- **Name**: Ranai Srivastav
- **Andrew ID**: ranais

NOTE:
- All code must be run from the python directory
- Assumed matrices are invertible based on Piazza post

Question 1

Q1.1 Prove a Homography exists

P, P' are 3×4 projection matrices.

$$P = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \qquad P' = \begin{bmatrix} p'_{11} & p'_{12} & p'_{13} & p'_{14} \\ p'_{21} & p'_{22} & p'_{23} & p'_{24} \\ p'_{31} & p'_{32} & p'_{33} & p'_{34} \end{bmatrix}$$

All points X_π that lie in Plane Π are related via a Homography.
Thus,
point x_π is related to point x via projection P_1 when viewed from camera C
point x_π is related to point x' via a projection P_2 when viewed from camera C'

Let $x_\pi = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$. In 3D world coordinates, since all points lie on plane Π, we can set $z = 0$ at plane Π.

Then fully expressed x and x' are,
$x = \lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$ and $x' = \lambda' \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$

and the projection matrix links the points as

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = P \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} \qquad \lambda' \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = P' \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix}$$

However, since we assume $z = 0$ we can disregard the 3rd column from projection matrix P and the homogenous world coordinates to give a simplified

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{14} \\ p_{21} & p_{22} & p_{24} \\ p_{31} & p_{32} & p_{34} \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} \qquad \lambda' \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} p'_{11} & p'_{12} & p'_{14} \\ p'_{21} & p'_{22} & p'_{24} \\ p'_{31} & p'_{32} & p'_{34} \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = H \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} \qquad \lambda' \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H' \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

$$\lambda H^{-1} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \lambda' H'^{-1} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

$$\lambda H^{-1} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \lambda' H'^{-1} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

$$\frac{\lambda}{\lambda'} H^{-1} H' \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

Question 1.2

1.2.1

h has 8 degrees of freedoms. h_{33} is constrained to be 1.

1.2.2

Atleast 4 correspondences (4 point pairs) are required to solve the equation.

1.2.3

$$\lambda \mathbf{x}_1^i \equiv H \mathbf{x}_2^i$$

$$\lambda \begin{bmatrix} x_1^i \\ y_1^i \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & \dots & h_{13} \\ \vdots & \vdots & \vdots \\ h_{31} & \dots & h_{33} \end{bmatrix} \begin{bmatrix} x_2^i \\ y_2^i \\ 1 \end{bmatrix}$$

Solving the above, we get

$$A_i = \begin{bmatrix} -x_2^i & -y_2^i & -1 & 0 & 0 & 0 & x_1^i x_2^i & x_1^i y_2^i & x_1^i \\ 0 & 0 & 0 & -x_2^i & -y_2^i & -1 & y_1^i x_2^i & y_1^i y_2^i & y_1^i \end{bmatrix} \qquad h = \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix}$$

1.2.4

The trivial solution for $Ah = 0$ will be $h = 0$.
A non-trivial solution exists only if A is not a full-rank matrix, i.e, The null-space of A is atleast 1. Since full-rank for a non-square matrix is defined as the minimum of row space and col space, and we have 1 more column than rows, our max possible rank is 8. In a perfect world with no noise, the smallest eigenvalue after performing SVD would be 0. However due to noise and an over-constrained system, the smallest singular value will be close to be 0 but not exactly 0 after following the least square minimization approach. The eigenvalues are the square root of the values of $A^T A$. The smallest singular value gives us a measure of how close we are to the perfect solution.

Question 1.4

1.4.1 Rotating around the center of the camera

The world coordinates are related via a rotation through

$$\begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \end{bmatrix} = R \begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \end{bmatrix}$$

where R is the rotation matrix in $R_{3\times3}$.

Expressing 3D world coordinates in 2D can be expressed as under no translation or rotation

$$\lambda \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = K_1 \begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \end{bmatrix}$$

under no translation but rotation

$$\lambda \begin{bmatrix} x_R \\ y_R \\ 1 \end{bmatrix} = K_2 R \begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \end{bmatrix} \implies \lambda R^{-1} K_2^{-1} \begin{bmatrix} x_R \\ y_R \\ 1 \end{bmatrix} = \begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \end{bmatrix}$$

This is possible because $[RT] \in \mathbb{R}_{3\times4}$ where T is a 0 vector projects everything to $\begin{bmatrix} a \\ b \\ c \\ 0 \end{bmatrix}$ where $a, b, c \in \mathbb{R}$ after rotation from $[RT]$. Thus, we can drop T and the last row from the above.

Substituting from the eqn above,

$$\lambda \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = K_1 R^{-1} K_2^{-1} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

where $K_1 R^{-1} K_2^{-1}$ is the homography matrix H that satisfies the equation of the form $x1 \equiv Hx_2$

1.4.2

From the above question, we get $H_\theta = K_1 R^{-1} K_2^{-1}$.
We also know that if R_θ defines a rotation matrix that rotates the corresponding point by θ,
$R_{2\theta} = R_\theta^2$
Thus $K_1 = K_2$, and
$H_\theta^2 = (K_1 R^{-1} K_2^{-1})^2 = (K_1 R^{-1} K_2^{-1})(K_1 R^{-1} K_2^{-1}) = (K R^{-1} R^{-1} K^{-1})$

1.4.3

Planar homographies require the object to be sufficiently far away to ensure that the distant object can be assumed to be a plane and all points in the distance can be assumed to lie this plane. A homography when used to transform an image given these constraints is also bound by the range of it's transformation, i.e., the homography matrix can only warp what it has seen to the new perspective. It is not possible for the homography transformation to create new data, and thus, a homography will only be able to warp the existing information in the frame and this yields "black regions" in implementation (areas where the homography does not have data about).

