methods

What's Hamming Distance:

Hamming distance is a metric to compare two binary data strings. If those strings have equal length, Hamming distance is the number of bit positions in which the two bits are different.

Hamming Distance in BRIEF:

BRIEF selects a set of n points (pi, qi) to compare pairwise intensity level. if I(p) ¡ I(q), the result is 1, otherwise 0. Applying it on the n pairs of poits gives us an n dimensional bitstring. Now we can use Hamming distance to easily get the descriptors.

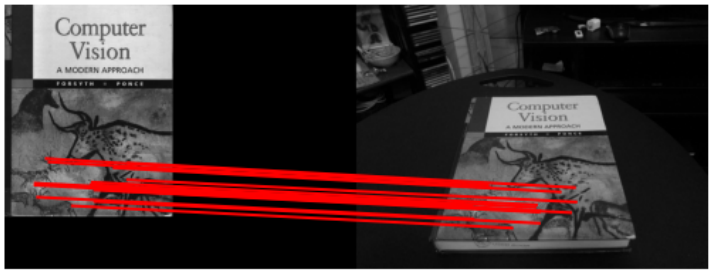
Benefits:

Hamming distance makes computations faster because we only need to apply XOR and bit count to find the distance. These are very fast with the computer hardware and instructions we have. Euclidean distance will be much more computationally expensive by taking original numbers, squaring the difference and then the square root, and requires more storage space.

2.1.4 Feature Matching

```
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
        I1_corners = corner_detection(I1, sigma)
        I2_corners = corner_detection(I2, sigma)

        # TODO: Obtain descriptors for the computed feature
                                            locations
        I1_descriptor, locs1 = computeBrief(I1, I1_corners)
        I2_descriptor, locs2 = computeBrief(I2, I2_corners)

        # TODO: Match features using the descriptors
        matches = briefMatch(I1_descriptor, I2_descriptor, ratio)
        print("Number of matches: ", len(matches))

        return matches, locs1, locs2
```

The best match image from the above


Computer
Vision
A MODERN APPROACH
FORSYTH • PONCE
Computer
Vision

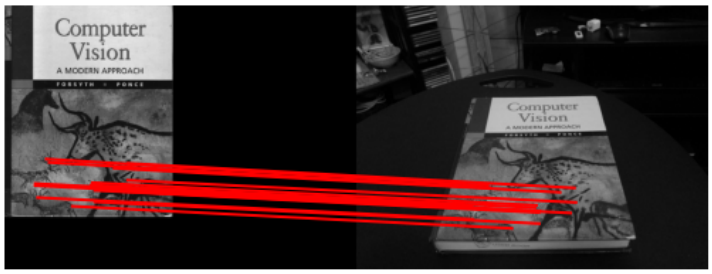
2.1.5 Feature Matching and Parameter Tuning

Effect of sigma and ratio

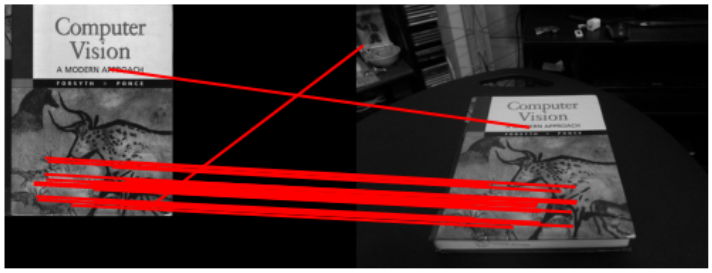
Sigma is associated with the corner detector, The higher the sigma, the less corners it detects as there needs to more significant gradient difference between the points.

The higher the ratio, the less strict it is when filtering the matched pairs, and more prone to errors, but if ratio is too small, there might not be enough correct matches.

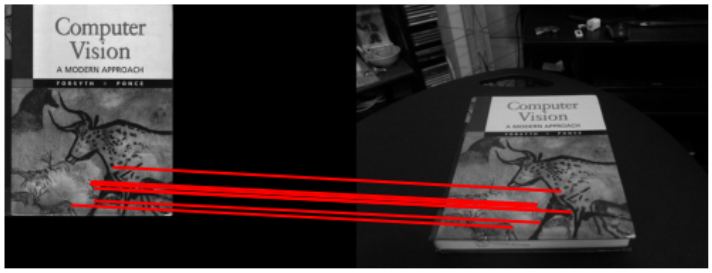
Best parameters: –sigma 0.1 –ratio 0.55



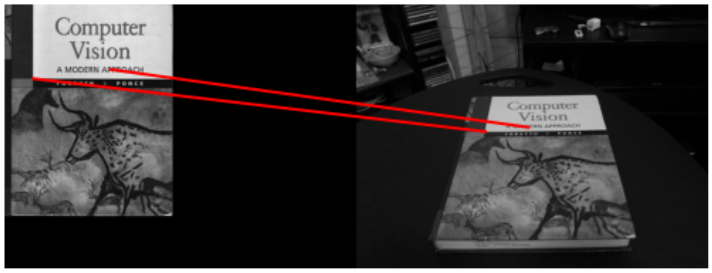
Increasing ratio: –sigma 0.1 –ratio 0.6, more errors



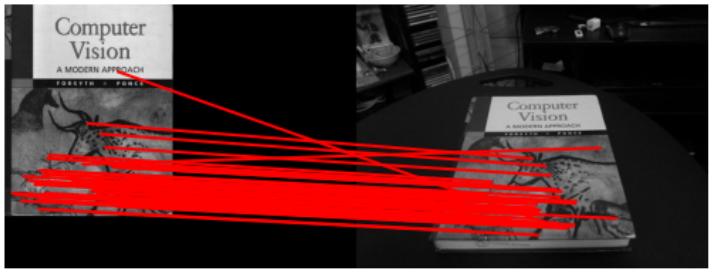
Decreasing ratio: –sigma 0.1 –ratio 0.45, too few matches



Increasing sigma: –sigma 0.3 –ratio 0.6



Decreasing sigma: –sigma 0.04 –ratio 0.6



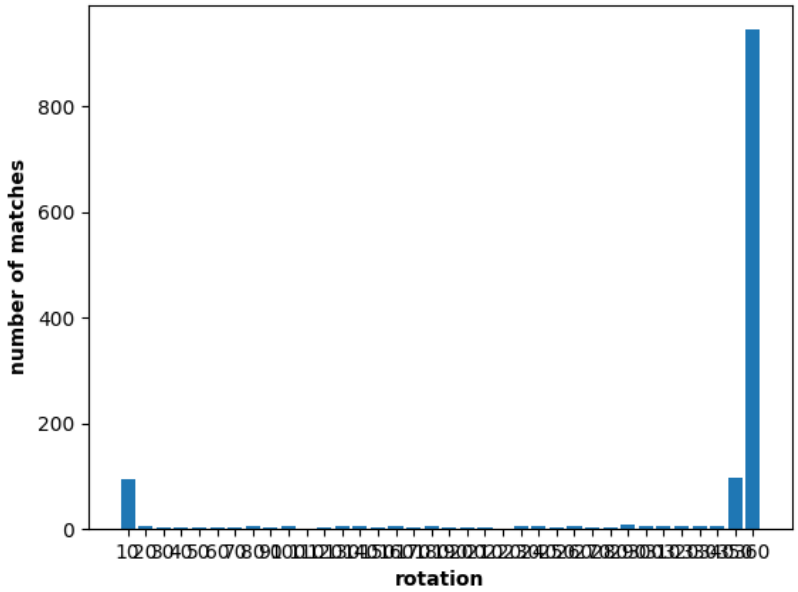
2.1.6 BRIEF and Rotations

BRIEF descriptors calculate intensity differences from selected points, and even when we use the same image, when the orientation changes significantly, the intensity differences in all directions change significantly as well. Therefore there will be much fewer matches.

When the image rotates back to its original orientation, the selected points match exactly as intensity differences are the same in all directions.

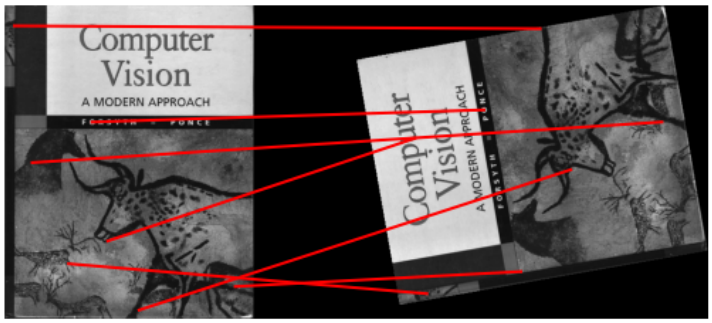
```
import numpy as np
import cv2
from matchPics import matchPics
from opts import get_opts

from scipy import ndimage, misc
import matplotlib.pyplot as plt

#Q2.1.6

def rotTest(opts):

    # TODO: Read the image and convert to grayscale, if necessary
    I = cv2.imread('../data/cv_cover.jpg')
    degrees = []
    counts = []

    for i in range(36):

        # TODO: Rotate Image
        d = 10*i + 10
        print("rotation: ", d)
        I_rotate = ndimage.rotate(I, d, mode = 'constant')

        # TODO: Compute features, descriptors and Match features
        matches, locs1, locs2 = matchPics(I, I_rotate, opts)
        match_count = len(matches)

        # TODO: Update histogram
        degrees.append(d)
        counts.append(match_count)

    # TODO: Display histogram
    fig, ax = plt.subplots()
    ax.set_xlabel('rotation', fontweight ='bold')
    ax.set_ylabel('number of matches', fontweight ='bold')
    ax.bar([str(item) for item in degrees], counts)
    plt.savefig("2_1_6_result.png")
    plt.show()

if __name__ == "__main__":

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

BRIEF rotation test result

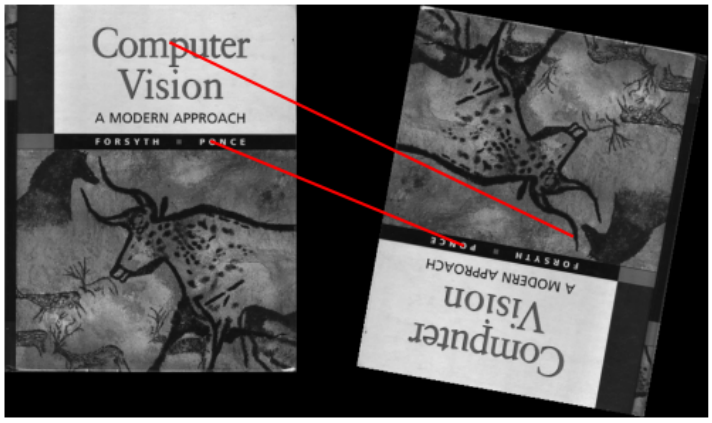


10 degrees rotation

100 degrees rotation



180 degrees rotation



2.1.7 Extra Credit - Improving Performance

(1) The algorithm of my solution is as follows:

a. Read Image1, Image 2

b. Rotate Image2 from 0 degree to 360 degrees with a predefined (tunable) step value, record the number of matches for each rotation, and the corresponding rotation degree

c. Identify the degree with the highest match counts, compute a ratio by dividing the maximum count with the average counts, see whether or not the count is significant enough to deduce that the images might match. This ratio is also a tunable parameter.

d. In the degree value intervals to the left and right of this rotational degree, try with smaller steps to identify the rotation angel that produces the highest number of matches

e. When visualizing, rotate the matched points in Image2 to the original points before the rotation, while padding Image1 to the size of original Image2 before rotation

Command:

```
!python briefRotTest.py --sigma 0.1 --ratio 0.55 --invariant 1
                                    --rot_thresh 1.5 --step 30
```

The algorithm is implemented in matchPics.py as shown below

```
# -- rotation invariant matchPics
import numpy as np
import cv2
import skimage.color
from helper import briefMatch
from helper import computeBrief
from helper import corner_detection

from scipy import ndimage
import math
from helper import *

# 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
        I1_corners = corner_detection(I1, sigma)
        I2_corners = corner_detection(I2, sigma)

        # TODO: Obtain descriptors for the computed feature
                                    locations
        I1_descriptor, locs1 = computeBrief(I1, I1_corners)
        I2_descriptor, locs2 = computeBrief(I2, I2_corners)

        # TODO: Match features using the descriptors
        matches = briefMatch(I1_descriptor, I2_descriptor, ratio)
        print("Number of matches: ", len(matches))

        return matches, locs1, locs2

# -- rotation invariant matchPics
def matchPics_rotinvariant(I1, I2, opts):

        if (opts.invariant is None) or (opts.invariant == 0):
          matches, locs1, locs2 = matchPics(I1, I2, opts)
        # -- if we require rotation invariance
        elif opts.invariant == 1:
          # -- rot_thresh: how much shoud the max count differ from
                                      the mode to be
                                    considered
          rot_thresh = opts.rot_thresh
          step = opts.step
          # -- step through different rotations to look for a best
                                    precise range
          degrees, counts = rotation_match(I1, I2, 0, 360, step,
                                    opts)
          m = np.average(counts)
          # -- if there exists a rotation which produce a
                                    significant number of
                                    matches
          if (max(counts) / m) > rot_thresh:
            # -- identify a precise range
            target_idx = counts.index(max(counts))
            if target_idx == len(counts) - 1:
                target_idx1 = target_idx - 1
                target_idx2 = 0
            else:
                target_idx1 = target_idx - 1
                target_idx2 = target_idx + 1

            # -- find the best rotation in the precise range
            #d1 = min(degrees[target_idx1], degrees[target_idx2])
            #d2 = max(degrees[target_idx1], degrees[target_idx2])
```

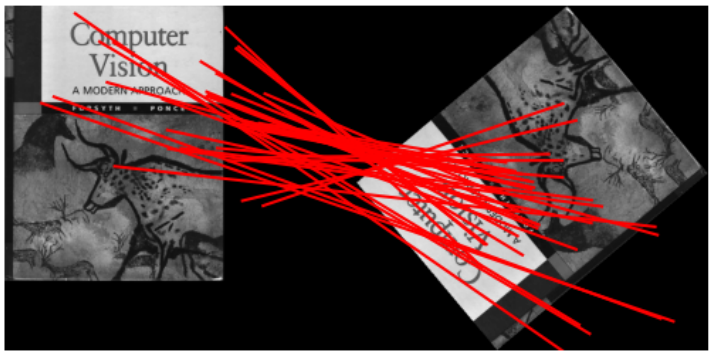
```
  d1 = degrees[target_idx1]
  d2 = degrees[target_idx2]
  if d2>d1:
    degrees2, counts2 = rotation_match(I1, I2, d1, d2+1,
                                       60, opts)
  else:
    degrees2, counts2 = rotation_match(I1, I2, d1, d2-1,
                                       -20, opts)
  best_degree = degrees2[counts2.index(max(counts2))]

# -- find the best rotation degree
I_rbest = ndimage.rotate(I2, best_degree, mode = '
                                     constant')
print("best rotation degree ", str(best_degree))
matches, locs1, locs2 = matchPics(I1, I_rbest, opts)
locs1[:, 0], locs1[:, 1] = locs1[:, 1], locs1[:, 0].copy
                                     ()
locs2[:, 0], locs2[:, 1] = locs2[:, 1], locs2[:, 0].copy
                                     ()

# -- find the original loc2 from I2 before rotation
new_w = I_rbest.shape[1]
new_h = I_rbest.shape[0]

ox, oy = int(0.5*new_w), int(0.5*new_h)

original_loc2 = []

for point in locs2:
  px = point[0]
  py = point[1]
  qx = ox + math.cos(-best_degree) * (px - ox) - math.sin
                                      (-best_degree) * (py
                                      - oy)
  qy = oy + math.sin(-best_degree) * (px - ox) + math.cos
                                      (-best_degree) * (py
                                      - oy)
  x_scale = I2.shape[1] / new_w
  y_scale = I2.shape[0] / new_h
  original_loc2.append([int(qx*x_scale), int(qy*y_scale)]
                                     )