1.4.4

The equation of a line in 3D is given as $x = \mathbf{x1} + \lambda(\mathbf{x_2} - \mathbf{x_1})$ where $\mathbf{x1}, \mathbf{x2}, \mathbf{x}$ are 4 element homogenized 3D world coordinates.
Let $P \in \mathbb{R}_{3\times4}$ be a projection matrix such that $x = P\mathbf{x}$ where x is the homogenized 2D world coordinate.
Then

$$P\mathbf{x} = P(\mathbf{x_1} + \lambda(\mathbf{x_2} - \mathbf{x_1}))$$

$$P\mathbf{x} = P\mathbf{x_1} + \lambda P\mathbf{x_2} - \lambda P\mathbf{x_1}$$

Using the equation $x = P\mathbf{x}$

$$x = x_1 + \lambda x_2 - \lambda x_1 \implies x = x_1 + \lambda(x_2 - x_1)$$

Since each of the points is a 2D homogenous coordinate, the expression for a line in 3D can now be expressed in 2D.

Question 2

2.1.1

Harris Corner Detector
The Harris corner detector works by first computing the following:

- I_X the gradient of the image in the x direction calculated using a convolution
- I_Y the gradient of the image in the y direction calculated using a convolution
- $A = \begin{bmatrix} I_X^2 & I_X I_Y \\ I_Y I_X & I_Y^2 \end{bmatrix}$ calculated by $\begin{bmatrix} I_X \\ I_Y \end{bmatrix} \begin{bmatrix} I_X & I_Y \end{bmatrix}$

Based on these, corners in the image can be detected using the formula $Det(A) - \alpha Trace(A)^2$ where α is a parameter that can be tuned.

The runtime complexity of Harris Corner Detection is $O(WHK^2D)$ where the kernel is of size $K \times K$ and the image is of size $W \times H$, and D is the time complexity of calculating the Determinant/Eigenvalues of a matrix.

Harris corner detection is not resistant to scale or environment brightness can be made so using additional methods such as Gaussian Pyramids. It is invariant to rotation since the change in theta will not change the gradient.

FAST
The FAST algorithm works by defining a Bresenham circle (can be thought of as a pixelated circle) around a pixel of interest I where the radius can be assumed to be R. The brightness of I is then added and subtracted by a threshold t. The algorithm then checks for a contiguous circle of length n' where pixels in the circle are brighter or darker than the corresponding thresholded values. We define another parameter n as a threshold defined such that if a contiguous arc in the Bresenham circle is found of length greater than n pixels, the center of the circle I is considered a corner. If $n' > n$ then this particular corner is considered to be a corner.

The following piecewise function is defined to classify a pixel on the circle as either darker, similar, or brighter

$$S_{p\to x} = \begin{cases} d, & I_{p\to x} \leq I_p - t & \text{(darker)} \\ s, & I_p - t < I_{p\to x} < I_p + t & \text{(similar)} \\ b, & I_p + t \leq I_{p\to x} & \text{(brighter)} \end{cases}$$

This process can be farther optimized by checking the top most, bottom most, left most, and right most pixels in the circle (corresponding to where the unit circle intersects the x-y axes) and continuing with the full calculation described above (comparing n with n') *iff* 3 of the 4 points belong to the same classification.

FAST is resistant to rotations since the circle is infinitely symmetrical to rotation. However, scale and changes in brightness without changing the threshold t will lead to loss of performance.

FAST has a runtime complexity of (WHR) where R is the radius of the Bressenham circle in pixels and the length of the contiguous pixel circle is proportional to the radius of the circle in pixels.

2.1.2

Feature: A part of the image that stands out and is of interest for some reason, in this case, identifying common areas across images. Similar to keypoint.
Descriptor: A unique fingerprint of a feature in an image that compares it to the nearby pixels to help match features across images.

BRIEF Descriptor
BRIEF descriptor is a method that relies on binary strings to uniquely identify a feature in an image. BRIEF considers certain locations in the image and then compares the intensities of these images at other points in the image:
- 1 if original pixel intensity is lesser than other pixel intensity
- 0 otherwise

BRIEF is preferred because of it is very easy to compute through XOR operations and in certain architectures that support bit counting, it can be significantly faster than on those that do not.

BRIEF is sensitive to rotations and scaling and thus, the input to brief must be normalized in some way to take into account rotation and scale across features.

Filter banks

Filter banks are banks of kernels that can be applied to images through convolutions to identify features in an image. The filter bank discussed in lecture (LeungMalik) consists of filters that apply Gaussian blurring then first order derivatives, second order derivatives, and simple blurring operations. Each of these helps identify features such as edges at different angles and of different lengths. Filters can be helpful in identifying features but are not helpful in identifying (descriptors) on their own. However, combining multiple filters might create a unique fingerprint by creating a N dimensional vector for each pixel where N is the number of filters. However, a filter bank will not be an efficient descriptor since similar features will have similar results.

More complex filter banks can be used to identify higher level features that are specific to the task at hand (classifiers for apples and bananas might have circular filters, etc.), and such information can be used to uniquely identify parts of images.

Filter banks are rotation invariant and depending on construction might be scale invariant upto a certain point.

2.1.3 Matching methods

Hamming distance

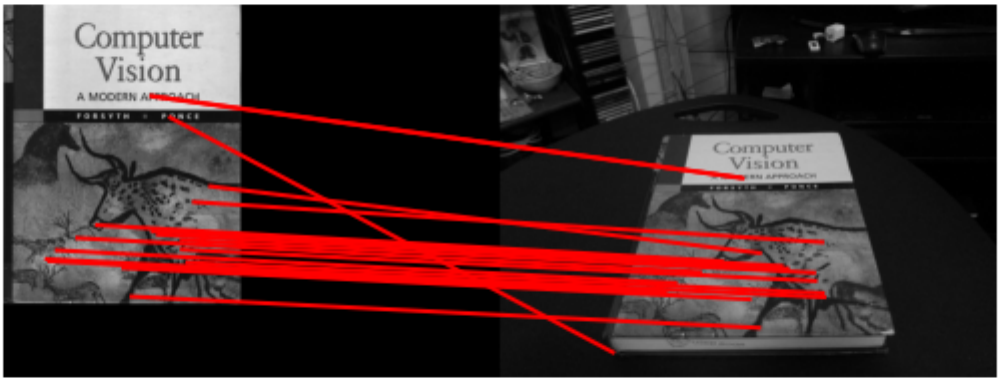
This method relies on elementwise XOR operation to identify if two binary strings are the same. The hamming distance for a binary string is a measure of how many bitflips need to be made to convert one string to another.

The Euclidean distance, also called the L2 Norm is the colloquial definition of distance. Computing the euclidean distance of a n-binary string would mean squaring the difference of numerous 1s and 0s, adding those and then taking the square root of the sum. This would be a slow operation and is equivalent to the counting the number of 1s in the string and then taking it's square root. Thus the Hamming distance is an easier computation and the provides the same information as the Euclidean distance.

The only difference between the two methods is the absence of square root in the Hamming distance and this makes the Hamming distance a more tangible metric.

2.1.4 Feature Matching

Best output of `DisplayMatches`:



Code:

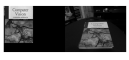
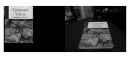
```
import cv2
import json
import numpy as np
from tqdm import tqdm
from opts import get_opts
from helper import plotMatches
from matchPics import matchPics


def displayMatched(opts, image1, image2):
    """
    Displays matches between two images
    Input    -----    opts: Command line args    image_left, im_right_src:
Source images    """    results = {}

    sigma = [0.10, 0.125, 0.15, 0.175, 0.200, 0.400]
    ratio = [0.3, 0.5, 0.7, 0.9, 1.1, 2.2]

    with open("displayMatch_values.json", "w") as f:
```

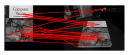
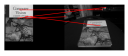
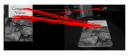
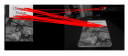
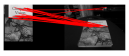
```
        for i in tqdm(range(len(sigma))):
            results[sigma[i]] = {}
            for j in tqdm(range(len(ratio))):

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

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

        f.write(json.dumps(results, indent=4))


if __name__ == "__main__":
    opts = get_opts()
    image1 = cv2.imread('../data/cv_cover.jpg')
    image2 = cv2.imread('../data/cv_desk.png')

    displayMatched(opts, image1, image2)
```

2.1.5

Sigma refers to the threshold used by FAST to assign contiguous pixels one of three state Darker, Similar, Lighter. For an image to be considered darker, it must be `image_intensity - threshold` and vice versa for lighter. Thus, increasing the `sigma` value corresponds to lesser matches because lesser points meet that criteria. This is visible when going bottom to top in the table and the number of matches increasing.

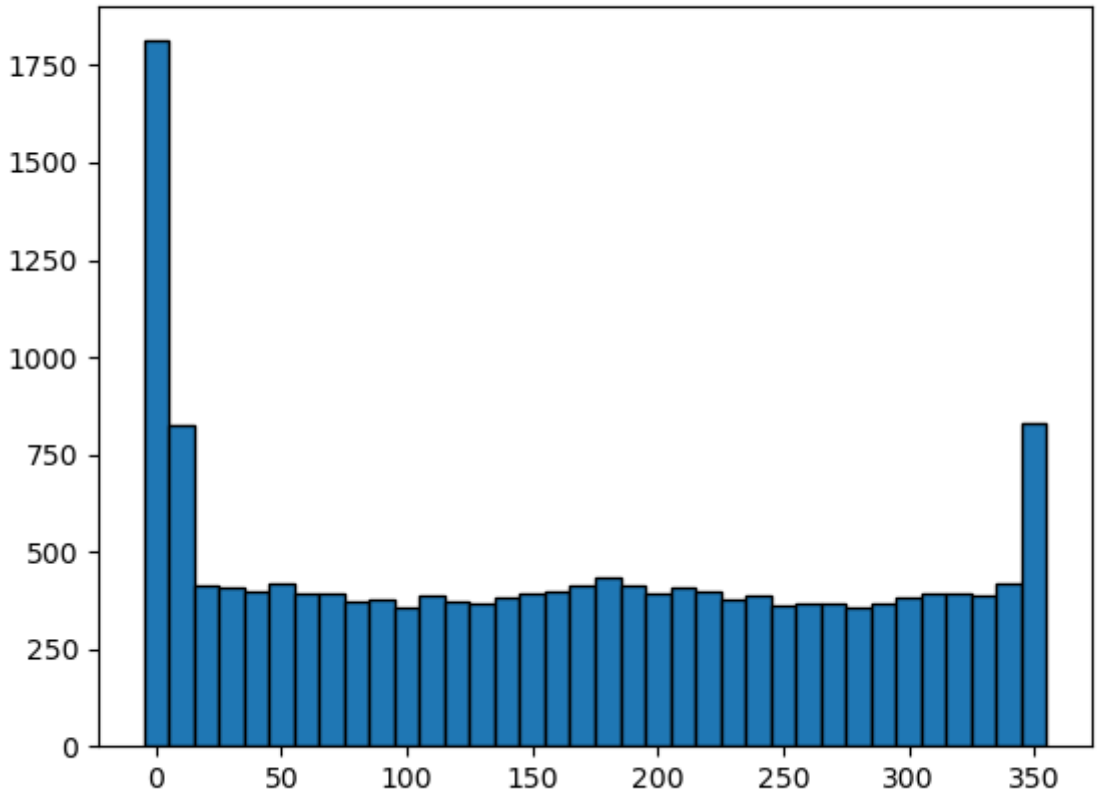
Ratio in terms of feature matching in this context is the ration of the Hamming distance between the first and the second closest descriptor and is sued to filter ambiguous matches. Increasing this value allows to include more ambiguous matches. This is visible when going from left to right and the value increasing.