I1_padding = np.zeros((I2.shape[0], I2.shape[1], 3))
centerx = int(0.5*(I2.shape[0]))
centery = int(0.5*(I2.shape[1]))
x = int(0.5*(I1.shape[0]))
y = int(0.5*(I1.shape[1]))
print(x, y, centerx, centery)
for i in range(I1.shape[0]):
  for j in range(I1.shape[1]):
    distx = i - x
    disty = j - y
    I1_padding[centerx+distx][centery+disty] = I1[i][j]
cv2.imwrite("I_padding.jpg", I1_padding)

I_padding = cv2.imread("I_padding.jpg")
plotMatches(I1, I2, matches, locs1, np.array(
```

```
                                                original_loc2), "rot-
                                                invariant.png")

        return matches, locs1, locs2

# -- perform rotation and count matches for each rotation degree
def rotation_match(I1, I2, start, end, step, opts):
        degrees = []
        counts = []
        for d in range(start, end, step):
          print("rotation (invariant ver.): ", d)
          I2_rotate = ndimage.rotate(I2, d, mode = 'constant')

          # TODO: Compute features, descriptors and Match features
          matches, locs1, locs2 = matchPics(I1, I2_rotate, opts)
          match_count = len(matches)

          # TODO: Update histogram
          degrees.append(d)
          counts.append(match_count)
        return degrees, counts
```

Tested with the below code in briefRotTest.py

```
opts = get_opts()
 I = cv2.imread('../data/cv_cover.jpg')
 I_rotate = ndimage.rotate(I, 130, mode = 'constant')
 matches, locs1, locs2 = matchPics_rotinvariant(I, I_rotate,
                                    opts)
```

The rotation invariant matching:



2.2.1 Computing the Homography

```
def computeH(x1, x2):
    #Q2.2.1
    # TODO: Compute the homography between two sets of points
    # -- x_1i = Hx_2i, A_1h = 0, derive A which is (N*2) rows * 9
                                        columns
    # -- here x_1 and x_2 are locs1 and locs2 after the match
    A = []
    for i in range(len(x1)):
      # -- append the two rows per x_2i
      A.append([x2[i][0], x2[i][1], 1, 0, 0, 0, -x2[i][0]*x1[i][0],
                                          -x2[i][1]*x1[i][0], -x1[i]
                                         [0]])
      A.append([0, 0, 0, x2[i][0], x2[i][1], 1, -x2[i][0]*x1[i][1],
                                          -x2[i][1]*x1[i][1], -x1[i]
                                         [1]])
    A = np.matrix(np.array(A))
    # -- h is the corresponding eigen-vector (column 9 of V) to the
                                         smallest eigenvalue of (A^T)
                                        A
    U, Sigma, VT = np.linalg.svd(A)
    V = VT.T
    h = V[:, -1]
    H2to1 = np.reshape(h, (3,3))
    return H2to1
```

2.2.2 Homography Normallization

```
def computeH_norm(x1, x2):
    #Q2.2.2
    # TODO: Compute the centroid of the points
    c1 = np.mean(x1, axis = 0)
    c2 = np.mean(x2, axis = 0)

    # TODO: Shift the origin of the points to the centroid
    # -- subtract the points with the centroid
    x1_shifted = x1 - c1
    x2_shifted = x2 - c2

    # TODO: Normalize the points so that the largest distance from
                                    the origin is equal to sqrt(2
                                    )
    x1_shifted_norm = np.linalg.norm(x1_shifted, axis = 1)
    x2_shifted_norm = np.linalg.norm(x2_shifted, axis = 1)
    max_norm1 = np.max(x1_shifted_norm)
    max_norm2 = np.max(x2_shifted_norm)
    scale1 = np.sqrt(2) / max_norm1
    scale2 = np.sqrt(2) / max_norm2
    x1_normalized = x1_shifted * scale1
    x2_normalized = x2_shifted * scale2

    # TODO: Similarity transform 1
    # -- T 3 by 3, with normalization and shifting
    T1 = np.array([
      [scale1, 0, -scale1*c1[0]],
      [0, scale1, -scale1*c1[1]],
      [0, 0, 1]
    ])

    # TODO: Similarity transform 2
    T2 = np.array([
      [scale2, 0, -scale2*c2[0]],
      [0, scale2, -scale2*c2[1]],
      [0, 0, 1]
    ])