Sigma\Ratio	0.3	0.5	0.7	0.9	1.1	2.2
0.1						
0.125						
0.15						

Sigma\Ratio	0.3	0.5	0.7	0.9	1.1	2.2
0.175						
0.200						
0.400						

2.1.6

BRIEF feature descriptor is not rotation invariant because it is performs random sampling of the point coordinates. It is not guaranteed that 2 point coordinates will similarly differ in terms of pixel intensity after rotation, and thus, it is not rotation invariant. This is evident from the below graph.



Code:

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

#Q2.1.6

def rotTest(opts):

    # TODO: Read the image and convert to grayscale, if necessary
    img = cv2.imread("../data/cv_cover.jpg")
    num_matches = {}
    for i in range(36):

        rot = scipy.ndimage.rotate(img, i * 10, reshape=False)

        # TODO: Compute features, descriptors and Match features
        matches, loc1, loc2 = matchPics(img, rot, opts)

        # TODO: Update histogram
        num_matches[i*10] = len(matches)

    # TODO: Display histogram
    hist = plt.bar(list(num_matches.keys()), list(num_matches.values()),
width=10, edgecolor='black')
    print(list(num_matches.values()))
    plt.savefig("../data/BRIEF_rotation_test")


if __name__ == "__main__":

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

2.1.7

1. A possibility of making BRIEF rotation invariant is to create a Bressenham circle with the radius min(image_width, image_height). The algorithm must then use the same metric as BRIEF - 1 if intensity of px1 is greater than px2, 0 otherwise. The Bressenham circle will ensure there are infinite axes of symmetry. To make it more robust, we can also take fractions of the minimum value to add an element of scale invariance.

2.2

2.2.1 `computeH`

```
def computeH(x1, x2):
    """
    Computes the Homography matrix H [3x3] given a set of points
    Inputs:    -----------    x1, x2 are [Nx2] matrices    """    #Q2.2.1

    full_mat = []

    x1 = x1.T
    x2 = x2.T

    for i in range(x1.shape[0]):
        p0 = x1[i]  # Example point from locs1 = cv_cover
        p1 = x2[i]  # Example point from locs2 = cv_desk

        x_vec = [-p0[0], -p0[1], -1,      0,      0,  0,  p0[0] * p1[0],
p0[1] * p1[0],  p1[0]]    # finds relationship w.r.t p1[0]
        y_vec = [     0,      0,  0, -p0[0], -p0[1], -1,  p0[0] * p1[1],
p0[1] * p1[1],  p1[1]]    # finds relationship w.r.t p1[1]

        full_mat.append(x_vec)
        full_mat.append(y_vec)

    full_mat = np.array(full_mat)

    # Compute the homography between two sets of points
    u, sigma, v = np.linalg.svd(full_mat)

    H_2to1 = v[-1].reshape((3, 3))

    return H_2to1    # desk to cover
```

2.2.2 `computeH_norm`

```
def computeH_norm(x1, x2) -> np.ndarray:
    """
    Computes the Homography matrix for a normalized state space     x1, x2:
Nx2 arrays
     Output:     ---------     Homography matrix [3x3]    """    #Q2.2.2
```

```
    # Compute the centroid of the points    x1_centroid = get_centroid(x1)
# Nx2 np arrays
    x2_centroid = get_centroid(x2)

    # Move the points to the centroid
    x1_shifted = x1 - x1_centroid
    x2_shifted = x2 - x2_centroid

    # Normalize the points so that the largest distance from the origin is
equal to sqrt(2)
    x1_max = np.max(x1_shifted, axis=0)  # [2 elem vec] get the largest x,
y, z value from all of x1
    x2_max = np.max(x2_shifted, axis=0)  # [2 elem vec]
x2

    # Similarity transform 1    T1 = np.array([[np.sqrt(2) / x1_max[0],
0, -1 * np.sqrt(2) * x1_centroid[0]/x1_max[0]],
                   [                    0, np.sqrt(2) / x1_max[1], -1 *
np.sqrt(2) * x1_centroid[1]/x1_max[1]],
                   [                    0,                    0,  1]])
    x1_norm = T1 @ x1.T

    # Similarity transform 2
    T2 = np.array([[np.sqrt(2) / x2_max[0],                    0, -1 *
np.sqrt(2) * x2_centroid[0]/x2_max[0]],
                   [                    0, np.sqrt(2) / x2_max[1], -1 *
np.sqrt(2) * x2_centroid[1]/x2_max[1]],
                   [                    0,                    0, 1]])
    x2_norm = T2 @ x2.T

    # TODO: Compute homography
    H = computeH(x1_norm, x2_norm)

    # TODO: Denormalization
    denorm_H_2to1 = np.linalg.inv(T1) @ H @ T2

    return denorm_H_2to1
```

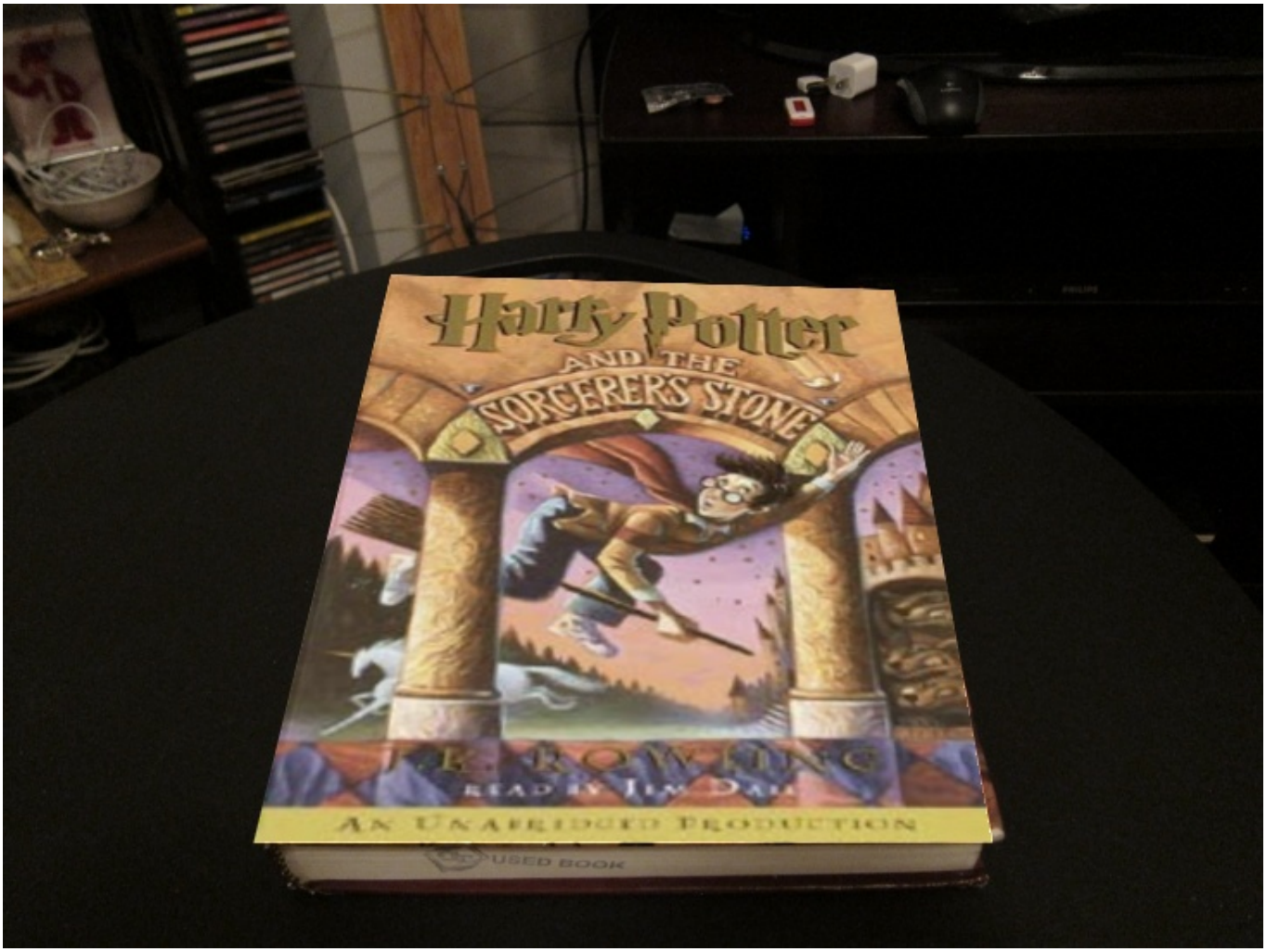
2.2.3 computeH_ransac

```
def computeH_ransac(locs1, locs2, opts):
    """
    Computes the Homography matrix H and detects outliers
    Inputs:    ----------    locs1, locs2 [Nx2]: Corresponding index matched
```

```
descriptors across the two images    """    #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
    n = range(len(locs1))

    best_inlier_count = -1
    best_H = -1.0

    padding = np.ones((locs1.shape[0], 1))
    if locs1.shape[1] == 2:
        locs1 = np.hstack((locs1, padding))

    if locs2.shape[1] == 2:
        locs2 = np.hstack((locs2, padding))

    for i in range(max_iters):
        sampled_points = np.random.choice(n, 4, replace=False)
        H_maybe = computeH_norm(locs1[sampled_points],
locs2[sampled_points])   # locs1 is cv_cover,

# locs2 is cv_desk,
# [Nx3]
        x1 = H_maybe @ locs2.T                      # 3x3 @ 3xN = 3xN
        x1 = (x1.T / x1.T[:, 2].reshape(-1, 1)).T  # [xL, yL, L] -> [x, y,
1]
        err = np.linalg.norm(x1 - locs1.T, axis=0) # 1xN with L2 distance

        inlier_bool_idx = err < inlier_tol    # 1xN with booleans where
tolerance is lesser
        curr_inliers_count = len(err[inlier_bool_idx])
        print(f" Curr Inlier Count: {curr_inliers_count} Best Inlier Count:
{best_inlier_count}")
        if curr_inliers_count > best_inlier_count:
            best_inlier_count = len(err[inlier_bool_idx])
            best_H = H_maybe

    return best_H, inlier_bool_idx
```

2.2.4 `warpImage`

Harry Potterize Image



Code:

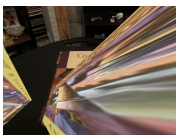
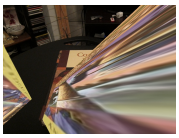
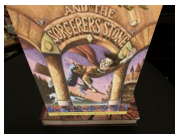
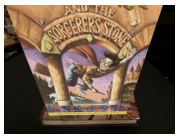
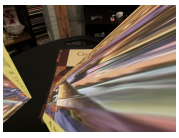
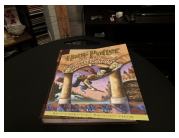
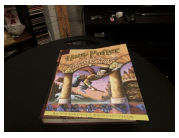
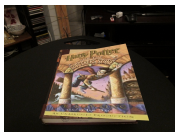
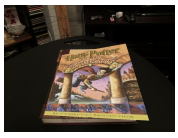
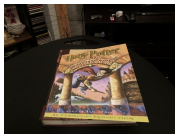
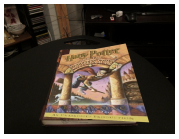
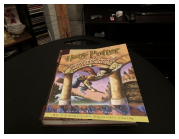
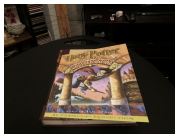
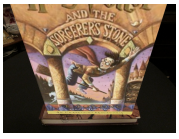
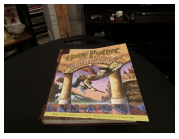
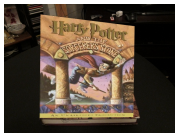
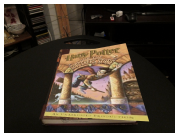
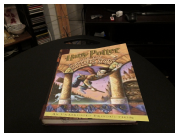
```
def warpImage(opts):
    cv_cover = cv2.imread("../data/cv_cover.jpg")
    cv_desk = cv2.imread("../data/cv_desk.png")
    hp_cover = cv2.imread("../data/hp_cover.jpg")

    matches, locs1, locs2 = matchPics(cv_cover, cv_desk, opts)  # locs1 is
cv_cover [N1x2]
    # locs2 is cv_desk  [N2x2]
    l1_idx = matches[:, 0]  # extracting locs1 matches
    l2_idx = matches[:, 1]
    locs1 = locs1[l1_idx]  # Erasing extraneous matches and aligning locs 1
and 2
    locs2 = locs2[l2_idx]  # locs1 and locs2 [Nx2]

    bestH, inliers = computeH_ransac(locs1, locs2, opts)  # locs1 is
cv_cover, locs2 is cv_desk

    hp_cover = cv2.resize(hp_cover, (cv_cover.shape[1], cv_cover.shape[0]))
```

```
    fin_img = compositeH(np.linalg.inv(bestH), hp_cover, cv_desk)

    cv2.imshow("FIN IMG", fin_img)
    cv2.waitKey(0)

    # cv2.imwrite(f"../data/{opts.inlier_tol}-{opts.max_iters}.jpg",
fin_img)
```

2.2.5 RANSAC parameter tuning

Inlier Tolerance is the value that RANSAC uses to caulculate if values are in-range or not when classifying outliers. Increasing this value leads to too many values being classified as inliers and they might skew the data. Similarly, if this value is too small, we might get lesser than 4 values, which makes SVD fail since we have an under-constrained system.

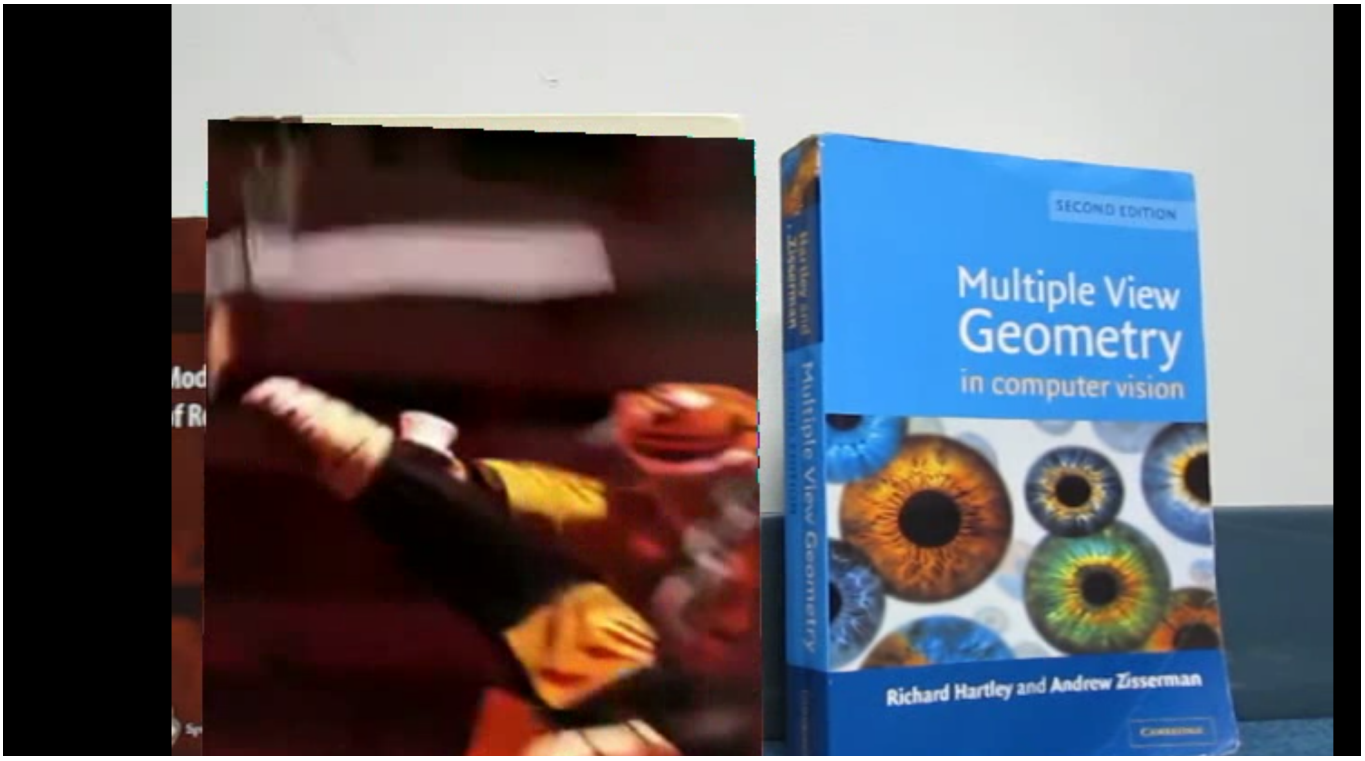
Max_iters: Is the number of iterations that RANSAC runs for. Increasing this number runs RANSAC for longer and leads to more exploration of the state space but also takes more time.