    # TODO: Compute homography
    H = computeH(x1_normalized, x2_normalized)

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

2.2.3 Implement 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

    matches_count = len(locs1)
    inlier_count = 0
    # -- idx of best case inliers
    inliers = []
    bestH2to1 = 0

    for i in range(max_iters):
      # -- because we need a minimum of 4 points, we randomly
                                          select 4 indices
      selected_idx = random.sample(range(len(locs1)), 4)
      selected_points1 = np.array([locs1[i] for i in selected_idx])
      selected_points2 = np.array([locs2[i] for i in selected_idx])
      # -- exact H produced by the sampled 4 points
      H_selected = computeH_norm(selected_points1, selected_points2
                                          )
      # -- the points not selected
      others1 = np.delete(locs1, selected_idx, axis = 0)
      others2 = np.delete(locs2, selected_idx, axis = 0)
      # -- put 1 to augment the vectors
      ones = np.ones((others1.shape[0], 1))
      others2_extend = np.concatenate((others2, ones), axis = 1)
      # -- transform x2 to get predicted x1
      x2_transformed = np.stack([np.matmul(H_selected,
                                          others2_extend[j].T) for j
                                          in range(others2_extend.
                                          shape[0])])
      x2_transformed = (x2_transformed / x2_transformed[:,2])[:, :2
                                          ]
      # -- compute inliers
      delta = x2_transformed - others1
      delta_norm = np.linalg.norm(delta, axis = 1)
      inliers_idx = []
      for k in range(len(delta_norm)):
        if delta_norm[k] < inlier_tol:
          inliers_idx.append(k)
      if (len(inliers_idx) > inlier_count):
        inlier_count = len(inliers_idx)
        inliers = inliers_idx
        bestH2to1 = H_selected

    if type(bestH2to1) is not int:
      return bestH2to1, inliers
    else:
      return H_selected, [0 for l in range(len(delta_norm))]
```

2.2.4 Automated Homography Estimation and Warping

PlanarH.py

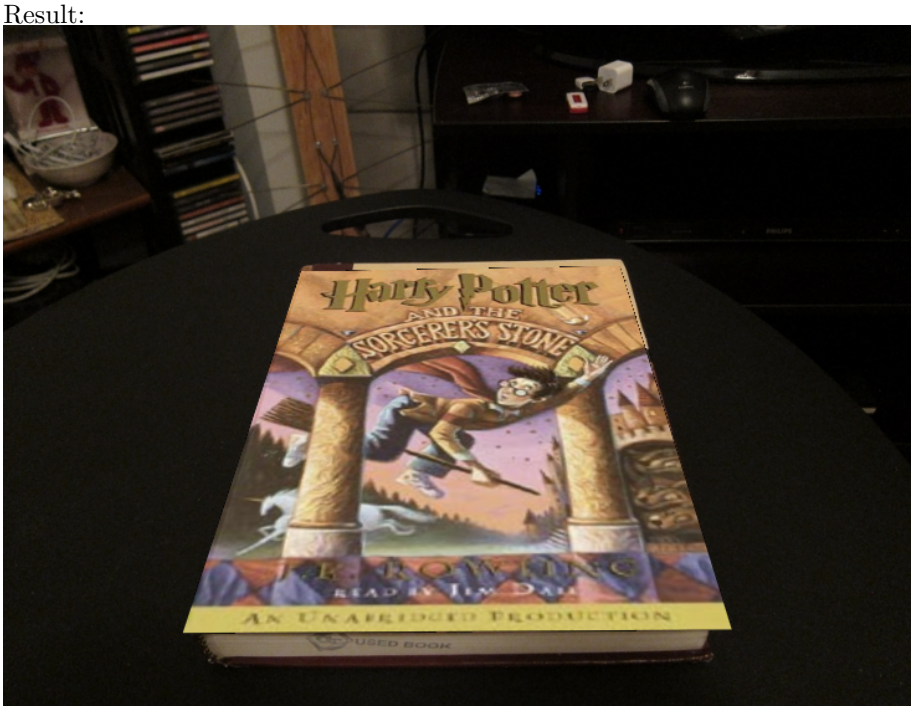
```
def compositeH(H2to1, template, img):

    #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.

    # TODO: Create mask of same size as template
    temp_height = template.shape[0]
    temp_width = template.shape[1]
    mask = np.ones((temp_height, temp_width, 3))

    # TODO: Warp mask by appropriate homography
    H1to2 = np.linalg.inv(H2to1)
    height = img.shape[0]
    width = img.shape[1]
    dim = (width, height)
    mask_w = cv2.warpPerspective(mask, H1to2, dim)

    # TODO: Warp template by appropriate homography
    template_w = cv2.warpPerspective(template, H1to2, dim)

    # TODO: Use mask to combine the warped template and the image
    composite_img = np.where(mask_w, template_w, img)
    return composite_img
```

HarryPotterize.py

```
import numpy as np
import cv2
import skimage.io
import skimage.color
from opts import get_opts

# Import necessary functions
from matchPics import *
from planarH import *
from helper import *
from displayMatch import *

# Q2.2.4
def warpImage(image1, image2, template, fname, opts):

    # -- compute Homography
    matches, locs1, locs2 = matchPics(image1, image2, opts)
    # -- column swap method from Stack Overflow https://
                                        stackoverflow.com/questions/
                                        4857927/swapping-columns-in-a
                                        -numpy-array
    locs1[:, 0], locs1[:, 1] = locs1[:, 1], locs1[:, 0].copy()
```

```
    locs2[:, 0], locs2[:, 1] = locs2[:, 1], locs2[:, 0].copy()
    matched1 = np.array([locs1[item[0]] for item in matches])
    matched2 = np.array([locs2[item[1]] for item in matches])
    bestH2to1, _ = computeH_ransac(matched1, matched2, opts)

    # -- resize template
    h1 = image1.shape[0]
    w1 = image1.shape[1]
    dim = (w1, h1)
    temp_resized = cv2.resize(template, dim)

    # -- composite the images
    composite_img = compositeH(bestH2to1, temp_resized, image2)
    cv2.imwrite(fname, composite_img)
    return composite_img

if __name__ == "__main__":
    opts = get_opts()
    image1 = cv2.imread('../data/cv_cover.jpg')
    image2 = cv2.imread('../data/cv_desk.png')
    cover = cv2.imread('../data/hp_cover.jpg')
    warpImage(image1, image2, cover, '2-2-4-result.png', opts)
```

Result:

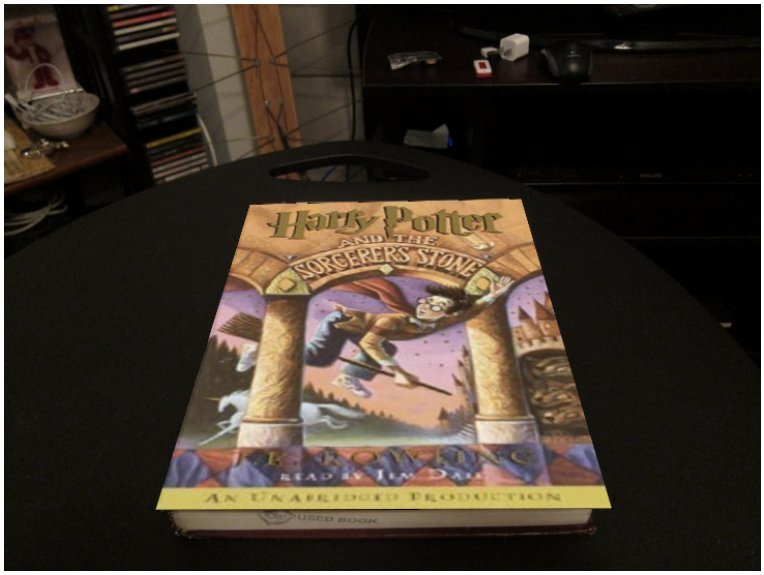


2.2.5 RANSAC Parameter Tuning

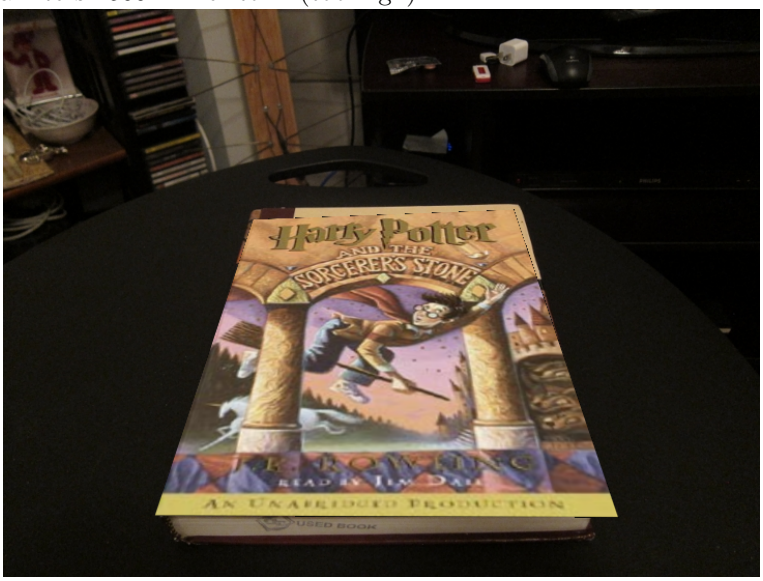
More iterations can increase the chance of selecting a good group of points that provide an accurate homography H. Too few iterations might lead to the point pair selections be biased and not fall on good, descriptive point pairs. Therefore, we want more iterations but within a time limit, as too many iterations make computation expensive.