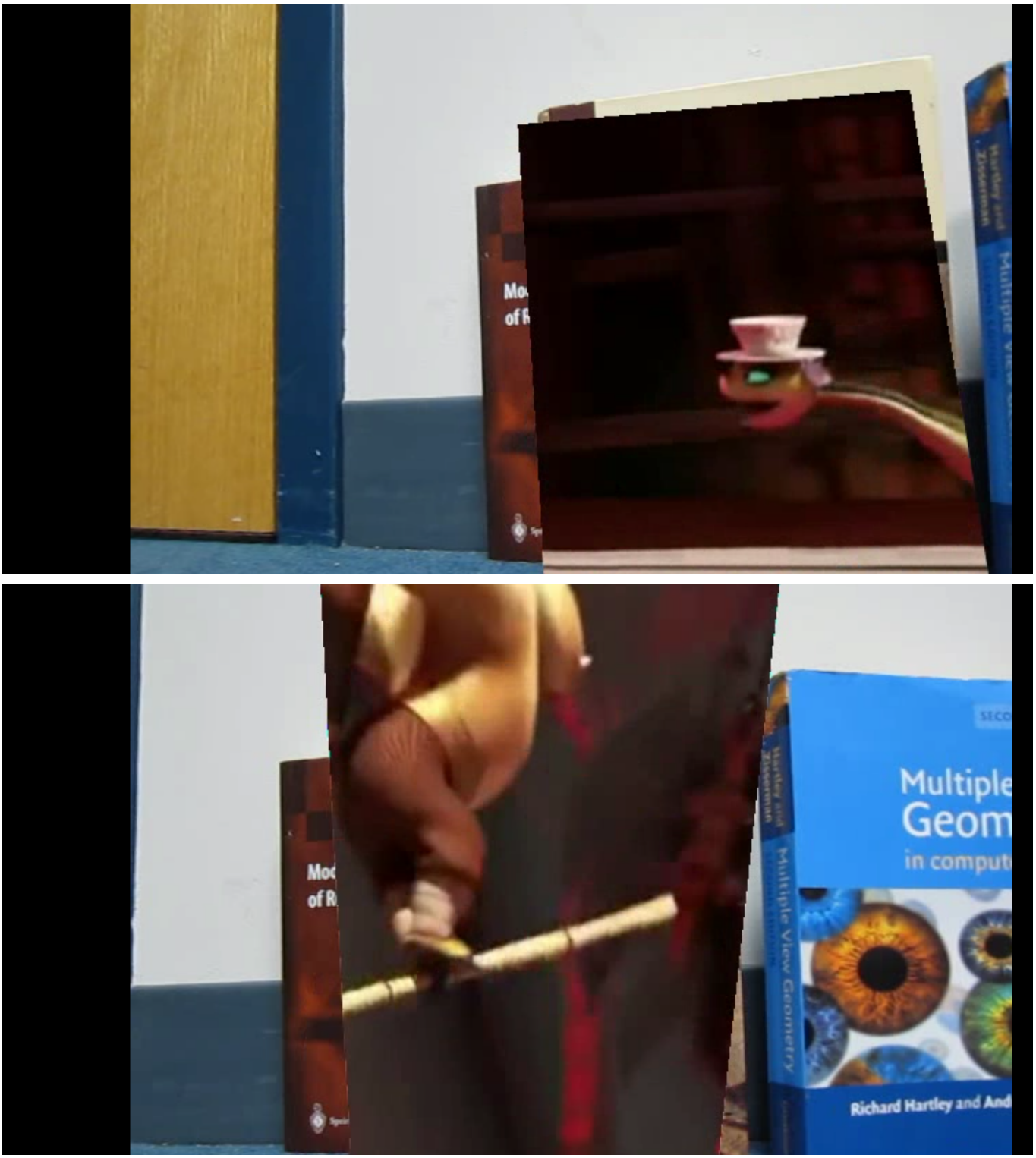
Max_iters\tolerance	0.5	1	2	4	8
100					
200					
400					
800					
1600					