Tolerance decides what we consider as inliers. Too high tolerance can lead to us selecting H with even large errors, but too low tolerance lead us to identify too few inliers, which might end up not computing any H that produce inliers.

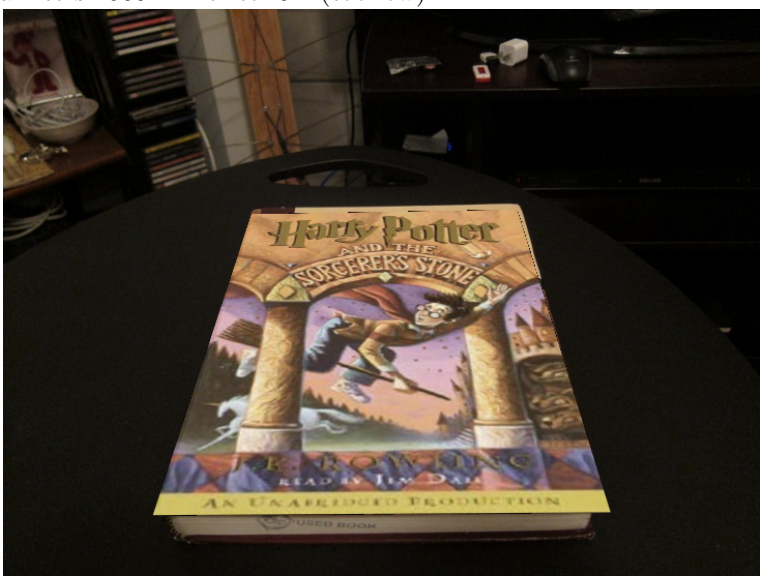
Best: –max iters 2500 –inlier tol 0.5



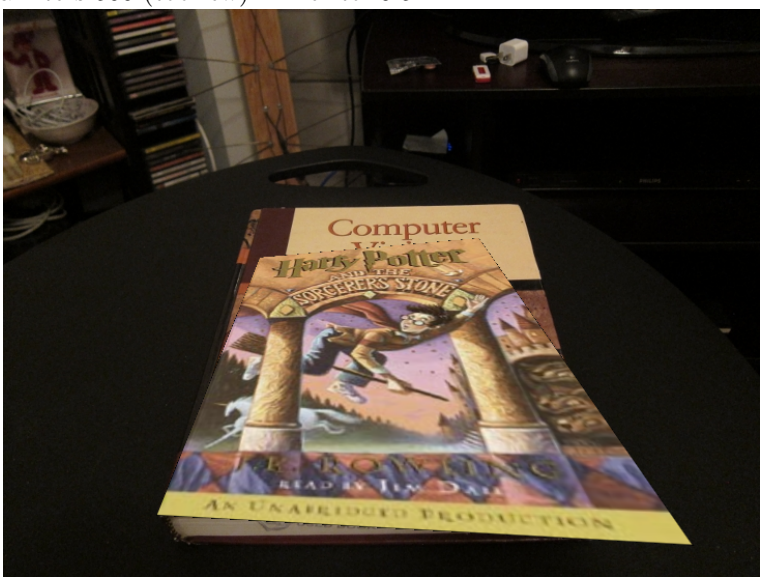
–max iters 2500 –inlier tol 2 (too high)



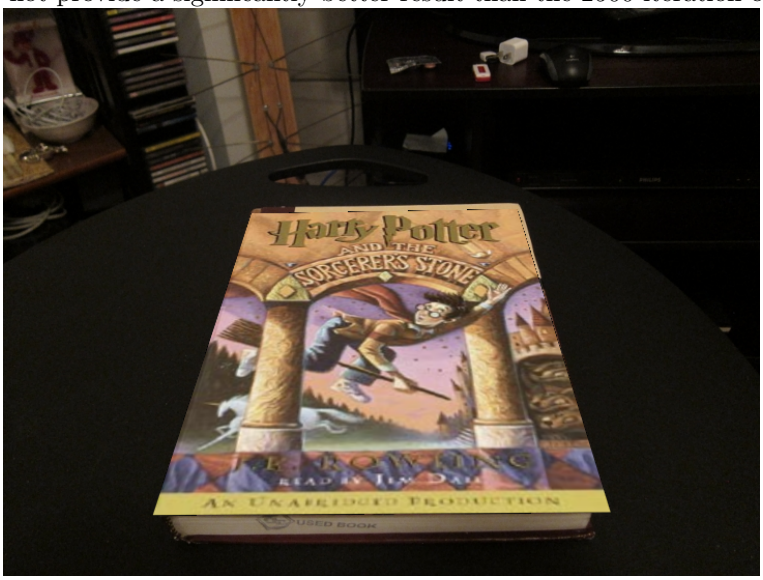
–max iters 2500 –inlier tol 0.2 (too low)



–max iters 500 (too few) –inlier tol 0.5



–max iters 10000 (too high) –inlier tol 0.5 This took more than 20 seconds but did not provide a significantly better result than the 2500 iteration one



Question 3

3.1 Incorporating Video

link to my video:
https://drive.google.com/file/d/1BldK5T5hR2Ep7ocgCKzC_HBw6pUNEx9s/view?usp=sharing

ar.py

```
import numpy as np
import cv2

#Import necessary functions
from opts import get_opts
from matchPics import *
from planarH import *
from helper import *
from displayMatch import *
from HarryPotterize import *

def incorporate_vid(src_path, dst_path, obj_path, save_f, opts):

    # -- retrieve frames
    frames_src = loadVid(src_path) # the panda frames
    frames_dst = loadVid(dst_path) # the book frames
    obj = cv2.imread(obj_path) # the cv book cover pic
    frames = [] # to store all the composite frames

    # -- match and warp frame by frame
    min_len = min(len(frames_src), len(frames_dst))
    for i in range(min_len):
      img = frames_src[i] # panda frame
      img2 = frames_dst[i] # book frame
      positions = np.nonzero(img)
      top, bottom = 55, 320 # remove the top and bottom dark spaces
      left, right = int(img.shape[1]/3), int(img.shape[1]*(2/3))
      img = img[top:bottom, left:right]

      # -- warp resized panda to book frame
      try:
        composite_img = warpImage(obj, img2, img, str(i) + 'result.
                                            png', opts)
        frames.append(composite_img)
        print("processed frame ", i)
      except:
        print("failed on frame ", i)
        continue

    # -- save to video
    out = cv2.VideoWriter("4-output.mp4", cv2.VideoWriter_fourcc(*'
                                        mp4v'), 30, (img2.shape[1],
                                        img2.shape[0]))
    for frame in frames:
        out.write(frame) # frame is a numpy.ndarray with shape (
                                            length, width, 3)
    out.release()
```

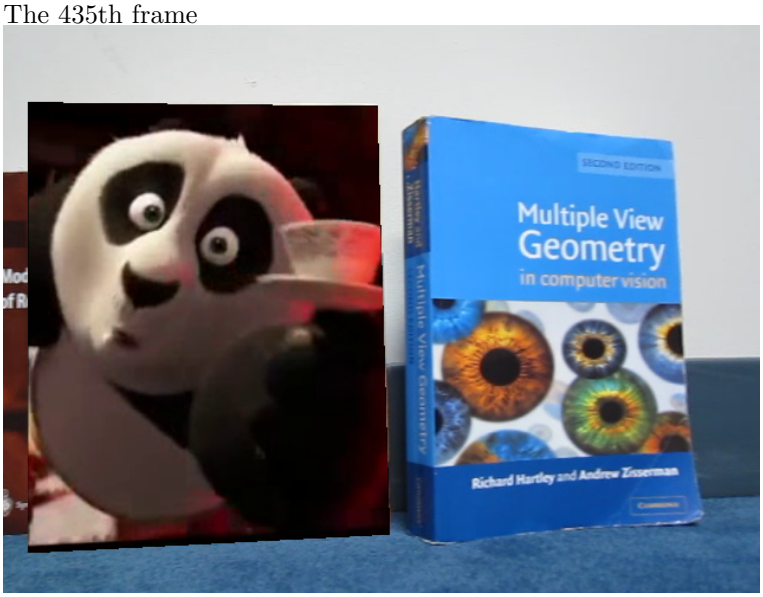
```
#Write script for Q3.1
if __name__ == "__main__":
    opts = get_opts()
    incorporate_vid("../data/ar_source.mov", "../data/book.mov", "
                                        ../data/cv_cover.jpg", "3-1-
                                        result.mov", opts)
```

The 11th frame

The 227th frame

The 255th frame

The 435th frame



Question 4 Create a Simple Panorama

The algorithm used in this question was discussed in a study group with Bhuvan Jhamb (bjhamb) and Roshan Roy (Roshanr)

panorama.py

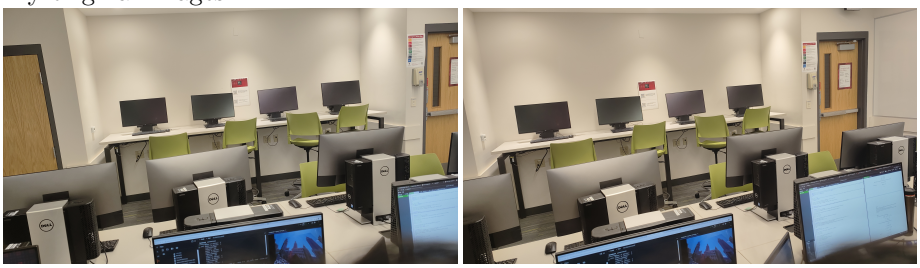
```
import numpy as np
import cv2
# Import necessary functions
from matchPics import *
from planarH import *
from helper import *
from displayMatch import *

# Q4
def pano(image1, image2, fname, ops):

    # -- create bigger canvas
    blank = np.zeros((image2.shape[0],int(0.5*image2.shape[1]),3),
                                    np.uint8)
    canvas = cv2.hconcat([blank, image2])

    # -- compute image 1's transformation onto canvas
    matches, locs1, locs2 = matchPics(image1, canvas, opts)
    plotMatches(image1, canvas, matches, locs1, locs2, "4-match.jpg
                                    ")
    # -- column swap method from Stack Overflow https://
                                    stackoverflow.com/questions/
                                    4857927/swapping-columns-in-a
                                    -numpy-array
    locs1[:, 0], locs1[:, 1] = locs1[:, 1], locs1[:, 0].copy()
    locs2[:, 0], locs2[:, 1] = locs2[:, 1], locs2[:, 0].copy()
    matched1 = np.array([locs1[item[0]] for item in matches])
    matched2 = np.array([locs2[item[1]] for item in matches])
    H, _ = computeH_ransac(matched2, matched1, opts)

    # -- warp image1 onto canvas
    image1_pred = cv2.warpPerspective(image1, H, (canvas.shape[1],
                                    canvas.shape[0]))
    mask = np.ones((image2.shape[0], image2.shape[1], 3))
    mask_w = cv2.warpPerspective(mask, H, (canvas.shape[1], canvas.
                                    shape[0]))
    dst = np.where(mask_w, image1_pred, canvas)

    # -- canvas cropping
    stiched_gray = cv2.cvtColor(dst, cv2.COLOR_BGR2GRAY)
    _, thresh = cv2.threshold(stiched_gray, 1, 255, cv2.
                                    THRESH_BINARY)
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.
                                    CHAIN_APPROX_SIMPLE)
    cnt = contours[0]
    approx = cv2.approxPolyDP(cnt, 0.009 * cv2.arcLength(cnt, True)
                                    , True)
    max_x = np.max(approx[:, 0, 0])
```

```
    max_y = np.max(approx[:, 0, 1])
    min_x = np.min(approx[:, 0, 0])
    min_y = np.min(approx[:, 0, 1])
    output = dst[min_y:max_y,min_x:max_x]
    cv2.imwrite("cropped-"+fname, output)


if __name__ == "__main__":
    opts = get_opts()
    image1 = cv2.imread('../data/left6.jpg')
    image2 = cv2.imread('../data/right6.jpg')
    pano(image1, image2, "4-pano-test6.jpg", opts)
```

Original images

Panorama

My original images



Panorama of my original images