Question 3: AR

3.1 Video

Link to full video:
https://drive.google.com/file/d/1d9sJR9lZoHJhAxMVtAGEDdalQCkKxokH/view?usp=sharing






```
import cv2

import numpy as np

from opts import get_opts

from helper import loadVid
```

```
from matchPics import matchPics

from multiprocessing import Pool

from planarH import computeH_ransac



def process_one_frame(args):

"""Process one frame of the AR video by calculating homography and warping



Args:

args (frame1, frame2, cover_frame, options, index): to calculate the
homography between image and video and warp it



Returns:

(index, img: 1 processed frame

"""



panda_frame, book_frame, cv_cover, opts, i = args

panda_frame_reshaped = cv2.resize(panda_frame, (cv_cover.shape[1],
cv_cover.shape[0]))



matches, pts_cv, pts_book = matchPics(cv_cover, book_frame, opts)

pts_cv = pts_cv[matches[:, 0]][:, ::-1]

pts_book = pts_book[matches[:, 1]][:, ::-1]



H_booksToCV, _ = computeH_ransac(pts_cv, pts_book, opts) # Should I invert
this?
```

```
mask = np.ones((panda_frame_reshaped.shape[0],
panda_frame_reshaped.shape[1]), dtype=np.uint8) * 255

warped_mask = cv2.warpPerspective(mask, np.linalg.inv(H_booksToCV), dsize=
(book_frame.shape[1], book_frame.shape[0])).astype(np.uint8)

warped_panda = cv2.warpPerspective(panda_frame_reshaped,
np.linalg.inv(H_booksToCV), dsize=(book_frame.shape[1],
book_frame.shape[0]))

composite_img = cv2.bitwise_and(book_frame, book_frame,
mask=cv2.bitwise_not(warped_mask))

masked_panda = cv2.bitwise_and(warped_panda, warped_panda, mask=warped_mask)

composite_img = cv2.add(composite_img, masked_panda)



return i, composite_img



def ar():

"""Main function

"""

ar_panda_frames = loadVid('../data/ar_source.mov') # 511 frames x 360 width
x 640 height x 3 channels

book_frames = loadVid('../data/book.mov') # 641 frames x 480 width x 640
height x 3 channels

cv_cover = cv2.imread('../data/cv_cover.jpg') # 440 height x 350 width x 3
channels

panda_unbarred = ar_panda_frames[:, 40:315, :, :]

width_desired = int(cv_cover.shape[1] / cv_cover.shape[0] *
panda_unbarred.shape[1]) # Aspect ratio
```

```
height_desired = panda_unbarred.shape[1] # That one



x_center = int(np.floor(panda_unbarred.shape[2] / 2))

x_start = int(x_center - np.floor((width_desired / 2)))

x_end = int(x_center + np.floor((width_desired / 2)))



y_start = 0

y_end = height_desired

ar_panda_frames = panda_unbarred[:, y_start:y_end, x_start:x_end, :]

final_frames = np.zeros((len(ar_panda_frames), book_frames[0].shape[0],
book_frames[0].shape[1], 3), dtype=np.uint8)



# arguments for multiproc

args = []

for i in range(len(ar_panda_frames)):

arg = (ar_panda_frames[i], book_frames[i], cv_cover, get_opts(), i)

args.append(arg)



with Pool(processes=16) as pool:

results = list(pool.imap(process_one_frame, args))



results.sort(key=lambda x: x[0])



width = book_frames[0].shape[1]
```

```
height = book_frames[0].shape[0]



fourcc = cv2.VideoWriter_fourcc(*'XVID')

video = cv2.VideoWriter('../data/panda_video.avi', fourcc, 25, (width,
height))



for i, composite_img in results:

final_frames[i] = composite_img

video.write(composite_img)



video.release()



if __name__ == "__main__":

ar()
```

3.2 Realtime AR EC

NOT ATTEMPTED

4 Panaroma

```
import cv2
import numpy as np

def compositeH_pano(H2to1, template, img):

    mask = np.ones(template.shape[:2], dtype=np.uint8) * 255
    warped_mask = cv2.warpPerspective(mask, H2to1, (img.shape[1],
img.shape[0])).astype(np.uint8)
    warped_template = cv2.warpPerspective(template, H2to1, (img.shape[1],
```

```
img.shape[0])).astype(np.uint8)
    composite_img = cv2.bitwise_and(np.uint8(img),
np.uint8(warped_template), mask=np.uint8(warped_mask))

    bg = cv2.bitwise_and(img, img, mask=cv2.bitwise_not(warped_mask))
    fg = cv2.bitwise_and(warped_template, warped_template, mask=warped_mask)
    composite_img = cv2.add(bg, fg)

    cv2.imwrite('Panaroma.png', warped_mask)
    cv2.waitKey(0)

    return composite_img

im_resize_k = 0.5

im_right_src = cv2.imread('../data/middle.jpeg')
im_left_src = cv2.imread('../data/left_most.jpeg')

im_left  = cv2.resize(im_left_src, (int(im_resize_k * im_left_src.shape[1]),
int(im_resize_k * im_left_src.shape[0])))
im_right = cv2.resize(im_right_src, (int(im_resize_k *
im_left_src.shape[1]), int(im_resize_k * im_left_src.shape[0])))

im_left_gray = cv2.cvtColor(im_left, cv2.COLOR_BGR2GRAY)
im_right = cv2.copyMakeBorder(
    im_right,
    250, 250, 250, 250,
    borderType=cv2.BORDER_CONSTANT,
    value=[0, 0, 0]
)
im_right_gray = cv2.cvtColor(im_right, cv2.COLOR_BGR2GRAY)

orbject = cv2.ORB_create()
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

pts_left, desc_left = orbject.detectAndCompute(im_left_gray, None)
pts_right, desc_right = orbject.detectAndCompute(im_right_gray, None)
matches = bf.match(desc_left, desc_right)
left_matched = np.float32([pts_left[m.queryIdx].pt for m in
matches]).reshape(-1, 1, 2)
right_matched = np.float32([pts_right[m.trainIdx].pt for m in
matches]).reshape(-1, 1, 2)
H_rt2lf = cv2.findHomography(left_matched, right_matched, cv2.RANSAC, 4.0)
[0]

composite_image = compositeH_pano(H_rt2lf, im_left, im_right)
```

```
im_ht, im_w, _ = composite_image.shape
composite_image = composite_image[250:im_ht - 250, 0:im_w-250]

cv2.imshow('../data/Panaroma_home.jpg', composite_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Bibliography

Referenced the following sources

- Slideset used in class
- https://www.youtube.com/@firstprinciplesofcomputerv3258
- OpenCV tutorials on GeeksForGeeks, OpenCV documentation, skimage documentation

Collaborated with

- Abhishek Iyer
- Prajwal Gurunath
- Aayush Fadia
- Sreeharsha Paruchari